Jürgen Becker
Marco Platzner
Serge Vernalde (Eds.)

LNCS 3203

Field-Programmable Logic and Applications

14th International Conference, FPL 2004
Antwerp, Belgium, August/September 2004
Proceedings

Springer

Lecture Notes in Computer Science 3203

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Jürgen Becker Marco Platzner
Serge Vernalde (Eds.)

Field-Programmable Logic and Applications

14th International Conference, FPL 2004
Antwerp, Belgium, August 30–September 1, 2004
Proceedings

Springer

Volume Editors

Jürgen Becker
Universität Karlsruhe (TH)
Institut für Technik der Informationsverabeitung (ITIV)
Engesserstr. 5, 76128 Karlsruhe, Germany
E-mail: becker@itiv.uni-karlsruhe.de

Marco Platzner
Swiss Federal Institute of Technology (ETH) Zurich
Computer Engineering and Networks Lab
Gloriastr. 35, 8092 Zurich Switzerland
E-mail: marco.platzner@computer.org

Serge Vernalde
IMEC vzw
Kapeldreef 75, 3001 Leuven, Belgium
E-mail: serge.vernalde@imec.be

Library of Congress Control Number: 2004111058

CR Subject Classification (1998): B.6-7, C.2, J.6

ISSN 0302-9743
ISBN 3-540-22989-2 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 11311713 06/3142 5 4 3 2 1 0

Preface

This book contains the papers presented at the 14th International Conference on Field Programmable Logic and Applications (FPL) held during August 30th–September 1st 2004. The conference was hosted by the Interuniversity Micro-Electronics Center (IMEC) in Leuven, Belgium.

The FPL series of conferences was founded in 1991 at Oxford University (UK), and has been held annually since: in Oxford (3 times), Vienna, Prague, Darmstadt, London, Tallinn, Glasgow, Villach, Belfast, Montpellier and Lisbon. It is the largest and oldest conference in reconfigurable computing and brings together academic researchers, industry experts, users and newcomers in an informal, welcoming atmosphere that encourages productive exchange of ideas and knowledge between the delegates.

The fast and exciting advances in field programmable logic are increasing steadily with more and more application potential and need. New ground has been broken in architectures, design techniques, (partial) run-time reconfiguration and applications of field programmable devices in several different areas. Many of these recent innovations are reported in this volume.

The size of the FPL conferences has grown significantly over the years. FPL in 2003 saw 216 papers submitted. The interest and support for FPL in the programmable logic community continued this year with 285 scientific papers submitted, demonstrating a 32% increase when compared to the year before. The technical program was assembled from 78 selected regular papers, 45 additional short papers and 29 posters, resulting in this volume of proceedings. The program also included three invited plenary keynote presentations from Xilinx, Gilder Technology Report and Altera, and three embedded tutorials from Xilinx, the Universität Karlsruhe (TH) and the University of Oslo.

Due to the inclusive tradition of the conference, FPL continues to attract submissions from all over the world. The accepted contributions were submitted by researchers from 24 different countries:

USA	37	Canada	6	Netherlands	3	Mexico	2
Spain	21	Portugal	6	Austria	2	Switzerland	2
Germany	20	Brazil	5	Belgium	2	Australia	1
UK	11	Finland	3	Czechia	2	China	1
Japan	9	Ireland	3	Greece	2	Estonia	1
France	7	Poland	3	Italy	2	Lebanon	1

We would like to thank all the authors for submitting their first versions of the papers and the final versions of the accepted papers. We also gratefully acknowledge the tremendous reviewing work done by the Program Committee members and many additional reviewers who contributed their time and expertise towards the compilation of this volume. We would also like to thank the members of the Organizing Committee for their competent guidance and work in the last month. Especially, we acknowledge the assistance of Michael Hübner and Oliver Sander from Universität Karlsruhe (TH) and Rolf Enzler from ETH Zurich in compiling the final program. The members of our Program and Organizing Committees as well as all other reviewers are listed on the following pages.

We would like to thank Altera, Synplicity and Xilinx for their sponsorships. We are indebted to Richard van de Stadt, the author of CyberChair. This extraordinary free software made our task of managing the submission and reviewing process much easier.

We are grateful to Springer-Verlag, particularly Alfred Hofmann, for his work in publishing this book.

June 2004

Jürgen Becker
Marco Platzner
Serge Vernalde

Organization

Organizing Committee

General Chair	Serge Vernalde, IMEC vzw, Belgium
Program Chair	Jürgen Becker, Universität Karlsruhe, Germany
Ph.D.-Forum Chair	Jürgen Teich, University of Erlangen-Nuremberg, Germany
Publicity Chair	Reiner Hartenstein, University of Kaiserslautern, Germany
Proceedings Chair	Marco Platzner, ETH Zurich, Switzerland
Exhibition/Sponsors Chair	Erik Watzeels, IMEC vzw, Belgium
Finance Chair	Diederik Verkest, IMEC vzw, Belgium
Local Arrangements Chair	Annemie Stas, IMEC vzw, Belgium

Program Committee

Nazeeh Aranki	Jet Propulsion Laboratory, USA
Jeff Arnold	Stretch, Inc., USA
Peter Athanas	Virginia Tech, USA
Jürgen Becker	Universität Karlsruhe, Germany
Neil Bergmann	Queensland University of Technology, Australia
Dinesh Bhatia	University of Texas, USA
Eduardo Boemo	Universidad Autonoma de Madrid, Spain
Gordon Brebner	Xilinx, Inc., USA
Andrew Brown	University of Southampton, UK
Klaus Buchenrieder	Infineon Technologies AG, Germany
João Cardoso	University of the Algarve, Portugal
Steve Casselman	Virtual Computer Corporation, USA
Peter Cheung	Imperial College London, UK
George Constantinides	Imperial College London, UK
Carl Ebeling	University of Washington, USA
Hossam ElGindy	University of New South Wales, Australia
Manfred Glesner	Darmstadt University of Technology, Germany
Fernando Gonçalves	Technical University of Lisbon, Portugal
Steven Guccione	Quicksilver Technology, USA
Reiner Hartenstein	University of Kaiserslautern, Germany
Scott Hauck	University of Washington, USA
Tom Kean	Algotronix Consulting, UK
Andreas Koch	University of Braunschweig, Germany
Dominique Lavenier	University of Montpellier II, France
Philip Leong	Chinese University of Hong Kong, China
Wayne Luk	Imperial College London, UK
Patrick Lysaght	Xilinx, Inc., USA
Oskar Mencer	Imperial College London, UK
Toshiyaki Miyazaki	NTT Network Innovation Labs, Japan
Fernando Moraes	PUCRS, Brazil
Klaus Müller-Glaser	Universität Karlsruhe, Germany
Brent Nelson	Brigham Young University, USA
Horácio Neto	Technical University of Lisbon, Portugal
Sebastien Pillement	ENSSAT, France
Marco Platzner	ETH Zurich, Switzerland
Viktor Prasanna	University of Southern California, USA
Franz Rammig	Universität Paderborn, Germany
Ricardo Reis	Universidade Federal do Rio Grande do Sul, Brazil
Jonathan Rose	University of Toronto, Canada
Zoran Salcic	University of Auckland, New Zealand
Sergej Sawitzki	Philips Research, The Netherlands
Hartmut Schmeck	University of Karlsruhe, Germany
Subarna Sinha	Synopsys, Inc., USA

Gerard Smit	University of Twente, The Netherlands
Jose T. de Sousa	Technical University of Lisbon, Portugal
Rainer Spallek	Dresden University of Technology, Germany
Adrian Stoica	Jet Propulsion Laboratory, USA
Jürgen Teich	University of Erlangen-Nuremberg, Germany
Lothar Thiele	ETH Zurich, Switzerland
Lionel Torres	University of Montpellier II, France
Nick Tredennick	Gilder Technology Report, USA
Stephen Trimberger	Xilinx, Inc., USA
Milan Vasilko	Bournemouth University, UK
Stamatis Vassiliadis	Delft University of Technology, The Netherlands
Ranga Vemuri	University of Cincinnati, USA
Serge Vernalde	IMEC vzw, Belgium
Martin Vorbach	PACT Informationstechnologie, Germany
Steve Wilton	University of British Columbia, Canada
Roger Woods	Queen's University Belfast, UK
Andrej Zemva	University of Ljubljana, Slovenia

Steering Committee

Jose T. de Sousa	Technical University of Lisbon, Portugal
Manfred Glesner	Darmstadt University of Technology, Germany
John Gray	Independent Consultant, UK
Herbert Grünbacher	Carinthia Technical Institute, Austria
Reiner Hartenstein	University of Kaiserslautern, Germany
Andres Keevallik	Tallinn Technical University, Estonia
Wayne Luk	Imperial College London, UK
Patrick Lysaght	Xilinx, Inc., USA
Michel Renovell	University of Montpellier II, France
Roger Woods	Queen's University Belfast, UK

Additional Reviewers

Waleed Abdulla
Ali Ahmadinia
Kupriyanov Alexey
Iosif Antochi
Chris Assad
Liping Bai
Zachary Baker
Marcelo Barcelos
Michael Beauchamp
Marcus Bednara
Stevan Berber
Jean-Luc Beuchat
Eduardo Augusto Bezerra
Carsten Bieser
Abbas Bigdeli
Morteza Biglari-Abhari
Christophe Bobda
Peter Boll
Matthias Bonn
Christos-Savvas Bouganis
Ney Calazans
Humberto Calderon
Nicola Campregher
Joao Paulo Carvalho

Ewerson Luiz de Souza Carvalho
Bryan Catanzaro
Mark L. Chang
Francois Charot
C.C. Cheung
Charles Chiang
Daniel Chillet
Seonil Choi
Tim Courtney
G. Figueiredo Coutinho
S. Craven
Dan Crisu
Miro Cupak
Raphael David
Joze Dedic
A. Derbyshire
Steven Derrien
Dirk Desmet
Florent de Dinechin
Matthias Dyer
Ilos Eix
Rolf Enzler
C.T. Ewe
Erwan Fabiani
Suhaib Fahmy
George Ferizis
A. Fidjeland
Matjaz Finc
Robert Fischer
A. Abdul Gaffar
Carlo Galuzzi
Georgi N. Gaydadjiev
Gokul Govindu
Michael Guntsch
Said Hamdioui
Manish Handa
Frank Hannig
Michael Haselman
Christian Haubelt
Michael Hübner
Fabiano Hessel
Samih Hijwel
Clint Hilton
Th. Hinze
Christian Hochberger
Mark Holland
Thomas Hollstein
Zhedong Hua
Renqiu Huang
Finn Hughes
Jung Hyun Choi
Leandro Soares Indrusiak
Hideyuki Ito
Xin Jia
Eric Johnson
Ralf König
Thorsten Köster
Heiko Kalte
F. Gusmão de Lima Kastensmidt
Jamil Kawa
Ronan Keryell
Jawad Khan
Mohammed Ghiath Khatib
Kichul Kim
Dirk Koch
Franci Kopac
Georgi Kuzmanov
Julien Lamoureux
Pepijn de Langen
Dong-U Lee
Gareth Lee
Seokjin Lee
Yujie Lee
Stan Liao
Yang Liu
Marcelo Lubaszewski
Ralf Ludewig
A. Mahar
Mateusz Majer
Usama Malik
André Malz
Freddy Mang
César Augusto Missio Marcon
Theodore Marescaux
John McAllister
Wim J. C. Melis
Aline Vieira de Mello
Jean-Yves Mignolet
Will Moffat
Sumit Mohanty

Leandro Heleno Möller
Anca Molnos
Edson Ifarraguirre Moreno
Carlos Morra
Gareth Morris
Elena Moscu Panainte
Erdem Motuk
Tudor Murgan
Takahiro Murooka
Kouichi Nagami
Yoshiki Nakane
Vincent Nollet
Juan Jesus Ocampo-Hidalgo
Yuichi Okuyama
Luciano Copello Ost
Jingzhao Ou
K. Paar
José Carlos Sant'Anna Palma
Joseph Palmer
Alex Panato
Sujan Pandey
Mihail Petrov
Jean-Marc Philippe
Shawn Phillips
Thilo Pionteck
Christian Plessl
Kara Poon
Adam Postula
Bernard Pottier
Brad Quinton
Frédéric Raimbault
M. Rans
Rainer Rawer
Darren Reilly
T. Rissa
Pasko Robert
Partha Roop
Andrew Royal
Stéphane Rubini
Luc Rynders
Oliver Sander
Gilles Sassatelli
Bernd Scheuermann
Clemens Schlachta
Joerg Schneider
Ronald Scrofano
Pete Sedcole
Olivier Sentieys
Balasubramanian Sethuraman
A. Shahbahrami
Akshay Sharma
Nalin Sidahao
Reetinder Sidhu
Ivan Saraiva Silva
Mihai Sima
Alastair Smith
Keith So
O. Soffke
Raphael Some
Galileu Batista de Sousa
N. Steiner
Piotr Stepien
S. Stone
Thilo Streichert
Qing Su
Peter Sutton
Alex Thomas
D. Thomas
Andrej Trost
Richard Turner
Michael Ullmann
G. Vanmeerbeeck
Matjaz Verderber
François Verdier
Carlos Y Villalpando
Patrick Vollmer
Herbert Walder
Jian Wang
Xin Wang
John Williams
Laurent Wojcieszak
Christophe Wolinski
Stephan Wong
James Wu
Yunfei Wu
Andy Yan
Jenny Yi-Chun Kua
Jae Young Hur
Frank Zee
Cong Zhang

Ling Zhuo
Daniel Ziener
Heiko Zimmer
Peter Zipf

Table of Contents

Plenary Keynotes

FPGAs and the Era of Field Programmability 1
W. Roelandts

Reconfigurable Systems Emerge 2
N. Tredennick, B. Shimamoto

System-Level Design Tools Can Provide Low Cost Solutions in FPGAs: TRUE or FALSE? 12
M. Dickinson

Organic and Biology Computing

Hardware Accelerated Novel Protein Identification 13
A. Alex, J. Rose, R. Isserlin-Weinberger, C. Hogue

Large Scale Protein Sequence Alignment Using FPGA Reprogrammable Logic Devices 23
S. Dydel, P. Bała

Security and Cryptography 1

A Key Management Architecture for Securing Off-Chip Data Transfers 33
J. Graf, P. Athanas

FPGA Implementation of Biometric Authentication System Based on Hand Geometry 43
C. López-Ongil, R. Sanchez-Reillo, J. Liu-Jimenez, F. Casado, L. Sánchez, L. Entrena

Platform Based Design

SoftSONIC: A Customisable Modular Platform for Video Applications . . . 54
T. Rissa, P.Y.K. Cheung, W. Luk

Deploying Hardware Platforms for SoC Validation: An Industrial Case Study 64
A. Bigot, F. Charpentier, H. Krupnova, I. Sans

Algorithms and Architectures

Algorithms and Architectures for Use in FPGA Implementations of Identity Based Encryption Schemes 74
T. Kerins, E. Popovici, W. Marnane

Power Analysis Attacks Against FPGA Implementations of the DES 84
F.-X. Standaert, S.B. Örs, J.-J. Quisquater, B. Preneel

Acceleration Application 1

Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer 95
M. Gokhale, J. Frigo, C. Ahrens, J.L. Tripp, R. Minnich

Stochastic Simulation for Biochemical Reactions on FPGA 105
M. Yoshimi, Y. Osana, T. Fukushima, H. Amano

Architecture 1

Dynamic Adaptive Runtime Routing Techniques in Multigrain Reconfigurable Hardware Architectures 115
A. Thomas, J. Becker

Interconnecting Heterogeneous Nodes in an Adaptive Computing Machine 125
F. Furtek, E. Hogenauer, J. Scheuermann

Improving FPGA Performance and Area Using an Adaptive Logic Module 135
M. Hutton, J. Schleicher, D. Lewis, B. Pedersen, R. Yuan, S. Kaptanoglu, G. Baeckler, B. Ratchev, K. Padalia, M. Bourgeault, A. Lee, H. Kim, R. Saini

A Dual-V$_{DD}$ Low Power FPGA Architecture 145
A. Gayasen, K. Lee, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, T. Tuan

Physical Design 1

Simultaneous Timing Driven Clustering and Placement for FPGAs 158
G. Chen, J. Cong

Run-Time-Conscious Automatic Timing-Driven FPGA Layout Synthesis 168
J. Anderson, S. Nag, K. Chaudhary, S. Kalman, C. Madabhushi, P. Cheng

Compact Buffered Routing Architecture 179
A. Lodi, R. Giansante, C. Chiesa, L. Ciccarelli, F. Campi, M. Toma

On Optimal Irregular Switch Box Designs 189
H. Fan, Y.-L. Wu, C.-C. Cheung, J. Liu

Arithmetic 1

Dual Fixed-Point:
An Efficient Alternative to Floating-Point Computation 200
C.T. Ewe, P.Y.K. Cheung, G.A. Constantinides

Comparative Study of SRT-Dividers in FPGA 209
G. Sutter, G. Bioul, J.-P. Deschamps

Second Order Function Approximation
Using a Single Multiplication on FPGAs 221
J. Detrey, F. de Dinechin

Efficient Modular Division Implementation
(ECC over GF(p) Affine Coordinates Application) 231
G. Meurice de Dormale, P. Bulens, J.-J. Quisquater

Multitasking

A Low Fragmentation Heuristic for Task Placement
in 2D RTR HW Management 241
J. Tabero, J. Septién, H. Mecha, D. Mozos

The Partition into Hypercontexts Problem
for Hyperreconfigurable Architectures 251
S. Lange, M. Middendorf

Circuit Technology

A High-Density Optically Reconfigurable Gate Array
Using Dynamic Method 261
M. Watanabe, F. Kobayashi

Evolvable Hardware for Signal Separation and Noise Cancellation
Using Analog Reconfigurable Device 270
D. Keymeulen, R. Zebulum, A. Stoica, V. Duong, M.I. Ferguson

Memory 1

Implementing High-Speed Double-Data Rate (DDR) SDRAM Controllers on FPGA 279
E. Picatoste-Olloqui, F. Cardells-Tormo, J. Sempere-Agullo, A. Herms-Berenguer

Logic Modules with Shared SRAM Tables for Field-Programmable Gate Arrays 289
F. Kocan, J. Meyer

Network Processing

A Modular System for FPGA-Based TCP Flow Processing in High-Speed Networks 301
D.V. Schuehler, J.W. Lockwood

Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs 311
Z.K. Baker, V.K. Prasanna

Testing

BIST Based Interconnect Fault Location for FPGAs 322
N. Campregher, P.Y.K. Cheung, M. Vasilko

FPGAs BIST Evaluation 333
A. Parreira, J.P. Teixeira, M.B. Santos

Applications

Solving SAT with a Context-Switching Virtual Clause Pipeline and an FPGA Embedded Processor 344
C.J. Tavares, C. Bungardean, G.M. Matos, J.T. de Sousa

Evaluating Fault Emulation on FPGA 354
P. Ellervee, J. Raik, V. Tihhomirov, K. Tammemäe

Arithmetic 2

Automating Optimized Table-with-Polynomial Function Evaluation for FPGAs 364
D.-U. Lee, O. Mencer, D.J. Pearce, W. Luk

Multiple Restricted Multiplication 374
N. Sidahao, G.A. Constantinides, P.Y.K. Cheung

Signal Processing 1

Area*Time Optimized Hogenauer Channelizer Design Using FPL Devices 384
U. Meyer-Bäse, S. Rao, J. Ramírez, A. García

A Steerable Complex Wavelet Construction and Its Implementation on FPGA 394
C.-S. Bouganis, P.Y.K. Cheung, J. Ng, A.A. Bharath

Computational Models and Compiler

Programmable Logic Has More Computational Power than Fixed Logic 404
G. Brebner

JHDLBits: The Merging of Two Worlds 414
A. Poetter, J. Hunter, C. Patterson, P. Athanas, B. Nelson, N. Steiner

A System Level Resource Estimation Tool for FPGAs 424
C. Shi, J. Hwang, S. McMillan, A. Root, V. Singh

The PowerPC Backend Molen Compiler 434
E. Moscu Panainte, K. Bertels, S. Vassiliadis

Dynamic Reconfiguration 1

An Integrated Online Scheduling and Placement Methodology 444
M. Handa, R. Vemuri

On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities 454
M. Ullmann, M. Hübner, B. Grimm, J. Becker

Techniques for Virtual Hardware on a Dynamically Reconfigurable Processor – An Approach to Tough Cases – 464
H. Amano, T. Inuo, H. Kami, T. Fujii, M. Suzuki

Throughput and Reconfiguration Time Trade-Offs: From Static to Dynamic Reconfiguration in Dedicated Image Filters 474
M.R. Boschetti, S. Bampi, I.S. Silva

Network and Optimization Algorithms

Over 10Gbps String Matching Mechanism for Multi-stream Packet Scanning Systems 484
Y. Sugawara, M. Inaba, K. Hiraki

Hardware Design of a FPGA-Based Synchronizer for Hiperlan/2 494
M.J. Canet, F. Vicedo, V. Almenar, J. Valls, E.R. de Lima

Three-Dimensional Dynamic Programming for Homology Search 505
Y. Yamaguchi, T. Maruyama, A. Konagaya

An Instance-Specific Hardware Algorithm
for Finding a Maximum Clique 516
S. Wakabayashi, K. Kikuchi

System-on-Chip 1

IP Generation for an FPGA-Based
Audio DAC Sigma-Delta Converter 526
*R. Ludewig, O. Soffke, P. Zipf, M. Glesner, K.P. Pun,
K.H. Tsoi, K.H. Lee, P. Leong*

Automatic Creation of Reconfigurable PALs/PLAs for SoC 536
M. Holland, S. Hauck

High Speed Design

A Key Agile 17.4 Gbit/sec Camellia Implementation 546
D. Denning, J. Irvine, M. Devlin

High Performance True Random Number Generator
in Altera Stratix FPLDs 555
V. Fischer, M. Drutarovský, M. Šimka, N. Bochard

Security and Cryptography 2

A Universal and Efficient AES Co-processor
for Field Programmable Logic Arrays 565
N. Pramstaller, J. Wolkerstorfer

Exploring Area/Delay Tradeoffs in an AES FPGA Implementation 575
J. Zambreno, D. Nguyen, A. Choudhary

Architectures 2

Reconfigurable Instruction Set Extension for Enabling ECC
on an 8-Bit Processor 586
S. Kumar, C. Paar

Memory 2

Dynamic Prefetching in the Virtual Memory Window of Portable Reconfigurable Coprocessors 596
M. Vuletić, L. Pozzi, P. Ienne

Storage Allocation for Diverse FPGA Memory Specifications 606
D. Dagher, I. Ouaiss

Image Processing 1

Real Time Optical Flow Processing System 617
J. Díaz, E. Ros, S. Mota, R. Carrillo, R. Agis

Methods and Tools for High-Resolution Imaging 627
T. Todman, W. Luk

Network-on-Chip

Network-on-Chip for Reconfigurable Systems: From High-Level Design Down to Implementation 637
T.A. Bartic, D. Desmet, J.-Y. Mignolet, T. Marescaux, D. Verkest, S. Vernalde, R. Lauwereins, J. Miller, F. Robert

A Reconfigurable Recurrent Bitonic Sorting Network for Concurrently Accessible Data 648
C. Layer, H.-J. Pfleiderer

Power Aware Design 1

A Framework for Energy Efficient Design of Multi-rate Applications Using Hybrid Reconfigurable Systems 658
S. Mohanty, V.K. Prasanna

An Efficient Battery-Aware Task Scheduling Methodology for Portable RC Platforms 669
J. Khan, R. Vemuri

IP-Based Design

HW/SW Co-design by Automatic Embedding of Complex IP Cores 679
H. Lange, A. Koch

Increasing Pipelined IP Core Utilization in Process Networks Using Exploration 690
C. Zissulescu, B. Kienhuis, E. Deprettere

Distribution of Bitstream-Level IP Cores for Functional Evaluation Using FPGAs 700
R. Siripokarpirom

SOC and RTOS: Managing IPs and Tasks Communications 710
A. Segard, F. Verdier

Power Aware Design 2

The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays 719
S.J.E. Wilton, S.-S. Ang, W. Luk

A Methodology for Energy Efficient FPGA Designs Using Malleable Algorithms 729
J. Ou, V.K. Prasanna

Power-Driven Design Partitioning 740
R. Mukherjee, S.O. Memik

Power Consumption Reduction Through Dynamic Reconfiguration 751
M.G. Lorenz, L. Mengibar, M.G. Valderas, L. Entrena

Coprocessing Architectures

The XPP Architecture and Its Co-simulation Within the Simulink Environment 761
M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, M. Glesner

An FPGA Based Coprocessor for the Classification of Tissue Patterns in Prostatic Cancer 771
M.A. Tahir, A. Bouridane, F. Kurugollu

Increasing ILP of RISC Microprocessors Through Control-Flow Based Reconfiguration 781
S. Köhler, J. Braunes, T. Preußer, M. Zabel, R.G. Spallek

Using of FPGA Coprocessor for Improving the Execution Speed of the Pattern Recognition Algorithm for ATLAS – High Energy Physics Experiment 791
C. Hinkelbein, A. Khomich, A. Kugel, R. Männer, M. Müller

Embedded Tutorials

Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs..... 801
B. Blodget, C. Bobda, M. Huebner, A. Niyonkuru

SystemC for the Design and Modeling of Programmable Systems........ 811
A. Donlin, A. Braun, A. Rose

An Evolvable Hardware Tutorial 821
J. Torresen

Dynamic Reconfiguration 2

A Runtime Environment
for Reconfigurable Hardware Operating Systems 831
H. Walder, M. Platzner

A Dynamically Reconfigurable
Asynchronous FPGA Architecture 836
X. Jia, J. Rajagopalan, R. Vemuri

Hardware Support for Dynamic Reconfiguration in Reconfigurable
SoC Architectures.. 842
B. Griese, E. Vonnahme, M. Porrmann, U. Rückert

Physical Design 2

Optimal Routing-Conscious Dynamic Placement
for Reconfigurable Devices 847
A. Ahmadinia, C. Bobda, S.P. Fekete, J. Teich, J.C. van der Veen

Optimizing the Performance of the Simulated Annealing
Based Placement Algorithms for Island-Style FPGAs 852
A. Danilin, S. Sawitzki

Automating the Layout of Reconfigurable Subsystems
via Template Reduction 857
S. Phillips, A. Sharma, S. Hauck

Acceleration Application 2

FPGA Acceleration of Rigid Molecule Interactions 862
T. Van Court, Y. Gu, M. Herbordt

Mapping DSP Applications to a High-Performance Reconfigurable
Coarse-Grain Data-Path 868
M.D. Galanis, G. Theodoridis, S. Tragoudas, D. Soudris, C.E. Goutis

Exploring Potential Benefits of 3D FPGA Integration................. 874
C. Ababei, P. Maidee, K. Bazargan

System Level Design

System-Level Modeling of Dynamically Reconfigurable Co-processors 881
Y. Qu, K. Tiensyrjä, K. Masselos

A Development Support System for Applications That Use
Dynamically Reconfigurable Hardware 886
J. Canas Ferreira, J. Silva Matos

Physical Interconnect

Interconnect-Aware Mapping of Applications to Coarse-Grain Reconfigurable Architectures 891
N. Bansal, S. Gupta, N. Dutt, A. Nicolau, R. Gupta

Analysis of a Hybrid Interconnect Architecture for Dynamically Reconfigurable FPGAs 900
R. Huang, M. Handa, R. Vemuri

Computational Models

Mapping Basic Recursive Structures to Runtime Reconfigurable Hardware 906
H. ElGindy, G. Ferizis

Implementation of the Extended Euclidean Algorithm for the Tate Pairing on FPGA 911
T. Ito, Y. Shibata, K. Oguri

Acceleration Applications 3

Java Technology in an FPGA 917
M. Schoeberl

Hardware/Software Implementation of FPGA-Targeted Matrix-Oriented SAT Solvers 922
V. Sklyarov, I. Skliarova, B. Pimentel, J. Arrais

The Chess Monster Hydra 927
C. Donninger, U. Lorenz

Arithmetic 3

FPGA-Efficient Hybrid LUT/CORDIC Architecture 933
I. Janiszewski, H. Meuth, B. Hoppe

A Multiplexer-Based Concept for Reconfigurable Multiplier Arrays 938
O.A. Pfänder, R. Hacker, H.-J. Pfleiderer

Design and Implementation of a CFAR Processor for Target Detection 943
C. Torres-Huitzil, R. Cumplido-Parra, S. López-Estrada

Signal Processing 2

A Parallel FFT Architecture for FPGAs 948
J. Palmer, B. Nelson

FPGA Custom DSP for ECG Signal Analysis and Compression 954
M.M. Peiró, F. Ballester, G. Paya, J. Belenguer, R. Colom, R. Gadea

FPGA Implementation of Adaptive Multiuser Detector for DS-CDMA Systems 959
Q.-T. Ho, D. Massicotte

System-on-Chip 2

Simulation Platform for Architectural Verification and Performance Analysis in Core-Based SoC Design 965
U. Bidarte, A. Astarloa, J.L. Martín, J. Andreu

A Low Power FPAA for Wide Band Applications 970
E. Schüler, L. Carro

Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs 975
E.L. Horta, J.W. Lockwood

Image Processing 2

Real-Time Computation of the Generalized Hough Transform 980
T. Maruyama

Minimum Sum of Absolute Differences Implementation in a Single FPGA Device 986
J. Olivares, J. Hormigo, J. Villalba, I. Benavides

Design and Efficient FPGA Implementation of an RGB to YCrCb Color Space Converter Using Distributed Arithmetic 991
F. Bensaali, A. Amira

Cryptography and Compression

High Throughput Serpent Encryption Implementation 996
J. Lázaro, A. Astarloa, J. Arias, U. Bidarte, C. Cuadrado

Implementation of Elliptic Curve Cryptosystems over GF(2^n) in Optimal Normal Basis on a Reconfigurable Computer 1001
S. Bajracharya, C. Shu, K. Gaj, T. El-Ghazawi

Wavelet-Based Image Compression on the Reconfigurable Computer ACE-V 1006
H. Gädke, A. Koch

Network Applications and Architectures

A Reconfigurable Communication Processor Compatible with Different Industrial Fieldbuses 1011
M.D. Valdés, M.Ángel Domínguez, M.J. Moure, C. Quintáns

Multithreading in a Hyper-programmable Platform for Networked Systems 1017
P. James-Roxby, G. Brebner

An Environment for Exploring Data-Driven Architectures 1022
R. Ferreira, J.M.P. Cardoso, H.C. Neto

FPGA Implementation of a Novel All Digital PLL Architecture for PCR Related Measurements in DVB-T 1027
C. Mannino, H. Rabah, C. Tanougast, Y. Berviller, M. Janiaut, S. Weber

Network on Chip and Adaptive Architectures

A Dynamic NoC Approach for Communication in Reconfigurable Devices 1032
C. Bobda, M. Majer, D. Koch, A. Ahmadinia, J. Teich

Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems 1037
M. Huebner, M. Ullmann, L. Braun, A. Klausmann, J. Becker

FiPRe: An Implementation Model to Enable Self-Reconfigurable Applications 1042
L. Möller, N. Calazans, F. Moraes, E. Brião, E. Carvalho, D. Camozzato

A Structured Methodology for System-on-an-FPGA Design 1047
P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, W. Luk

Debugging and Test

Secure Logic Synthesis 1052
K. Tiri, I. Verbauwhede

Hardware and Software Debugging of FPGA Based Microprocessor Systems Through Debug Logic Insertion 1057
M.G. Valderas, E. del Torre, F. Ariza, T. Riesgo

The Implementation of a FPGA Hardware Debugger System with Minimal System Overhead 1062
J. Tombs, M.A.A. Echanóve, F. Muñoz, V. Baena, A. Torralba, A. Fernandez-León, F. Tortosa

Optimization of Testability of Sequential Circuits Implemented in FPGAs with Embedded Memory 1067
A. Krasniewski

Organic and Biology Computing (Poster)

FPGA Implementation of a Neuromimetic Cochlea for a Bionic Bat Head 1073
C. Clarke, L. Qiang, H. Peremans, Á. Hernández

FPGA-Based Computation for Maximum Likelihood Phylogenetic Tree Evaluation 1076
T.S.T. Mak, K.P. Lam

Processing Repetitive Sequence Structures with Mismatches at Streaming Rate 1080
A.A. Conti, T. Van Court, M.C. Herbordt

Artificial Neural Networks Processor – A Hardware Implementation Using a FPGA 1084
P. Ferreira, P. Ribeiro, A. Antunes, F.M. Dias

FPGA Implementation of the Ridge Line Following Fingerprint Algorithm 1087
E. Cantó, N. Canyellas, M. Fons, F. Fons, M. López

Security and Cryptography (Poster)

A Dynamically Reconfigurable Function-Unit for Error Detection and Correction in Mobile Terminals 1090
T. Pionteck, T. Stiefmeier, T.R. Staake, M. Glesner

Flow Monitoring in High-Speed Networks with 2D Hash Tables 1093
D. Nguyen, J. Zambreno, G. Memik

A VHDL Generator for Elliptic Curve Cryptography 1098
K. Järvinen, M. Tommiska, J. Skyttä

FPGA-Based Parallel Comparison of Run-Length-Encoded Strings 1101
A. Bogliolo, V. Freschi, F. Miglioli, M. Canella

Real Environments Image Labelling Based on Reconfigurable Architectures 1104
J.M. García Chamizo, A. Fuster Guilló, J. Azorín López

Mapping and Compilers (Poster)

Object Oriented Programming Paradigms for the VHDL 1107
J. Borgosz

Using Reconfigurable Hardware Through Web Services in Distributed Applications 1110
I. Gonzalez, J. Sanchez-Pastor, J.L. Hernandez-Ardieta, F.J. Gomez-Arribas, J. Martinez

Data Reuse in Configurable Architectures with RAM Blocks 1113
N. Baradaran, J. Park, P.C. Diniz

A Novel FPGA Configuration Bitstream Generation Algorithm and Tool Development 1116
K. Siozios, G. Koutroumpezis, K. Tatas, D. Soudris, A. Thanailakis

AAA and SynDEx-Ic: A Methodology and a Software Framework for the Implementation of Real-Time Applications onto Reconfigurable Circuits 1119
P. Niang, T. Grandpierre, M. Akil, Y. Sorel

Architectures (Poster)

A Self-Reconfiguration Framework for Multiprocessor CSoPCs 1124
A. Astarloa, J. Lázaro, U. Bidarte, J.L. Martín, A. Zuloaga

A Virtual File System for Dynamically Reconfigurable FPGAs 1127
A. Donlin, P. Lysaght, B. Blodget, G. Troeger

Virtualizing the Dimensions of a Coarse-Grained Reconfigurable Array 1130
T. Ristimäki, J. Nurmi

Design and Implementation of the Memory Scheduler for the PC-Based Router 1133
T. Marek, M. Novotný, L. Crha

Algorithms and IP (Poster)

Analog Signal Processing Reconfiguration for Systems-on-Chip Using a Fixed Analog Cell Approach 1136
E.E. Fabris, L. Carro, S. Bampi

Intellectual Property Protection for RNS Circuits on FPGAs 1139
L. Parrilla, E. Castillo, A. García, A. Lloris

FPGA Implementation of a Tool Breakage Detection Algorithm in CNC Milling Machines 1142
R. deJ. Romero-Troncoso, G.H. Ruiz

Implementation of a 3-D Switching Median Filtering Scheme with an Adaptive LUM-Based Noise Detector 1146
M. Drutarovský, V. Fischer

Using Logarithmic Arithmetic to Implement the Recursive Least Squares (QR) Algorithm in FPGA 1149
J. Schier, A. Heřmánek

Image Processing (Poster)

FPGA Implementation of a Vision-Based Motion Estimation Algorithm for an Underwater Robot 1152
V. Ila, R. Garcia, F. Charot, J. Batlle

Real-Time Detection of Moving Objects 1155
H. Niitsuma, T. Maruyama

Real-Time Visual Motion Detection of Overtaking Cars for Driving Assistance Using FPGAs 1158
S. Mota, E. Ros, J. Díaz, E.M. Ortigosa, R. Agis, R. Carrillo

Versatile Imaging Architecture Based on a System on Chip 1162
P. Chalimbaud, F. Berry

A Hardware Implementation of a Content Based Image Retrieval Algorithm 1165
C. Skarpathiotis, K.R. Dimond

PhD Forum (Poster)

Optimization Algorithms for Dynamic Reconfigurable Embedded Systems 1168
A. Ahmadinia

Low Power Reconfigurable Devices 1169
A. Gayasen

Code Re-ordering for a Class of Reconfigurable Microprocessors 1170
B.F. Veale, J.K. Antonio, M.P. Tull

Design Space Exploration for Distributed Hardware Reconfigurable Systems 1171
C. Haubelt

TPR: Three-D Place and Route for FPGAs 1172
C. Ababei

Implementing Graphics Shaders Using FPGAs 1173
D.B. Thomas, W. Luk

Preemptive Hardware Task Management 1174
D. Koch

Automated Speculation and Parallelism
in High Performance Network Applications 1175
G. Schelle, D. Grunwald

Automated Mapping of Coarse-Grain Pipelined Applications
to FPGA Systems 1176
H.E. Ziegler

A Specific Scheduling Flow
for Dynamically Reconfigurable Hardware 1178
J. Resano

Design and Evaluation of an FPGA Architecture
for Software Protection 1180
J. Zambreno

Scalable Defect Tolerance Beyond the SIA Roadmap 1181
M. Mishra

Run-Time Reconfiguration Management
for Adaptive High-Performance Computing Systems 1183
M. Taher, T. El-Ghazawi

Optimized Field Programmable Gate Array Based
Function Evaluation 1184
N. Sidahao

MemMap-pd: Performance Driven Technology Mapping Algorithm
for FPGAs with Embedded Memory Blocks 1185
R. Manimegalai, A. ManojKumar, B. Jayaram, V. Kamakoti

A System on Chip Design Framework
for Prime Number Validation Using Reconfigurable Hardware 1186
R.C.C. Cheung

On Computing Maximum Likelihood Phylogeny Using FPGA 1188
T.S.T. Mak, K.P. Lam

Minimising Reconfiguration Overheads in Embedded Applications 1189
U. Malik

Application Specific Small-Scale Reconfigurability 1190
V.V. Kumar

Efficient FPGA-Based Security Kernels 1191
Z.K. Baker

Author Index 1193

FPGAs and the Era of Field Programmability

Wim Roelandts

Xilinx Inc,
2100 Logic Drive,
San Jose, CA.

Abstract. The progress of the semiconductor industry over the last several decades has been described as a series of alternating cycles of standardization and customization. According to Makimoto's wave, as the model is known, we are now in an era of field programmability. This cycle is characterized by standardization in manufacturing and customization in application. The application drivers for this second digital wave are networking, digital consumer electronics and the convergence of communications and computing technologies.

Xilinx has been the foremost company in field programmable logic since inventing the FPGA over twenty years ago. Today, Xilinx FPGAs are the key technology enabling the era of programmability. From this vantage point, we begin this talk by surveying the key technologies that define the state-of-the-art in field programmable logic. We proceed to explore future directions and consider some of the most important research challenges that lie ahead.

Solutions to the research challenges will require the best efforts of many talented minds around the world. Continued collaboration between academia and industry is vital at the graduate research level. Equally important is the need to ensure that young engineers are adequately prepared for the era of programmability. We close by describing some of the novel work emerging from Xilinx research and the initiatives that we are taking to ensure the ongoing success of our partnership with academia worldwide.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1, 2004.

Reconfigurable Systems Emerge

Nick Tredennick and Brion Shimamoto

Gilder Technology Report, 1625 Sunset Ridge Road, Los Gatos, CA 95033
bozo@computer.org

Abstract. As the world of electronics shifts from tethered devices to mobile devices, reconfigurable systems will emerge. After twenty years, the PC is now good enough for most consumers' needs. As PC development becomes less important, engineering emphasis shifts to mobile devices: digital cameras, MP3 players, and cell phones. Mobile devices redirect the design goal from cost performance to cost-performance-per-watt. Smaller transistors don't help because they are too expensive and because they leak too much. The microprocessor has effectively stalled hardware design for thirty years, and it will not be the workhorse in mobile devices of the future. Both microprocessors and DSPs are unsuitable for mobile devices because instruction-based processing is computationally inefficient and because they use too much energy. Today's memory components are also unsuitable. New programmable logic devices, based on next-generation non-volatile memory, will enable efficient reconfigurable systems.

1 Introduction

A few years ago, when a speaker promised a surge in robotic applications, a skeptic in the audience interrupted: "Robotics advocates have promised soaring applications for fifty years and haven't delivered. What's different this time?" I don't recall the speaker's answer, but my answer would have been: "Today, we can put millions of transistors of intelligence in a knee joint for less than a dollar."

Advocates for reconfigurable systems, who have a history going back to the beginnings of Altera and Xilinx more than twenty years ago, face similar skepticism. Times change. Robots are surging into applications—reconfigurable systems will soon follow.

To put the conclusion up front, the engineering community is coming out of a thirty-year stall in the development of design methods that was caused by the enormous success of the microprocessor. The microprocessor will move from its central role in problem solving to a supervisory role. "Paged" circuit definition will displace instruction-based algorithms as the workhorse of systems. For the engineering community, it will be a reluctant transition, one that will be forced by the rapidly growing market for consumer mobile devices.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 2–11, 2004.

2 The Microprocessor

The transistor was the first generic semiconductor. The transistor led to the integrated circuit. The integrated circuit combined transistors on a single chip to form building blocks called “logic macros.” Families of logic-macro chips proliferated because they raised the designer’s efficiency. Instead of designing with individual transistors, engineers built systems from building-blocks. The microprocessor, introduced in 1971, was the culmination: an integrated circuit that was a computer.

The microprocessor differed fundamentally from logic macros. The microprocessor brought the computer’s problem-solving method to system design. The computer’s breakthrough in problem solving was to separate the algorithm from the hardware. The computer separated physical structure from logical procedure. The computer provided general-purpose hardware, including a state sequencer. To solve problems, the engineer provided procedure in the form of a program.

The engineer no longer needed to design as much hardware and no longer needed to design a state sequencer. *Problem solving became programming*. This had two beneficial consequences. First, engineering productivity increased because one engineering team designed the microprocessor that supported a range of applications and because engineers solve problems faster with programs than with logic macros. Second, the community of programmers is probably ten times larger than the community of logic designers.

In designing with logic macros, the engineer built systems from blocks of hundreds or thousands of transistors. Logic-macro chips raised the engineer’s level of abstraction. For this gain in productivity, engineers forfeited efficiency because individual transistors could no longer be custom tailored to their position in the circuit.

Trained problem solvers are a critical resource. The microprocessor added programmers to the community of problem solvers.

The microprocessor consolidated logic macro chips into microprocessors, memory, and peripherals. Since the microprocessor was a generic component, its design cost could be amortized across a range of applications, lowering its cost and increasing its unit volumes. The microprocessor’s market grew from almost nothing at its introduction to billions of units per year.

Microprocessor-based design is entrenched in the engineering community. University engineering programs teach microprocessor-based design almost exclusively. Development software (compilers, assemblers, profilers, debuggers, simulators) and support systems (PCs, workstations, development boards, web sites) abound. Billion-dollar companies (Apple, IBM, Intel, Motorola, STMicroelectronics, Texas Instruments) sell chips and services for microprocessor-based design.

Can anything dislodge the microprocessor from its central role? Yes, the value PC and the value transistor.

3 The Value PC

For more than twenty years, the personal computer (PC) has been the focus of development for the semiconductor industry. In recent years, the PC industry has consumed forty percent of the dollar value of all semiconductors.

When the PC was introduced, its performance didn't satisfy any of its users. Over time, the PC's performance improved at a rate related to the Moore's-law improvement in its underlying components, especially its microprocessor and memory. Because they are consumer items, PCs are built for low cost. But the industry's profitability is tied to buyers' demand for performance.

The demand for performance seemed insatiable. PCs continued to offer more performance, and consumers bought all the performance they could afford. In the semiconductor industry, we got so used to this behavior that we think of demand as infinitely elastic. We forgot about the difference between supply and demand.

The PC industry supplies performance that rises with time. The demand for performance also rises with time. We expect more from our next PC than we did from our last one. But *there is no necessary correlation between the rise in the supply of performance and the rise in demand for performance*.

As nerds, we don't see the separation between supply and demand. We think supply *conjures* demand. The nerd community, including the readers of this paper, is demand's leading edge. Nerds are early adopters, always demanding more performance. But as the PC matures more consumers enter the market. These late adopters, clerks, accountants, school teachers, bureaucrats, and so on, are not the sophisticated users of the nerd community. They don't need as much and they don't expect as much. Thus, demand spreads; nerds, with leading-edge applications, want more performance, late adopters, with trailing-edge applications, don't need as much performance.

If the supply of performance rises faster than the demand for performance, then the PC's performance will eventually exceed the demand for some users. The supply of performance has been rising faster than the demand for performance is increasing. The PC's performance is good enough for a great many users. This situation creates the value PC. *Value PCs offer good-enough performance at low absolute prices*. The demand shifts from the leading-edge PCs to value PCs, decreasing industry profitability.

And each year's value PC offers more performance than last year's value PC, so each year's value PC covers a larger area under the demand curve. This further decreases industry profitability, creating incentive to reallocate engineering resources to more profitable systems. Engineering emphasis is shifting from tethered systems, such as the PC with its decreasing profitability, to more-profitable untethered systems.

The shift from tethered systems to untethered systems changes the design goal from cost performance to cost-performance-per-watt.

4 The Value Transistor

For more than forty years, the semiconductor industry's answer to the need for more performance at lower cost has been the shrinking transistor. Smaller transistors run faster and they use less energy to switch.

The same number of transistors fits in a smaller area, decreasing cost and lowering power. The smaller chip thus becomes cost-effective in applications for which the older-generation chips had been too expensive.

For a fixed area and cost, more transistors fit on the chip. With more transistors, the chip becomes adequate in applications for which older-generation chips did not have enough logic capacity or did not have the performance.

Fixed-*logic* chips got cheaper; fixed-*size* chips got more transistors for the same cost. This model of decreasing the cost for low-end applications and of increasing the performance and capacity for high-end applications is built into the business models of many semiconductor companies. The revenues from one product generation support shrinking the chip or they support designing the next higher-performance chip.

But, just as with the value PC, this model ignores the difference between supply and demand. As long as the transistor isn't good enough, Moore's-law improvements in transistors find a ready market. But there's no *assurance* that demand for smaller transistors is growing as fast as Moore's law can shrink transistors. Today's answer is: "It always has." The value transistor is changing the answer. *The value transistor is a transistor that is good enough for the application*. As we shall see, it's not always the smallest transistor.

Like the PC, when the transistor came out, it didn't satisfy any of its applications. Moore's-law improvements define the *supply* curve for transistors. And, like the PC, demand is a difficult-to-define family of curves that spreads with time. The leading-edge designs of the early adopters require leading-edge transistors; the trailing-edge designs of the late adopters do not require leading-edge transistors.

In the early days of integrated circuits, smaller transistors were adopted quickly. This worked for decades, but times change. As recently as 1996, for example, TSMC introduced its 350-nm process. By the beginning of 1997, about thirty percent of its wafer starts were in the new, smaller transistors. In two years, that rose to more than fifty percent. Contrast that with the adoption of a 150-nm process introduced in 1999, which by 2001 represented less than five percent of wafer starts.

A number of reasons account for the decline in adoption rates, but the primary reason is the value transistor; the old transistors are good enough for some applications. As time goes by, more and more applications will be satisfied with less than leading-edge transistors. Plant costs double with each process generation. The cost of the plant and its equipment must be amortized into the cost of the transistors it produces. With escalating costs, newer transistors can be more expensive than old transistors coming from already-amortized plants. Applications won't opt for newer, more-expensive transistors unless there's a compelling performance requirement—and for an increasing range of applications, such as microwave ovens and hair dryers, there isn't.

Even the cost of moving a design from an old process to a new one increases with each generation, so the incentive to cost-reduce old designs decreases with time. Smaller transistors use less switching energy, but they leak more. That wasn't a problem when chips held thousands of transistors because the leakage currents were several orders of magnitude smaller than the switching currents. But as designs grow to billions of transistors, leakage currents become significant. In modern performance-oriented designs, leakage currents can amount to half of the chip's power budget. More on this later.

5 Consequences

Here's the situation in the semiconductor industry. Microprocessor-based design is thoroughly entrenched. Progress in computer design has created the value PC, which is good enough for most users. Moore's-law progress in semiconductors has created the value transistor, which is good enough for most applications.

As value PCs displace leading-edge PCs in consumer preference, the decreased profits suggest moving engineering resources from PC development to more-profitable products. The development effort is shifting from tethered systems to untethered systems. This shift changes the design objective from cost performance to cost-performance-per-watt.

The shift in design objective to cost-performance-per-watt is a problem for microprocessor-based design. The microprocessor was invented to replace custom hardware in systems where its performance was adequate and where its cost was lower. The microprocessor's use was justified because it lowered the cost of design and it lowered the cost of components. It lowered the cost of design because engineers designed systems faster by programming ready-made, general-purpose, microprocessor-based systems than by creating custom hardware.

Efficiency, a requirement for cost-performance-per-watt, was never a hallmark of microprocessor-based designs. It couldn't be. The microprocessor is, after all, an instruction-based emulation of what would otherwise be custom hardware. Custom hardware is the benchmark for performance and for efficiency.

One way that microprocessors extended their range of applications was through increased performance. The primary means for increasing the microprocessor's performance has been by increasing its clock frequency. Roughly speaking, doubling the microprocessor's clock frequency doubles its performance. Unfortunately, clock frequency appears as a term in the power equation. Doubling the clock frequency doubles power use. Up to a point, that's not a problem for systems that use wall power. No one cares whether the microprocessor in a 1500-watt microwave oven dissipates a tenth of a watt or ten watts.

To control power use in microprocessors, designers have been lowering the voltage as the clock frequency increases. Fortunately, the voltage term in the power equation is squared. Halving the voltage lets the microprocessor run at four times the clock frequency for the same power. But there's a limit below which the microprocessor's transistors no longer function. Today's microprocessors are too close to that limit to benefit from further voltage reductions.

Microprocessors have reached their voltage floor while performance requirements, particularly for mobile communication systems, continue to climb. The microprocessor can't meet the cost-performance-per-watt requirements of untethered systems. And smaller transistors of the newer semiconductor processes won't help either. In two process generations, leakage currents increase by a factor of ten. Two watts of leakage in a 130-nm process becomes twenty watts in a 65-nm process. Performance requirements for mobile communication systems are growing faster than the Moore's-law rate of improvement in semiconductors. That means that, even in the absence of leakage-current problems, faster transistors wouldn't yield enough performance to make microprocessor-based implementations practical.

6 ASICs

Application-specific integrated circuits (ASICs) represent the ultimate in logic efficiency. Unfortunately, escalating requirements in mobile communication systems demand flexibility that ASICs, unlike programmed or programmable components, just don't have. Also, ASIC design costs have been growing rapidly. ASICs can still be cost effective, but their growing costs mean that they are cheaper for an ever-narrowing range of high-volume applications.

Microprocessors are flexible, but they lack efficiency; ASICs are efficient, but they lack flexibility. Two companies, ARC and Tensilica, offer a compromise between the two. They offer customizable instruction sets in the form of an ASIC. Designers adding custom instructions to a base microprocessor might achieve a ten- or hundred-fold improvement in performance for their application.

7 Programmable Logic

Altera and Xilinx have been selling programmable logic devices for more than twenty years. But these programmable logic devices won't do either. The SRAM-based versions, commonly called FPGAs (field-programmable gate arrays), come the closest to the requirements for custom logic and for flexibility. But they are too slow, too power hungry, and too expensive. This is so because Altera and Xilinx built their businesses around logic prototyping and on consolidating glue logic. Their device designs have followed the needs of their current customers and are, therefore, not well suited to the needs of mobile communication systems.

Altera and Xilinx offer soft-core microprocessors for their FPGAs. Altera's soft-core microprocessor is NIOS, which now offers application-specific customization of its instruction set. Xilinx, in addition to offering two soft-core microprocessors, MicroBlaze and PicoBlaze, offers hard-core PowerPC microprocessor cores on its high-end FPGA family. These microprocessor cores relieve system designers from having to design a state sequencer.

In addition to the microprocessor cores, both companies offer a wide range of intellectual-property cores for licensing. Altera and Xilinx provide some of the cores and third parties provide some of the cores. The development of in-house intellectual property is subsidized by chip sales. The customer base for both companies has been logic designers. With the move to microprocessor cores and to the availability of a wide range of intellectual-property cores, both companies are moving slowly toward programmers as customers.

As FPGA applications move from custom hardware prototyping and from glue-logic consolidation to general applications that employ on-chip hard- or soft-core microprocessors and other intellectual-property cores, the logic elements on the chips may become more application specific (rather than being the relatively fine-grained look-up tables that they are today).

8 Memory

The semiconductor industry wants to move to untethered systems, but it lacks a suitable vehicle. Microprocessors lack both the performance and the efficiency. ASICs lack the flexibility and will be restricted to large unit-volume markets due to their high development cost. FPGAs are too slow, too power hungry, and too expensive.

Not only are there no suitable components for implementing the logic, but today's memory chips are unsuitable as well.

Today's memory chips are DRAM, SRAM, and flash memory. All of these devices grew with the personal computer market. DRAM, the PC's working memory, is dense, but it is slow and it does not retain data through power cycles. SRAM, which speed-matches between the PC's fast microprocessor and its slow DRAM working memory, is fast, but its memory cells take a lot of space and they use a lot of power. SRAM also does not retain its data through power cycles. Flash memory, which holds the programs that initialize the PC on start, retains its data through power cycles, but it is slow for reading, very slow for writing, and it actually wears out. The PC exploits the advantages of these memory types and is relatively insensitive to their shortcomings. Riding the success of the PC, each of these memory types achieved high-volume production, and, therefore, low cost.

The ideal memory component would have the density of DRAM, the performance of SRAM, and the non-volatility of flash memory. Candidates for this ideal memory have been around for fifteen years, but have never been able to achieve the low cost that would let them displace any of the components in the PC.

In untethered systems, DRAM, SRAM, and flash memory alone and in combination are all unsuitable. This creates an enormous investment incentive for the industry to create the ideal memory component. It could be one of the long-time candidates, such as magnetoresistive random-access memory (MRAM), ferroelectric random-access memory (FRAM), or ovonic unified memory (OUM). Each of these has major backers and a long development history. But I think it just as likely that the ideal memory component will come from a startup such as Axon Technologies or Nantero.

Axon Technologies, for example, uses a solid electrolyte between two conductors, one of which is silver. A small potential across the conductors causes silver atoms to ionize and migrate through the electrolyte. They plate out on the opposite conductor and capture an electron. In a matter of ten nanoseconds, a physical bridge forms between the two conductors. The connection can be maintained indefinitely. The process is completely reversible. Nantero's memory is based on properties of carbon nanotubes. There are probably dozens of others. We should soon see commercial memory chips with characteristics ideal for untethered applications.

9 Programmable Logic Revisited

When we do get the ideal memory cell, it will also benefit FPGAs, which today are dominated by on-chip configuration memory and by interconnections. If the on-chip configuration memory cells, which today are SRAM-based, are replaced with ideal memory cells, FPGAs will gain enormously in logic capacity, in performance, and in convenience. Each one-transistor, non-volatile memory cell will displace at least six transistors in the configuration memory. This brings circuits closer together and it reduces interconnection distances, which increases performance. Since the configuration memory will then remember its data across power cycles, the chips won't require off-chip configuration storage. They won't have the configuration delay and configuration information will be more secure since it won't be passing across the chip interface each time the power comes on.

Combining the new non-volatile memory cells with application-oriented logic elements and on-chip microprocessor cores makes future FPGAs well suited to untethered applications. The microprocessor core provides the state sequencer that supervises the application. The combination of intellectual-property cores, custom instructions, and custom hardware provides efficient application-specific logic. Software, such as Altera's SoPC Builder or CriticalBlue's Cascade, makes these resources available to the community of programmers.

An implementation based on an FPGA with new non-volatile memory cells and with new application-oriented logic elements could be more efficient than an ASIC, even for an implementation that doesn't require the FPGA's flexibility. Yes, I think the future's FPGA, call it an "nvFPGA," could be more efficient than an ASIC, which is today's benchmark for custom-logic efficiency.

Think of the dozen or so substantially different tasks performed by a modern cell phone. With an ASIC, each task's hardware is permanently resident. They can't all be next to each other and they can't all be close to the data sources. All of the ASIC's functions will contribute to leakage currents and will subtract from the power budget. In the nvFPGA, only the task that is running has hardware present. The hardware-definition bits for each task are "paged" into the nvFPGA as needed. Instead of shipping multimedia data from unit to unit as would happen in an ASIC, hardware might be paged into the area where the data is. Moving the hardware to the data might be more efficient in power use and in time than moving the data to the hardware, particularly for data-intensive algorithms. Functions can be closer to the data and closer to communicating neighbors than they might be in an ASIC.

10 Getting There

Let's suppose all of this is true: the semiconductor industry is shifting from tethered systems to untethered systems, microprocessors won't do, ASICs are too expensive, the ideal memory cell will arrive soon, and the FPGA vendors will deliver components with the new non-volatile memory cell, with application-oriented logic elements, and with the software to take advantage of their resources. We still have the enormous problem of turning a whole industry that seems irreversibly committed to microprocessor-based design. Can this be done?

I think it can. What the industry needs is a compelling proof-of-concept for reconfigurable systems. It's about to get one from Stretch, Inc.

Stretch is a Silicon Valley startup company. Though Stretch never uses the term reconfigurable, it can still deliver the proof that's needed to turn the industry. Stretch is a little like Tensilica. It offers a Tensilica instruction set and it offers software to customize the implementation. But there's a difference. Tensilica appeals to logic designers who want to design an application-specific microprocessor for implementation in an ASIC. Stretch offers a generic component that is a combination of a Tensilica microprocessor and programmable logic. Stretch's chips appeal to programmers who cannot get the performance they want from off-the-shelf microprocessors. The engineer writes the application in a high-level language and profiles the program with Stretch's software. The engineer designates one or more performance-critical regions in the program. Stretch's software generates hardware equivalents of the performance-critical program regions.

Many have tried and many have failed with reconfigurable systems. But Stretch offers two decisive advantages over predecessors. First, Stretch offers generic chips that are customized in the field. Second, Stretch's customers are programmers, not logic designers.

A generic chip that is customized in the field has the advantage of production volume that is the sum of all the applications. Volume production means lower cost. Amortizing development cost across the *range* of applications makes the chip cheaper. Engineers might get close to the performance of a custom design without the costs of ASICs.

Stretch's chips are accessible to programmers. *Stretch doesn't have to convert logic designers into believers to succeed.* The community of programmers, which is perhaps ten times as large as the community of logic designers, gets access to hardware acceleration.

11 Industries in Transition

Even if I’m wrong about the transition from tethered systems to untethered systems and about the imminent arrival of non-volatile memory, reconfigurable systems will still happen.

There’s an ongoing, fruitless search for “the next killer app.” It’s time to give up the search; this view of the future is narrow. Instead, let’s look at opportunities offered by industry transitions. Here are examples.

The automotive industry is in transition from analog to digital, from mechanical to electrical, and from isolated to connected. The film and video industry is in transition from analog to digital and from isolated to connected. The consumer-products industry is in transition from analog to digital, from tethered to untethered, and from isolated to connected. (Consumer products also offer high-growth opportunities supplying everyday appliances to emerging economies.) The biomedical industry is in transition from analog to digital and from wet laboratories to bioinformatics. The telecom industry is in transition from copper to wireless, from copper to fiber, and from analog to digital. The computer industry is in transition from desktop to embedded.

These transitions will transform these industries. These transitions will create large markets for semiconductors, for microelectromechanical systems, and for computers and software over the next few years. The first big transitions, from machine-room computers to personal computers and from wired telephones to cell phones, were viewed as “killer apps.” But they were really pioneering applications that opened the gates for the general transition from analog to digital.

Many of these transitions, like the one from tethered systems to untethered systems, will demand power efficiency that cannot be gotten from microprocessor-based systems. Reconfigurable systems will emerge for those applications for which microprocessors lack the necessary efficiency and for which ASICs are too expensive or too inflexible.

System-Level Design Tools Can Provide Low Cost Solutions in FPGAs: TRUE or FALSE?

Mark Dickinson

Altera European Technology Center

Abstract. Due to fundamental dynamics in the semiconductor industry, FPGAs vendors are increasingly focusing on cost and EDA vendors are focusing on system-level design. This is good news for the end user because both time-to-market and device unit cost are top of their priority lists whilst performance and density now often take a back seat – these problems having been solved. However, are these two initiatives of cost reduction and better system-level design tools correctly aligned for FPGAs? This talk will explore some of the areas where they are aligned, and some areas where they are not. The analysis uses real system examples to illustrate the successes to date, as well as the challenges looking forward as more and more engineers develop low cost solutions using FPGAs.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 12, 2004.

Hardware Accelerated Novel Protein Identification

Anish Alex[1], Jonathan Rose[1], Ruth Isserlin-Weinberger[2], and Christopher Hogue[2]

[1] Department of Electrical and Computer Engineering, University of Toronto,
10 King's College Road, Toronto, Ontario, M5S 3G4, Canada
{anish,jayar}@eecg.toronto.edu
[2] Department of Biochemistry, University of Toronto,
University of Toronto, 1 King's College Circle, Toronto,
Ontario M5S 1A8, Canada Wim Roelandts
{isserlin,hogue}@mshri.on.ca

Abstract. The proteins in living organisms perform almost every significant function that governs life. A protein's functionality depends upon its physical structure, which depends on its constituent sequence of amino acids as specified by its gene of origin. Advances in mass spectrometry have helped to determine unknown protein sequences but the process is very slow. We review a method of de-novo (novel) protein sequencing that requires a fast search of the genome. In this paper, we present the design of a hardware system that performs this search in a very fast, cost-effective manner. This hardware solution is more than 30 times faster than a similar search in software on a single modern PC, and up to 40 times more cost effective than a computer cluster capable of the same performance. The hardware is FPGA-based to reduce the cost and allow modification of the algorithm, both key requirements of practical protein analysis tools.

1 Introduction

Proteins and their interactions regulate the majority of processes in the human body. From mechanical support in skin and bones to enzymatic functions, the operation of the human body can be characterized as a complex set of protein interactions. Despite the efforts of scientists, many proteins and their functions have yet to be discovered. The wealth of information that lies in these unknown proteins may well be the key to uncovering the mysteries that govern life. The subject of this paper is the use of digital hardware to aid in a specific technique used to discover new proteins.

Proteins are composed of long chains of molecules known as amino acids, and the order of these amino acids is known as the sequence of a protein [2]. Protein sequencing - the process of identifying the sequence of a given protein - is a means of establishing the protein's identity, from which its functionality can be inferred. Advances in technology over the past two decades introduced the concept of protein sequencing by mass spectrometry [3]. A mass spectrometer (MS) is a device that takes a biological or chemical sample as input and measures the masses of the constituent particles of the sample. This mass information is used to identify the molecules in the sample. Protein samples to be identified are broken down into smaller subunits known as peptides and fed into an MS for identification. For novel proteins, the common ap-

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 13–22, 2004.

proach is to identify each peptide, combine this information and thus determine the complete protein sequence. However, the sequence of novel proteins can be obtained from the information contained in genes which act to create proteins [2]. In effect, a complete genome can be interpreted as a complete protein database. For this technique to be useful in MS experiments however, very high speed searches of the genome are required, which are not feasible on general purpose processors due to the limited memory bandwidth. In such applications such as high speed searches, which involve identical repeated operations, custom hardware implementations are an ideal solution.

2 Background

A basic understanding of genetics and protein synthesis is required to comprehend the protein identification methods described in this paper. To this end, we provide a brief overview of protein synthesis and common methods of protein identification in use today.

2.1 Protein Synthesis

Within an organism, proteins are synthesized from the DNA template stored in the cell. DNA is contained in the *genes* of organisms where it is stored as a chain of nucleic acid molecules which consist of Adenine, Thymine, Cytosine and Guanine (A,T,C,G). These genes are translated into a chain of amino acids by a series of biological mechanisms. Thus if the set of all genes of the organism – its *genome* – is known, the set of all proteins it can create – its *proteome* – can be inferred. An example of protein synthesis is shown in Fig 1.

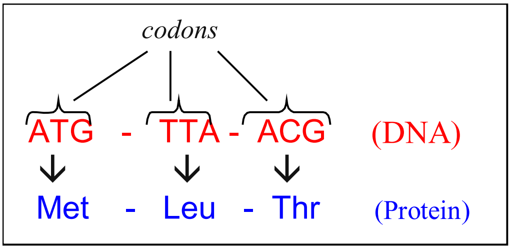


Fig. 1. Protein Synthesized from gene

Note that the DNA in the gene is grouped into sets of 3 DNA molecules, or *codons*. Each of these codons is then translated into an amino acid resulting in the protein chain. The rules for the translation from codons to amino acid are well known, and it is therefore easy to translate a codon string to its corresponding amino acids and vice versa [2]. However, it is difficult to reliably predict the starting point of a gene. The three DNA molecules in a codon, imply 3 different possibilities (known as *reading frames*). Further, every DNA strand is coupled with a complementary DNA strand that may also encode proteins resulting in a total of 6 reading frames.

2.2 Protein Identification

Prior to analysis by a mass spectrometer, proteins are digested or broken down into smaller subunits known as *peptides*. Digestion is performed by enzymes (such as trypsin) that cleave the protein at specific amino acid points. An example of the tryptic cleavage of a protein is shown in Figure 2.

MAVRAPCOKLHNWF
Original protein in sample

MAVR APCOK LHNWF
After digestion – 3 smaller tryptic peptides

Fig. 2. Trypsin Digestion of Proteins

Trypsin always cleaves the protein after the amino acids Argnine (R) and Lysine (K) with a few special exceptions. Mass spectrometric analysis of the peptides is then performed as follows:

1. **Selection:** An MS measures the masses of the tryptic peptides, and creates a *list of masses*. An operator then selects an individual peptide by mass from this list for further analysis.
2. **Identification:** The selected peptide is fragmented and analyzed by a second MS; this is followed by a complex computation that produces the sequence of the selected peptide.
3. **Repeat Cycle:** After a short delay (approx 1 sec.), another peptide is selected from the list and Step 2 is repeated. This is done for each peptide on the list.

The peptide sequences from individual peptides are grouped together and ordered to obtain the full sequence of the protein. With a few hundred peptides in a sample, a great deal of the delay in the MS process comes from having to repeat the identification process for each peptide. It is possible to use a single peptide sequence as a query to a protein database, which will return the protein that contains the query peptide as its substring. However, this technique only applies to known proteins, whose sequences exist in databases. For de-novo sequencing, i.e. the identification of novel proteins, a variant of this technique is used which relies on the core concept described in the previous section - *every protein is synthesized from the template of DNA stored in the genes*. This implies that given the knowledge of all the DNA of an organism (its *genome*) the sequence of all its proteins can be determined. In effect, the genome of an organism is translated into its *proteome* – the set of all its proteins. This proteome can then be used as a protein database, as described above to identify novel protein sequences.

3 Inferring Proteins from Genes

The concept of using genetic information to infer protein sequences is not new. Several researchers have proposed and implemented the concept in software. We employ an approach similar to that employed by Choudary et al [1]. The steps of the method are illustrated in Figure 3.

The process begins with step a) in Figure 3, which begins immediately after the second step of the MS analysis procedure described earlier. After one peptide has been identified (sequenced), it is reverse translated to the codons from which it could have originated. Thus a set of DNA strands that could have coded the peptide is now available. In step b) these DNA strands are used as queries to a genome database – in effect we are searching for the genes that may have coded the sequenced peptide. As indicated in b) it is likely that there will be multiple genes that match the set of queries. To uniquely resolve the true match, each gene must be individually considered. In step c) in Figure 3, each gene is translated to its protein equivalent as shown in Figure 1, and these translated proteins are then cleaved as shown in step d). Note that the translated protein is also cleaved at the K and R amino acids. The mass of each of the translated tryptic peptides is then calculated and compared to the list of masses produced by the first step of MS analysis. This process essentially compares the translated protein from the database against the protein actually detected by the MS. If a sufficient match is found, the protein sequence has been identified and no further work need be done. If not, the next matching gene is translated to its protein equivalent and the process is repeated.

The most obvious advantage of this approach is that the overall sequencing time will be greatly reduced. In the example in Figure 3, only a single peptide was required to obtain the full protein sequence. Note from step 3 of the MS analysis process that there is a 1 sec. (approx) delay before each peptide on the mass list is analyzed. If the algorithm described above is to be useful it must be performed within this delay. If not, an expensive downtime will be incurred as the MS instruments are stalled while the operator determines the next peptide to be analyzed. Note that several attempts at hardware sequencing have been implemented in the past [8] but we believe that this is the first published design targeting real time sequencing. Further, the work presented here is the only published design that uses the unannotated MS mass information to rank its outputs.

The key requirement of this approach is the ability to search the genome database and make comparisons to the mass list at high speeds. The Human genome contains 3.3 billion nucleic acids, and a search time of 1 second requires enormous throughput. Fortunately this kind of search is highly parallelizable in both software and hardware. Applications of this nature are good candidates for custom hardware implementation, thus the goal in this work is to design a hardware system that meets the requirements of the sequencing algorithm as described above. A number of hardware based genome search techniques have been developed over the years. Many of these, such as implementations of the Smith-Waterman algorithm are geared towards calculating the edit distance between two stringss. Other commercial devices such as DeCypher [8] are designed to perform BLAST and HMM searches which are not designed to use MS data to help identify true matches.

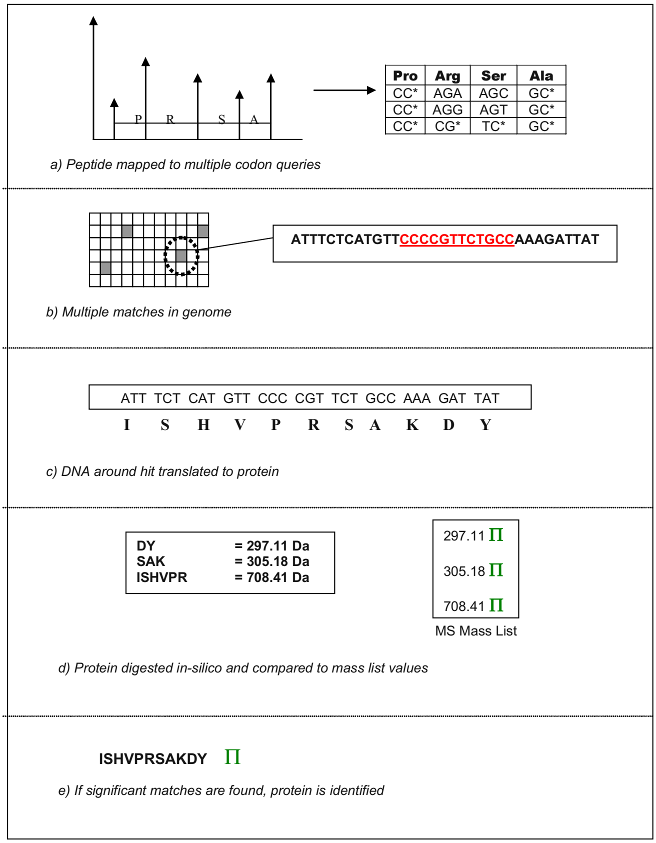

P R S A
Pro Arg Ser Ala
CC* AGA AGC GC*
CC* AGG AGT GC*
CC* CG* TC* GC*
a) Peptide mapped to multiple codon queries
ATTTCTCATGTTCCCCGTTCTGCCAAAGATTAT
b) Multiple matches in genome
ATT TCT CAT GTT CCC CGT TCT GCC AAA GAT TAT
I S H V P R S A K D Y
c) DNA around hit translated to protein
DY = 297.11 Da
SAK = 305.18 Da
ISHVPR = 708.41 Da
297.11 Π
305.18 Π
708.41 Π
MS Mass List
d) Protein digested in-silico and compared to mass list values
ISHVPRSAKDY Π
e) If significant matches are found, protein is identified

Fig. 3. Algorithm Outline

4 Design

The goal of the algorithm described in the previous section is to search through the genome in less than 1s. In addition a mass calculator is required to translate the genome to its tryptic peptide equivalents and calculate the peptide masses. Finally, a scoring system capable of comparing these calculated peptide masses to the masses measured by the MS is required.

The design takes three primary inputs, namely:

1. A peptide query from the MS, which is a string of 10 amino acids or less,
2. A genome database,
3. A list of peptide masses detected by the MS.

The design produces a set of outputs for a given peptide query:

1. A set of gene locations within the genome, which can code the input peptide query,

2. A set of scores for each potential protein gene location identified in the search. The scores rank the genes based on the likelihood that they coded the protein in the sample.

An overview of the units and their interconnection is shown in Figure 4.

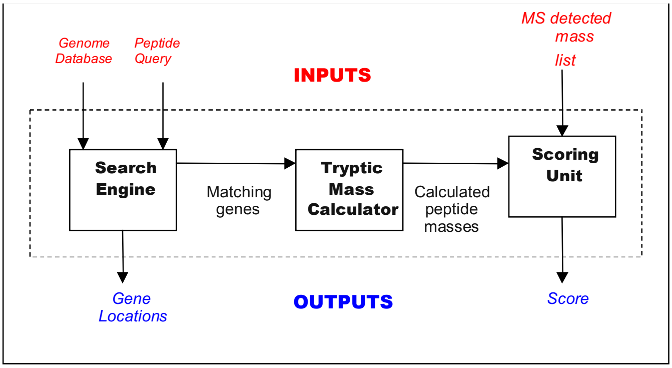


Fig. 4. Device Architecture

The search engine identifies all locations in the genome that can code the peptide query while the tryptic mass calculator translates these gene locations into their protein equivalents. The scoring unit then compares the peptides in the translated proteins to the peptides detected by the MS and provides a ranking for each gene location based on how well it matches the masses detected by the MS.

4.1 Search Engine

The first key component of our hardware is the search engine, which returns the location of every gene that can synthesize a query peptide. Many FPGA based text search techniques have been designed over the years, particularly for genome databases [8] and network security systems [6]. Most network security systems however require a new circuit to be synthesized for different sets of rules (i.e. different query strings) while genomic search engines are optimized for edit distance calculations. We require a far simpler circuit that serves merely to identify the presence of a query string in the genome.

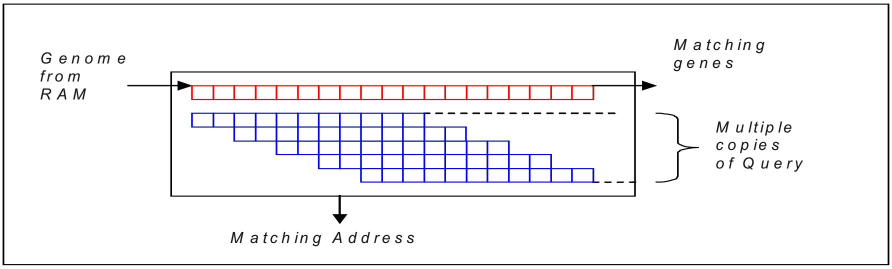


Fig. 5. Search Engine

Thus our search is simply a linear traversal through memory in which each address is read, and its contents are compared with the query. Figure 5 shows that multiple copies of the query are initialized in the hardware to simultaneously perform a parallel comparison with every position in the RAM word. If a match is detected, the corresponding memory address is sent to the user, who can then identify the coding gene at this location. Figure 5 illustrates a simplified view of the search engine, and additional hardware to implement wildcarded searches [7] has been removed from the diagram for brevity. This search looks at every nucleic acid position in the genome individually, thus no additional consideration of reading frames is required. Note further, that the matching genes are passed to the next unit, the tryptic mass calculator.

4.2 Tryptic Mass Calculator

As described, there may be several matching genes and it remains to determine which of these is the true coding gene. To this end, each gene must be translated to its protein equivalent to determine whether its constituent peptides have been detected by the mass spectrometer. To perform the translation of genes, the matching genes from the search engine are sent to the tryptic mass calculator, which interprets the DNA data from the genome as a set of codons or equivalently, a set of amino acids. It further accumulates the amino acid masses and identifies the tryptic digestion sites (the amino acids K and R) to produce a list of tryptic peptide masses. An example of the translation and calculation process is shown in Figure 6. It must be noted that the calculator interprets the gene as stored in RAM and also as the complement of the stored sequence - thus a calculator produces two reading frames of peptide masses for every gene sequence. To cover all possibilities, three calculator units are instantiated to translate all six frames of information simultaneously.

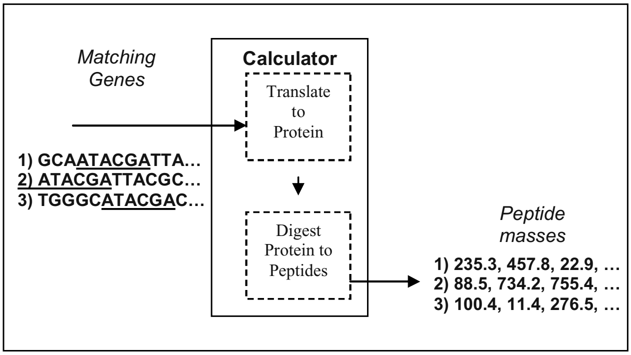


Fig. 6. Tryptic Mass Calculator

Each matching gene in Figure 6 is translated to a list of tryptic masses. It remains to identify whether these calculated peptide masses were detected by the MS. The scoring unit detailed in the following section performs this task.

4.3 Scoring Unit

Once each potential coding gene is translated to a tryptic mass list, it must be compared to the list of detected masses from the MS. The gene showing the closest match can then be identified as the true coding gene for the protein sample in the MS.
The first step of the scoring process is to store the MS mass list to chip. The values are stored using data associative techniques similar to those used in Content Addressable Memory (CAMs). A subset of the most significant bits of the mass value is used to divide the masses into specific ranges as illustrated in Figure 7. The ADDR_BITS most significant bits of the address are used as an address to store the MS measured-mass.

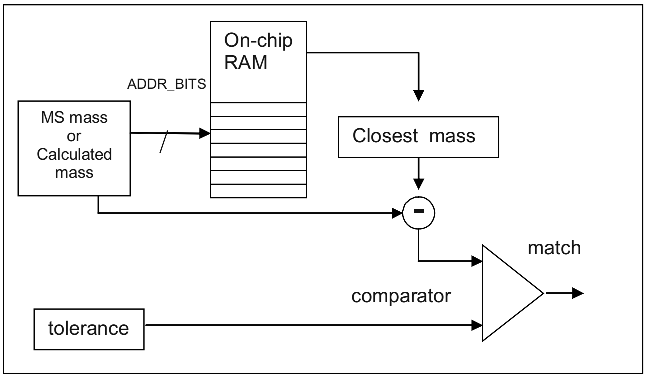


Fig. 7. Data Associative Mass Storage and Matching

Upon device initialization, each of the masses from the MS is stored in the on-chip RAM. The calculated mass is then used as an address to retrieve its closest matching mass from RAM. If the difference between these values meets a user specified tolerance, a match is signaled. It must be noted that the matches alone do not determine the final score. The final step of the scoring process once again divides masses into ranges in a manner similar to that depicted in Fig 7. In effect a histogram of masses is recorded in hardware. This histogram records the number of matches in each of the mass ranges and uses this information to calculate the final score. The score is calculated based on the MOWSE algorithm, which attempts to rank matches based on their significance. The interested reader can find the details of the MOWSE algorithm and our specific implementation in [5] and [6] respectively.

5 Implementation Performance and Costs

Variants of the design described above have been implemented in software. In this section we compare the software approach of Choudary et al [1] to our hardware. A slightly simplified version of the hardware design described has been successfully implemented and tested on the University of Toronto's Transmogrifier 3A platform [7][10].

The software scoring algorithm against which we compare our design is MASCOT [4], which is based on the MOWSE scoring algorithm. We choose the work in [1] as our software reference, as it shows the closest match to our hardware. Since the technique of real time genome scanning has generally been deemed infeasible, there have been comparatively few software implementations of search engines that use MS data for scoring. The operations in [1] were performed on a 600 MHZ Pentium III PC, resulting in search and score times of 3.5 minutes (210 s) per query. A significant portion of this time is spent searching through the genome for matches. We scale these values to current processor speeds, by assuming a linear increase in speed based on processor clock speed, which is optimistic. Based on this assumption, we state that the software can complete the task in 52.5 seconds on a 2.4 GHz processor. This scaling is unlikely, as memory bandwidth does not scale with processor speed, but this optimistic assumption presents the ideal performance of this algorithm in software. Table 1 shows what is required to achieve performance that is comparable to the hardware.

Table 1. Total Cost of Processor-based System

Number of CPUs	Scan time (s)	Cost (USD)
1	52.5	$1,962
32	1.6	$31,392
64	0.8	$62,784
512	0.1	$502,272

The original target for the hardware implementation was the Transmogrifier 3A, but as noted above, the design had to be simplified (i.e. lower bit-widths) to fit the design into the onboard FPGAs. To better reflect the capabilitets of modern FPGAs, the system was redesigned to target an Altera Stratix EP1S20 for the search engine and three Stratix EP1S40 FPGAs to implement[1] multiple parallel calculator and scoring units to maximize the processing throughput. The calculator and scoring units operate at a maximum frequency of 75 MHz, thus limiting the system to 2G char/sec. Table 2 below shows the costs of building a full system capable of performing the operations described here.

Table 2. Cost of Hardware Search and Score System

Scan Time (s)	Cost of RAM (USD)	Cost of PCB (USD)	Cost of FPGAs (USD)	Purchase Price [Full] [2] (USD)	Purchase Price [Search] (USD)
1.6	$344	$131	$6,950	$11,137	$1,530
0.8	$689	$262	$13,900	$25,426	$1,530
0.1	$5,512	$2,100	$111,200	$225,469	$12,087

[1] The units were implemented using Altera's Quartus II 3.0

[2] The Purchase Price columns include the cost of a PCB for each of the systems with an additional 50% margin.

The last two columns in Table 2 show the price of the full system (as described above) and the genome search engine as a standalone unit. We divide the systems in this manner as there are myriad applications that only require the search capabilities without the scoring described above. As a standalone search engine, the hardware is capable of out performing the software by a factor of 40 in terms of cost. With advances in mass spectrometry and the rapid progression of genetic and proteomic research, it is clear that custom hardware is a far more practical processing solution.

6 Conclusion

In this work we have studied the design of a hardware system designed to accelerate MS/MS based de-novo protein sequencing. The objective has been to study the feasibility of a custom hardware implementation of a real time protein-sequencing algorithm. The results of this work show that hardware implementations of certain key features of the sequencing system result in performance gains up to 30 times as compared to a modern processor. In addition the cost of a custom hardware solution ranges from 2 to 40 times less than that of a processor cluster capable of similar performance. With such obvious advantages, it is clear that a custom hardware implementation of this algorithm is the better choice for this protein sequencing technique.

References

[1] Choudary, Joyti S., Blackstock, Walter P., M.Creasy, D. Cottrell, John S. "*Interrogating the human genome using uninterpreted mass spectrometry data*", Proteomics, 1, pp. 651-667, 2001

[2] Lesk, Arthur M., *Introduction to Bioinformatics* . Oxford press, NY, 2002, pp.6-7

[3] McLuckey S.A. and Wells J.M., *"Mass Analysis at the Advent of the 21st Centur*y", Chem Rev. 101 (2), 2001, pp. 571-606

[4] MASCOT, http://www.matrixscience.com/cgi/

[5] Pappin, D.J.C., Hojrup, P. and Bleasby, A.J., "*Rapid identification of proteins by peptide mass fingerprinting*". Curr Biol, 1993, 3(6), pp 327-32

[6] Ioannis Sourdis, Dionisios N. Pnevmatikatos: Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. FPL 2003: 880-889

[7] Alex, Anish, "*Hardware Accelerated Protein Identification*", Master's Thesis, University of Toronto, 2003, http://www.eecg.toronto.edu/~jayar/pubs/theses/ Alex/AnishAlex.pdf

[8] Decypher ®, TimeLogic Corporation, http://www.timelogic.com/products.html

[9] Lewis D., Betz V., Jefferson D., Lee A., Lane C., Leventis P., Marquardt S., McClintock C., Pedersen B., Powell G., Reddy S., Wysocki C., Cliff R., and Rose J., *"The Stratix Routing and Logic Architecture"* FPGA '03, pp. 15-20, February 2003.

[10] Transmogrifier 3A, University of Toronto, http://www.eecg.toronto.edu/~tm3

Large Scale Protein Sequence Alignment Using FPGA Reprogrammable Logic Devices

Stefan Dydel and Piotr Bała

Faculty of Mathematics and Computer Science,
N. Copernicus University, Chopina 12/8, 87-100 Torun, Poland,
{stef, bala}@mat.uni.torun.pl

Abstract. In this paper we show how to significantly accelerate Smith-Waterman protein sequence alignment algorithm using reprogrammable logic devices – FPGAs (Field Programmable Gate Array). Due to perfect sensitivity, the Smith-Waterman algorithm is important in a field of computational biology but computational complexity makes it impractical for large database searches when running on general purpose computers.

Current approach allows for aminoacid sequence alignment with full substitution matrix which leads to more complex formula than used in DNA alignment and is much more memory demanding. We propose different parellization scheme than commonly used systolic arrays, leading to full utilization of PUs (Processing Units), regardless of sequence length. FPGA based implementation of Smith-Waterman algorithm can accelerate sequence alignment on a Pentium desktop computer by two orders of magnitude comparing to standard OSEARCH program from FASTA package.

1 Introduction

Sequence comparison is one of the most important bioinformatic problems. Number of published protein sequences increases rapidly and current databases contain gigabytes of the data. Processing of such amount of data, especially sequence comparisons necessary for the scientific discovery, requires, in addition to the efficient data storage, significant computational resources. Moreover, users would like to achieve results fast, in most cases interactively, which cannot be achieved using single, general purpose computer.

For the sequence comparison, the most widely used is Smith-Waterman algorithm [17]. The computational complexity of the algorithm is quadratic which additionally makes it difficult to use for the processing of large data sets. This makes ground for investigation of new algorithms and new acceleration techniques which can make processing of actual and future data feasible.

We propose efficient implementation of the Smith-Waterman algorithm in reconfigurable hardware. Recent Field Programmable Gate Array (FPGA) chips provide relatively low cost and powerful way of implementing highly parallel algorithms. Utilization of FPGAs in the biological sequence alignment has a

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 23–32, 2004.

long history [12], but most of the implementations cover special case of Smith-Waterman algorithm: DNA alignment with simple edit distance, which is equivalent to computing LLCS (length of longest common subsequence) [16]. Efficient bitvector implementations of LLCS dedicated to general purpose computers have been developed [16], potenitially leading to speeding up by a factor of 64 when implemented in modern processors. We investigate the possibility of implementing more general case of the Smith-Waterman algorithm allowing alignment of protein sequences (20 letters alphabet) with full substitution cost matrix (400 entries, 8 bits each) and 18 bits of alignment score using Xilinx Virtex 2 pro FPGAs. Currently Xilinx releases suitable FPGAs: XC2VP50 and XC2VP70.

Number of comparisons of protein residues per second using rigorous Smith-Waterman algorithm exceeds 10^{10} for a single XC2VP70 device and can scale up to 10^{12} in a case of a cluster of PCs each equipped with FPGA accelerator.

2 Problem Description

For the sequence comparison we assume widely used edit distance model. Distance between strings is measured in terms of edit operations such as deletions, insertions, and replacements of single characters within strings. Comparison of two strings is accomplished by using edit operations to convert one string into the other, minimizing the sum of the cost of operations used. This total cost is regarded as a degree of similarity between strings.
Let us define formally our goal, which is *local optimal alignment.*

Definition 1. *Let $A = \{a_1, a_2, \dots, a_n\}$ be a finite set of characters - alphabet. Let ϵ denote the empty string. Let us define A^* as a set of all strings over alphabet A. Notation for a string $w \in A^*$ having $w_i \in A$ as the i-th character is $w = w_1 w_2 \dots w_n,\ i = 1 \dots n$.*

Example 1. We are particularly interested in the following alphabets important in computational biology:

1. $\{C, T, G, A\}$ - nucleotides: Cytosin, Thymin, Guanin, Adenin
2. $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$ - 20 aminoacids.

Definition 2. *An edit operation is an ordered pair (γ, ξ), where γ, ξ are strings having length one or equal to ϵ. Let E_A denote a set of all edit operations over alphabet A.*

Edit operations can be viewed as operations describing rewriting rules: (γ, ϵ) denotes deletion of character γ, (ϵ, ξ) denotes insertion of character ξ and (γ, ξ) denotes substitution of character γ by ξ.

Definition 3. *An alignment of strings $s, t \in A^*$ is a sequence of edit operations $((\gamma_1, \xi_1), \dots, (\gamma_n, \xi_n))$ such that $s = \gamma_1 \dots \gamma_n, t = \xi_1 \dots \xi_n$. Let $Align(s, t)$ denote a set of all possible alignments of strings s, t.*

In order to work with optimal alignments we have to introduce optimization criterion.

Definition 4. *An edit score function $\rho_{edit} : E_A \to \Re$ assigns a real value to each edit operation.*

In computational biology function ρ_{edit} is known as a scoring matrix for proteins such as BLOSUM62 used by BLAST program [25].

Definition 5. *An alignment score function $\rho_{align} : Align(s,t) \to \Re$ is defined as $\rho_{align}(((\gamma_1,\xi_1),\dots,(\gamma_n,\xi_n))) = \sum_{i=1}^{n} \rho_{edit}((\gamma_i,\xi_i))$, where $((\gamma_1,\xi_1),\dots,(\gamma_n,\xi_n)) \in Align(s,t)$, and $s,t \in A^*$. We assign a value equal to sum of all edit operations constituting particular alignment.*

Finally, one can define optimal local alignment, and local alignment score describing optimization goal.

Definition 6. *An Optimal alignment score opt is defined as follows: $opt(s,t) = \max\left(\rho_{align}(\alpha); \alpha \in Align(\hat{s},\hat{t}),\ \hat{s} \subset s, \hat{t} \subset t\right)$. The alignment α having maximum score in this sense is called an optimal local alignment.*

Obvious brute-force approach relays on generating all possible alignments of all substrings of s,t and evaluating its score. This leads to unreasonably high complexity. Using Smith-Waterman algorithm it is possible to reduce complexity to quadratic one.

3 Smith-Waterman Local Alignment Algorithm

This algorithm computes both optimal local alignment and its score. Let $s = s_1 \dots s_m,\ t = t_1 \dots t_n \in A^*$ be fixed input strings. Let us define matrix M:

$$M[i][j] = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ max \begin{cases} 0 \\ M[i-1][j] + \rho_{edit}(s_i,\epsilon) \\ M[i][j-1] + \rho_{edit}(\epsilon,t_j) \\ M[i][j] + \rho_{edit}(s_i,t_j) \end{cases} & i = 1 \dots m,\ j = 1 \dots n \end{cases} \tag{1}$$

Optimal local alignment score is equal to $\max\left(\{M[i][j];\ i = 1 \dots m,\ j = 1 \dots n\}\right)$. In serial approach we compute matrix M row after row. The optimal local alignment itself can be retrieved by tracing back from the maximum entry in M. This algorithm requires mn steps and at least mn space.

3.1 One Row Version

In order to achieve effective FPGA implementation we need to reduce memory requirement of the algorithm. Only one row c of a matrix M and few temporary variables is sufficient to implement serial version, which is also effective for PC

```
int c[n+1]; for (j=0; j<=n; ++j) c[j]=0;
max_score=-MAXSCORE;
for (i=1; i<=m; ++i) {
    Diag=0;Left=0;
    for (j=1; j<=n; ++j) {
        Upper=c[j];
        DLU=max(Diag+sim(s[i],t[j]), max(Left,Upper) + GAP,0);
        c[j]=DLU;
        Diag=Upper; Left=DLU;
        max_score=max(DLU, max_score);
    }
}
```

Fig. 1. Modification of the Simth-Waterman algorithm using only one row c

implementation, because all data fits into cache memory. The most interesting is optimal alignment score which is computed in the FPGA. For those scarce sequences from database which turn out to be enough similar to the query sequence, optimal alignments can be computed in desktop computer within short time. The algorithm written in C-like syntax is shown in the Fig. 1. Similarity matrix $sim(s,t)$ in program is an alias for $\rho_{edit}(s,t)$ and we assume constant penalty for inserting a gap: $GAP = \rho_{edit}(s_i, \epsilon) = \rho_{edit}(\epsilon, t_j)$. This algorithm is a basis for a parallelized and pipelined FPGA implementation.

4 Pipelined and Parallelized FPGA Implementation of FPGA Smith-Waterman Algorithm

Direct implementation of the Smith-Waterman algorithm (Fig. 1) led to 40 MHz frequency of the main clock. To make algorithm faster, pipelining was used. Pipelining is a technique that can speed-up a complex operations by breaking it into smaller concurrently executing stages. The maximum speed of pipelined operation is as good as the slowest single stage, at the cost of lengthening number of clocks required to obtain completion of full complex operation. Pipelining can be used only while processing independent data. We need at least k independent data chunks, where k is the number of stages. The key observation in Smith-Waterman algorithm is that each cell in a matrix can be calculated using its only three neighboring cells (Left, Upper and Diagonal), so data laying on "diagonal" does not depend on each other as presented in the Fig. 2.

In real implementation we have more complex pipelining schema (compared to the three stages shown in Fig. 2). There are 7 stages, each dependent on calculations of the previous one: Move generator, Sequence reading, Score matrix reading, Maximum counting (takes 3 cycles), Global Max finder. Additionally, temporal signals update is made. All pipelining stages are executed concurrently, synchronized by clock signal, but each stage computes different entry in a matrix. In this way, every clock cycle there is one new matrix entry which have passed all

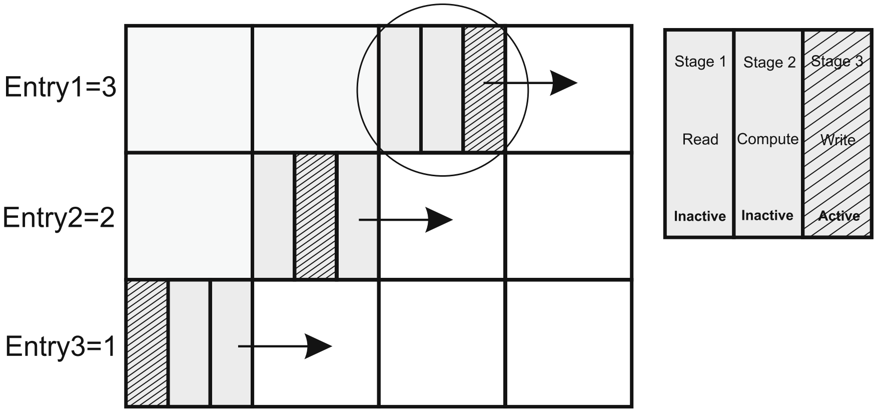


Fig. 2. Simplified view of the pipelining with three stages. For each matrix entry being calculated (shaded) there is stored its actual stage number executed: *(entry1)* is at stage 1, *(entry2)* at stage 2. Arrows indicate direction of movement. When an entry has passed all stages it moves to the next position in a row (if available) or advances to the next uncalculated row (shown on the Fig. 3)

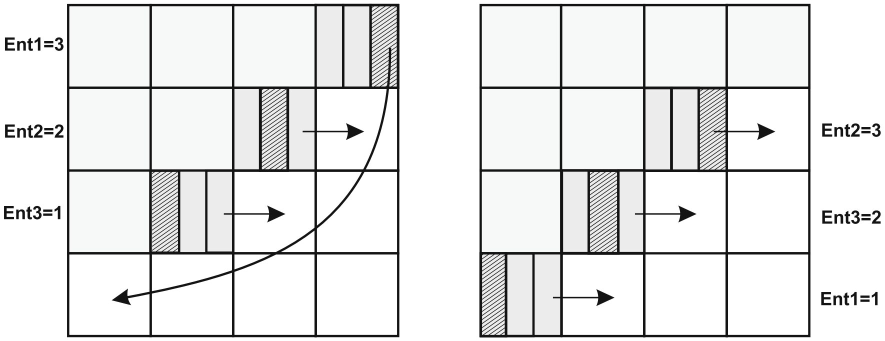


Fig. 3. Advancing to a new row: *(entry1)* has passed all stages and advances to the next uncalculated row, beginning with first stage

the stages of computation. Developed VHDL program uses a number of signals to keep track of calculations. More detailed view (showing only three entries being calculated) of temporal signals used in pipelining structure is presented in Fig. 4. In order to maintain boundary conditions properly, we use dummy buffer. It is needed when computations start from a new row, and during calculations of last rows. Each entry in a matrix needs information about its Upper, Left and Diagonal neighbor. For this purpose $H1, H2$ arrays are introduced. Depth of $H1$ and $H2$ is equal to number of stages and its width is 18 bits. These small arrays are implemented in Configurable Logic Blocks (CLBs). Each processing unit needs to know, which matrix entry is supposed to compute, so we need signals: $i_k, j_k, k = 1 \ldots 7$. The row c is red by $entry1$ and written to by $entry7$.

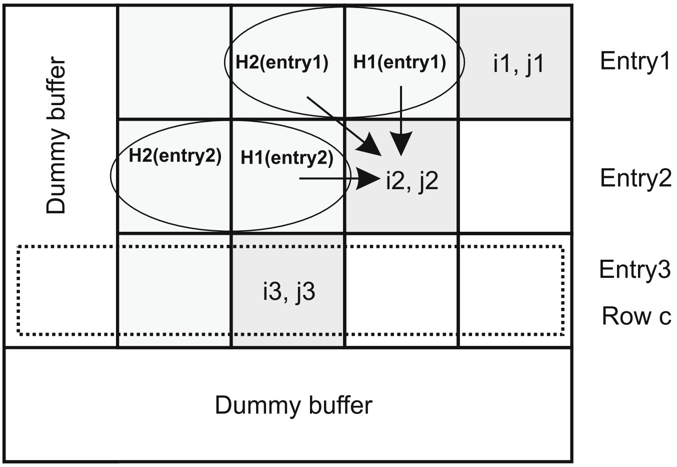


Fig. 4. How the history arrays $H1, H2$ and row c are kept and used: *(entry1)* uses the row c to retrieve its Upper, Diagonal neighbors when it advances to the next row. Other entries take this information from $H1, H2$ arrays. Row c is written by last entry. Dummy buffer filled with zeros is kept to maintain boundary conditions. In this way it is sufficient to store only one temporary row c, despite many stages

Information about neighbors is taken from arrays $H1, H2$ in a case of entries $2 \ldots 7$, and from the row c in a case of $entry1$. When $entry1$ is in first row, it reads c initially set to zero, but when it moves to next uncomputed row (Fig. 3) c is filled by the information provided by $entry7$.

Data Storage. Let us look at the details of the Xilinx Virtex 2 Pro implementation. Each aminoacid sequence is kept in 18 Kbit BlockRAM, which is true dual port memory with simultaneous read/write operations in each port. BlockRAMs are the key component of Xilinx devices, making it very suitable for the sequence alignment algorithms. Please note, that dual port BlockRAM memory is able to read from and write to a different address in a single clock. This enables to stages execute in parallel during reading and writing to a row c. Maximum BlockRAM density is found in Xilinx Virtex 2 Pro series (up to 444 in one chip), providing excellent maximum memory throughput. Large memory storage space is also needed for a row c, where temporal scores are kept. As it turn out, it is ineffective to keep aminoacid sequences of length 2048 and row c in distributed logic due to large number of CLBs utilized. This leads to low parallelism - we cannot implement many copies running in parallel within one chip. According to the algorithm properties we need five BlockRAMs for each algorithm copy: one for keeping edit score function ρ_{edit}, one for query sequence, one for database sequence, two for row c. Other signals are kept in CLBs. BlockRAMs can have different aspect ratio. In this implementation we store 2048 9 bit values, which is sufficient for keeping aminoacids. It is not enough for keeping temporary score in row c, so we use two BlockRAMs for that purpose, obtaining

18 bits. Assuming average protein sequence length $l = 300$ residues, one can see that I/O is not a bottleneck, because of quadratic complexity of the algorithm. If we implement 88 parallel PUs in FPGA we need to tranfer database sequences (query sequence is loaded once per database search) $88 * 5bit$ every 300 180 MHz cycles - giving about 30 MBytes/s transfer rate, and standard 32 bit/ 33 MHz PCI interface turns out to be sufficient. Data is transferred using second port of BlockRam used to store database sequence, which can be uploaded with a new database sequence independently of read performed during computations of current database sequence. For testing purposes we have used opencores [29] PCI interface and obtained 80 MB/s write transfer rate using Memec Design Spartan II PCI board.

Parallelism. Highly parallel FPGA implementation can be achieved at two levels. First is internal parallelism: by making many copies (up to 88) within one chip of the same processing unit executing the algorithm in order to process one fixed query sequence against many different sequences from a database. This is different parallelization method compared to often used systolic arrays, and leads to better PUs utilization. In systolic arrays average number of utilized PUs is dependent on sequence length [9].

Second level could involve hybrid solution: cluster of desktop computers, each equipped by FPGA accelerator in a form of PCI card. Not very intense communication between desktop computers can be performed by standard Ethernet. Parallelization level is limited by the FPGA size. Limiting factor in presented solution is the BlockRams count (5 BRams per PU). Control logic occupy about 44% of CLBs in Virtex 2 Pro when 97% of BlockRams are allocated (XC2VP70 chip). This explains difference in performance (Table. 1), between implementation in XC2V80 (contains 8 BRams) and XC2VP70 (contains 328 BRams).

5 Benchmarks

We used the following combination of software. Logical synthesis by Synplicity Amplify 3.5 [28] program. Place and Routing by ISE Xilinx tools [27]. Detailed timing analysis on post and route implementation revealed that the clock frequency is about 180 MHz on (-5) device speed version.

For the comparison purposes we have evaluated speed of widely used software from Pearson [19] FASTA 3.4 package - OSEARCH (dropnsw.c,v 3.4t23 March 5, 2003) running on Pentium IV 1.7 GHz CPU (gcc 3.2.2 , intel(c) 6.0 compiler) under Linux 9 and on SUN UltraSPARC-III 900 MHz with SUN Workshop 5.0 compilers. Best result are summarized in Table. 1. Please note that we have omitted Intel(c) icc compiler results, because it did not perform any better in this case (possibly due to lack of floating point operations). Benchmark was performed on aminoacid sequences of length 2048 residues.

Our program has been successfully verified by implementation on Memec Design Spartan II FPGA board, in limited version due to Spartan II architecture. Limitations apply to sequence length of 512 residues (smaller BlockRams).

Table 1. Comparison of various implementations of Smith-Waterman algorithm

Hardware	**Compiler**	**Program**	**Residues /s**
Pentium IV 1.7 GHz	gcc -O3 (3.3.2)	osearch34 -3	$49.30 \cdot 10^6$
UltraSPARC-III 900 MHz	cc -fast -xarch=v9a	osearch34 -3	$37.60 \cdot 10^6$
XC2V80-5	Amplify 3.5	this work	$180 \cdot 10^6$
XC2VP70-5	Amplify 3.5	this work	$11180 \cdot 10^6$

Implementation in currently available FPGA XC2VP70 turns out to be at least 200 times faster than Pentium IV 1.7 GHz.

6 Related Work

Acceleration of DNA sequence alignment using FPGAs and VLSI devices has a long history: VLSI implementation [12], SPLASH [11] and SPLASH2 [10] (computes simple edit distance for both 2 and 4 bit alphabet) system based on FPGA, however little attention has been paid to a more general case of protein seqence alignment with full 8 bit substitution matrix, where the computational problem does not reduce to simpler formulas used in LLCS computations. Moreover, it would be interesting to compare those DNA versions with accelerated by bitvector algorithms working on general purpose computers [16].

Direct comparison is quite difficult, because almost each implementation reported in literature is prepared for different parameter sets and is working on different hardware (different FPGA chips, VLSI or software). We list some of the implementations and compare its area of applications.

First group of implementations calculate DNA simple edit distance, mainly using systolic array parallelization. One of the latest achievements in DNA sequence alignment is presented in [1], where impressive number of 4,032 PEs were placed on an XCV1000E-6 device running 202 MHz, without the need for runtime reconfiguration. Similiar density is achieved using runtime configuration in [4]. Design working on 8 bit alphabet, but with constant value for a character mismatch is presented in [2]. Another solution based on FPGA is presented in [5], which is able to compute DNA simple edit distance score and the optimal alignment.

Second group involves existing implementations of more general case of the Smith-Waterman algorithm with full similarity matrix. [7,8,9,13] is a design using VLSI chips and is able to use full substitution matrix. Interesting FPGA design is presented in [3], but the sequence being compared is compiled into the design, which may be impractical for large FPGA chips. Commercial VLSI hardware is provided by Paracel [15]. Software implementation is given by commercial Celera [15]. The other known implementation is presented by Timelogic [26]. Their FPGA based machine Decypher is a commercial product, and details of their implementation are not published. Sencel company [6] sells interesting software solution, using MMX and similiar extended instruction set of general purpose processors to achieve parallelism, but it is limited to 8 bit score in order to achieve 6 times faster computations.

7 Conclusions

DNA alignment with a simple edit score problem has a long history and there exist a number of efficient FPGA implementations.

We investigate the possibility of implementing more general case of the Smith-Waterman algorithm allowing alignment of protein sequences (20 letters alphabet) with full substitution cost matrix (400 entries, 8 bits each) and 18 bits of alignment score using Xilinx Virtex 2 pro FPGAs.

The algorithm has been adopted to FPGA architecture which resulted in good parallelization and efficient pipelining. Presented benchmarks show that our implementation can run on a modern FPGA chip over high speed which results in 200 times faster execution compared to the standard OSEARCH program compiled on a PC workstation.

Future work will develop new more complex and faster homology search algorithms suited for FPGAs. Number of such algorithms has been developed for general purpose processors [19,20,21,22,24], but most of them has significant memory requirements and cannot be directly implemented in FPGAs.

Other interesting topic is multidimensional sequence alignment [23], which is useful in determining functional relationships, but computationally very demanding.

References

1. C.W. Yu, K.H. Kwong, K.H. Lee and P.H.W. Leong. A Smith-Waterman Systolic Cell.*Proceedings of the Tenth International Workshop on Field Programmable Logic and Applications (FPL'03)*, Lisbon, pp. 375-384, 2003
2. B. West, R. D. Chamberlain, R. Indeck, and Q. Zhang. An FPGA-based Search Engine for Unstructured Database. *Proc. of 2nd Workshop on Application Specific Processors*, December 2003.
3. N. Weaver, Y. Markovskiy, Y. Patel, J. Wawrzynek. Post Placement C-slow Retiming for the Xilinx Virtex FPGA. *11th ACM Symposium of Field Programmable Gate Arrays (FPGA)* , 2003
4. S. A. Guccione and E. Keller. Gene matching using JBits. *Field-Programmable Logic and Applications, Reconfigurable Computing 12th International Conference*, p. 1168-1171, September 2-4, 2002.
5. Y. Yamaguchi, T. Maruyama and A. Konagaya. High Speed Homology Search with FPGAs. *Pacific Symposium on Biocomputing* 7:271–282, 2002.
6. T. Rognes and E. Seeberg. Six-fold speedup of Smith-Waterman sequence database searches using parallel processing on common microprocessors.*Bioinformatics*, Vol. 16 no. 8, 2000, p. 699-706
7. D. Lavenier. Speeding up genome computations with a systolic accelerator. *SIAM News*, vol. 31, no. 8, Oct. 1998.

8. J. D. Hirshber, R. Hughey, K. Karplus. Kestrel. A Programmable Array for Sequence Analysis. *Proc. Int. Conf. Application-Specific Systems, Architectures, and Processors, IEEE CS*, Aug. 19-21, 1996, pp. 25-35
9. D. Lavenier. SAMBA: Systolic Accelerators for Molecular Biological Applications, *IRISA Report, (PI-988)* , March 1996.
10. D. T. Hoang. Searching genetic databases on splash 2. *Proceedings 1993 IEEE Workshop on Field-Programmable Custom Computing Machines*, p. 185-192, 1993.
11. Hoang, Dzung T. FPGA Implementation of Systolic Sequence Alignment. *International Workshop on Field Programmable Logic and Applications*, Vienna, Austria, Aug. 31 - Sept. 2, 1992.
12. R.J. Lipton, and D. Lopresti. A systolic array for rapid string comparison. *Proceedings of the Chapel Hill Conference on VLSI*, p. 363-376, 1985.
13. Paracel, inc. http://www.paracel.com.
14. Sencel's search software. http://www.sencel.com.
15. Celera genomics, inc. http://www.celera.com.
16. M. Crochemore, C.Iliopoulos, Y. Pinzon, J. Reid. A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem. *Information Processing Letters*, Vol. 80, Issue 6, p. 279 - 285, Dec. 2001
17. T. F. Smith, M. S. Waterman. Identifcation of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.
18. M. S. Waterman. *Introduction to Computational Biology: Sequences, Maps and Genomes.* Chapman and Hall, London, 1995.
19. W. R. Pearson. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics* 11(3):635-650, 1991
20. W. R. Pearson, D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA* 85(8), 2444–2448, 1988.
21. W. R. Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in Enzymology*, 183:63–98, 1990.
22. B. Ma, J. Tromp, M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics.* 18(3):440-5, 2002.
23. G. Z. Hertz, G. D. Stormo. Identifing DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics* 15(7/8):563–577, 1999.
24. A. Davidson. A Fast Pruning Algorithm for Optimal Sequence Alignment. *Proceedings 2nd Annual IEEE International Symposium on Bioinformatics and Bioengineering (BIBE 2001).* IEEE Comput. Soc. Los Alamitos CA, USA, pp.49–56, 2001.
25. S. Henikoff, J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. matl. Acad. Sci. USA.* 89, 10915-10919, 1992.
26. Timelogic home page. http://www.timelogic.com
27. Xilinx home page. http://www.xilinx.com
28. Synplicity home page. http://www.synplicity.com
29. Opencores home page. http://www.opencores.org

A Key Management Architecture for Securing Off-Chip Data Transfers

Jonathan Graf and Peter Athanas

Virginia Tech
Department of Electrical and Computer Engineering
340 Whittemore Hall, Blacksburg, VA, 24061, USA
{jgraf, athanas}@vt.edu

Abstract. Data security is becoming ever more important in embedded and portable electronic devices. The sophistication of the malicious techniques used by attackers is amazingly advanced. Defensive measures for protecting a device must be even more sophisticated and robust. This paper presents an architecture that manages cryptographic keys for a secure memory interface on an FPGA. The architecture includes functional units that serve to authenticate a user, create a key with multiple layers of security, and encrypt an external memory interface using that key. Cryptographic methods built into the system include an RSA-related secure key exchange, the Secure Hash Algorithm, a certificate storage system, and the Data Encryption Standard algorithm in counter mode.

1 Introduction

In today's world of advanced security cracking techniques, it is difficult to secure a digital device against unauthorized use or tampering. Companies and governments wishing to deploy digital hardware into situations where the device may be subject to malicious analysis risk losing secret algorithmic and functional information to competitive or hostile entities. There is a growing need for devices that are capable not only of authenticating a user to the device but also of masking the device function through cryptographic techniques. For example, a military unit may want to deploy a digital device running a secret algorithm in hostile territory without revealing the nature of the algorithm in the event the device is captured. Since the loss of a cryptographic key could compromise even the best of these devices, key management is integral to and arguably the most important aspect of any cryptographically secured system.

Attacks used to gain information from digital devices are commonly carried out on the authentication and memory systems. If the device's authentication system can be fooled into allowing unauthorized use, the process of further analysis is simplified. If a logic analyzer is placed on the address and data busses of an embedded device's external memory, the algorithmic function of the device can easily be compromised.

There are compelling reasons for choosing FPGAs as the technology for implementing a secure memory and key management scheme. Within a single FPGA, the user authentication system, the memory controller, and every necessary

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 33–42, 2004.

cryptographic algorithm can be realized. Such a system could cryptographically secure both the user authentication interface and the memory interface, effectively rendering the FPGA a black box capable of performing the task for which it was designed without betraying its internal methods or software to a hostile entity. Logic analysis of the authentication or memory interface would yield only encrypted information.

A secure key management system has been implemented using a Celoxica RC1000 development platform [1]. This prototype utilizes the key management system to secure a memory port on the FPGA, protecting the contents from discovery. Additionally, a Dallas Semiconductor Java-Powered iButton [2] is used as a secure token that can authenticate a user to the system by establishing a secure channel for conveying user identification data from the iButton's memory to the FPGA. On the FPGA, an Authentication Control Unit (ACU) establishes the other side of the secure channel, and a counter-mode Data Encryption Standard (DES) algorithm [3] is used to secure the RAM interface.

This paper presents this FPGA-based secure memory system from the perspective of its cryptographic key management scheme. Section 2 presents the general architecture of the FPGA functional units. Section 3 introduces the components used in the prototype. Section 4 discusses the cryptographic algorithms used to securely manage the keys as they pass from the iButton through the ACU to the encrypted memory controller (EMC). Section 5 details a vulnerability analysis for our system. Section 6 presents the level of completion of the current implementation.

2 Architecture

A block diagram of the entire architecture is shown in Figure 1. When a user wishes to be authenticated and use the device, they will plug an authorized iButton into a receptacle connected to the 1-Wire interface. The iButton then communicates with the ACU serially via the auxiliary I/O pins on the Xilinx Virtex XCV2000E FPGA [4]. A Dallas Semiconductor DS9097U Universal COM Port Adapter [5] translates between the iButton's 1-Wire serial protocol and the serial transmit and receive protocol expected by the UART in the FPGA. The link between the FPGA and the iButton is an exposed and potentially vulnerable interface. Measures must be taken to protect the presumably exposed data transferring across this link.

The ACU is responsible for establishing a secure channel with the iButton to retrieve the iButton's user identification (UID) and check it to ensure that the iButton holder is an authorized user. The UID is used by the EMC to create a DES key that is unique to the current user. This final DES key is used as the key for encrypting the data traveling between the embedded application and the external RAM. Once the final DES key is established in the ECU, the embedded application is signaled to allow it to begin secure operation. Section 4 presents this architecture and its cryptographic methods in detail.

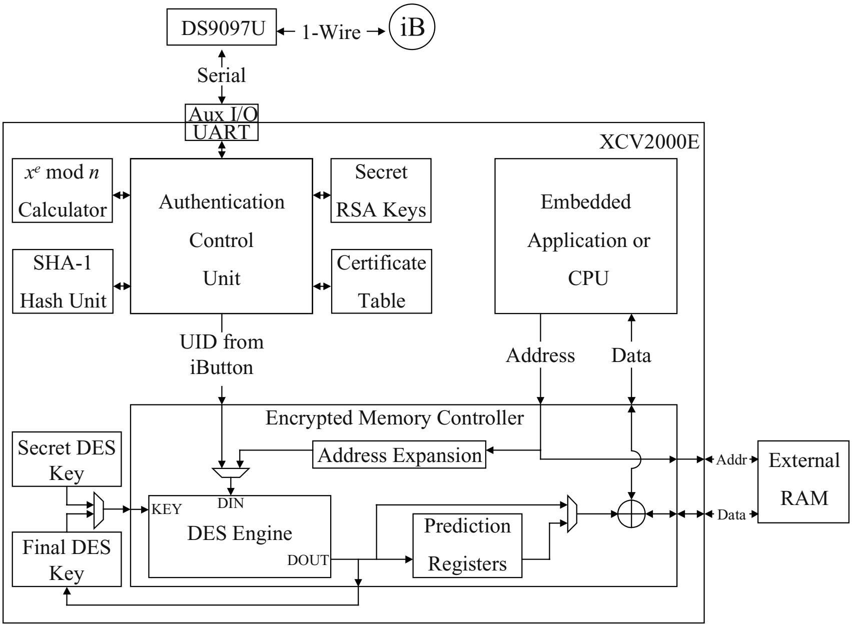


Fig. 1. Block diagram of system architecture

3 Platform

A block diagram of the Celoxica RC1000 platform is shown in Figure 2. The RC1000 is a PCI card that can interface with a Windows or Linux host PC. The RC1000 version used in this prototype contained an XCV2000E FPGA. It also includes four 512k×32 SRAM banks, each with their own address, data, and control interfaces to both the FPGA and the PCI bus. The PCI bus interface also includes control and status signals to the FPGA. A program running on the host PC can pass data to and from the SRAM interfaces with direct memory access transfers, control the FPGA clock, and communicate directly with the FPGA through the control and status signals. The RC1000 also includes fifty auxiliary I/O pins that allow direct connection to the FPGA's external I/O pins.

In our design, the Dallas Semiconductor Java-Powered iButton model DS1957B serves as a secure token. The DS1957B is packaged in a cylindrical stainless steel canister 5 mm high and 16 mm in diameter, and resembles a watch battery. Internally, it consists of a Java processor, a 1024-bit cryptographic math accelerator, a random number generator, 134-kilobytes of non-volatile RAM, and a serial communications interface for communicating over its proprietary 1-Wire interface protocol. Communications on this 1-Wire interface also serve to power the iButton whenever it is plugged into a 1-Wire compatible receptacle. The iButton's construction is physically tamper resistant due to a mechanism that will destroy the contents of its memory if its case is opened.

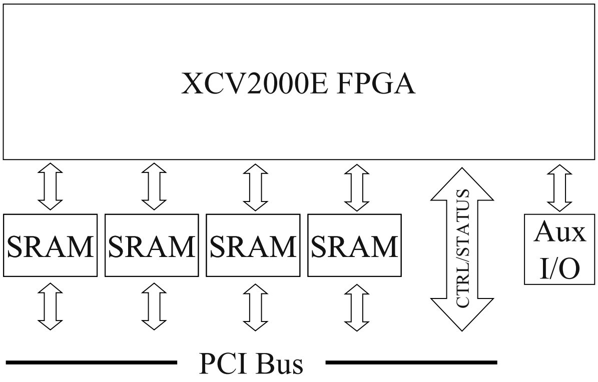


Fig. 2. Block diagram of Celoxica RC1000 with XCV2000E FPGA

Java applets can be programmed using Dallas Semiconductor's iButton Integrated Development Environment (iB-IDE) [7] and uploaded to the iButton. In our system, we have written an applet using iB-IDE to control the iButton's role in establishing a secure communications protocol between the iButton and the FPGA.

4 Key Management

Key management is the term used to describe the path that the UID travels through the secure authentication interface to the DES engine to become the final key used for memory encryption. A major theme in the proposed key management scheme is keeping the secret information on the iButton separated from the secret information on the FPGA to ensure that if either the iButton's or the FPGA's individual security methods are broken, the entire system will not be violated. This rule is enforced throughout the following key management units.

4.1 Authentication Control Unit

The ACU and the iButton negotiate a secure channel and establish each other's identities using an encryption, authentication, and certificate-checking system similar to the RSA-based [8] Public Key Infrastructure (PKI) [9]. There are a few significant differences. First, the public encryption keys for the iButton and the FPGA are never transmitted over an insecure channel. Since it is essential to our system that all keys remain secret, they will not be referred to as public and private keys. Those familiar with RSA encryption methods will recognize the encryption key as RSA's public key and the decryption key as RSA's private key. Second, certificates in our system are Secure Hash Algorithm (SHA) [10] hashes of valid iButton UIDs. SHA is a one-way hash, meaning that it is mathematically difficult to calculate the input to the hash using only the result of the hash. This makes it safe to store the hash results of the UIDs in a certificate table outside the iButton. This table could reside directly on the FPGA, which would require modifying the FPGA bitstream every time a new certificate was issued to an authorized user, or it could be retrieved by some other means from a trusted certificate authority. Our design places it within the FPGA.

There are two pairs of encryption and decryption keys and two moduli used in the authentication scheme. Prior to the programming of the iButton and the creation of the FPGA bitstream, four large prime numbers are chosen to create these key pairs and moduli. These primes, p_i, p_f, q_i, and q_f, are combined to form the moduli, n_i and n_f, through multiplication.

$$n_i = p_i q_i \tag{1}$$

$$n_f = p_f q_f \tag{2}$$

The encryption keys, e_i and e_f, are then chosen to be relatively prime to $(p_i–1)(q_i–1)$ and $(p_f–1)(q_f–1)$, respectively. Using e_i and e_f, the decryption keys, d_i and d_f, can be calculated.

$$d_i = e_i^{-1} \bmod (p_i–1)(q_i–1) \tag{3}$$

$$d_f = e_f^{-1} \bmod (p_f–1)(q_f–1) \tag{4}$$

In this way, a key pair and a modulus have been established for both the iButton—n_i, e_i, and d_i—and the FPGA—n_f, e_f, and d_f. All these values are considered secret. They are calculated before the iButton and FPGA designs are implemented and never transmitted over a clear channel. These secret keys and moduli are stored on the FPGA and iButton as shown in Table 1.

Table 1. Storage of secret keys and moduli.

FPGA	IButton
n_i, n_f, e_i, d_f	n_i, n_f, e_f, d_i

With the encryption key, e, the decryption key, d, and the modulus, n, chosen in the same way, the general RSA encryption algorithm computes an encrypted message, c, from a clear-text message, m, using using e, through a modular exponentiation.

$$c = m^e \bmod n \tag{5}$$

The RSA algorithm decrypts that message using d.

$$m = c^d \bmod n \tag{6}$$

The proposed authentication method uses this same relationship between the keys to encrypt communications between the iButton and the FPGA. This relationship has been thoroughly examined and proven mathematically.

In the prototype system, it is not only important for the iButton to authenticate itself as an authorized user to the FPGA, but the FPGA must also authenticate itself as an authorized host to the iButton. This prevents a man-in-the-middle attack wherein a device would pretend to be the the FPGA to the iButton while simultaneously pretending to be the iButton to the FPGA in an effort to steal the iButton's UID and falsely authenticate itself to the FPGA. Taking this double authentication into account, the ACU-iButton authentication method goes through the following steps:

1 iButton is plugged in to the 1-Wire interface.
2 FPGA recognizes presence of iButton.
3 iButton generates a random number, m_i, encrypts it using e_f and n_f, and sends the encrypted result, c_i, to the FPGA.
4 FPGA uses d_f and n_f to decrypt c_i and retrieve m_i.
5 FPGA uses e_i and n_i to encrypt m_i and sends the encrypted result, c_f, to the iButton.
6 iButton decrypts c_f using d_i and n_i and confirms that the FPGA has correctly returned m_i. If so, the FPGA has authenticated itself to the iButton, and the process continues. If not, the iButton does not respond to the FPGA and waits to be disconnected.
7 iButton encrypts the value of (UID + m_i) using e_f and n_f and sends the encrypted result, c_{UID}, to the FPGA. The UID is not encrypted directly, or the UID packet would be the same for every authorization session with that iButton.
8 FPGA uses d_f and n_f to decrypt c_{UID} and retrieves UID by subtracting m_i from the decryption result.
9 FPGA runs UID through the SHA hash and treats the result, h_{UID}, as the iButton's certificate.
10 FPGA checks h_{UID} against the authorized certificates in its certificate table. If it finds a match, the iButton has been authenticated to the FPGA. If not, the iButton represents an invalid user, and the FPGA waits for the iButton to be disconnected.

Once the iButton is authenticated to the FPGA and the FPGA to the iButton, the UID is sent to the EMC to create the final key for the DES engine.

4.2 Encrypted Memory Controller

Upon startup, the EMC uses a secret DES key that is unique to and known only to the FPGA. This key is used only in the creation of the final DES key, never for actually encrypting the external memory interface. The final DES key used for encrypting the external memory interface is formed by passing the UID from the iButton through the DES engine using the secret DES key. The first 56 bits of the 64-bit result of this operation are used as the final DES key. This is another example of the adopted philosophy of keeping the information necessary to crack the entire system separated between the iButton and the FPGA. Secret information from the iButton and the FPGA is required to calculate the actual key used to encrypt the memory interface that protects the embedded application data.

Once the final DES key is calculated, the EMC is ready for encrypted read and write operations to and from the external memory, and the embedded application or CPU is signaled to begin operation. The encryption method used by the EMC is related to DES counter-mode encryption [11]. In DES counter-mode encryption, a counter is placed as the input data to the DES engine, and the DES-encrypted result is used in an XOR operation with the data to be encrypted. Decryption in counter mode is similar. To decrypt, the same counter value is encrypted by the DES engine using the same key, and this number is used in an XOR operation with the encrypted data to retrieve the original data.

Similarly, when an address and a 32-bit data word are written to the EMC from the embedded application, the address is expanded using any 1-to-1 function to 64 bits and used as the input to the DES engine, using the final DES key as the key. The result of this is a 64-bit encrypted value. The 32-bit data word then goes through two XOR operations, one with the most significant 32 bits of the 64-bit DES result, the other with the least significant 32 bits. The result of these operations is an encrypted 32-bit number that is written to the original address value in the external memory. Figure 3 illustrates this encrypt-and-write operation. The original address bus width is not specified since the same method could be used for any size of address bus.

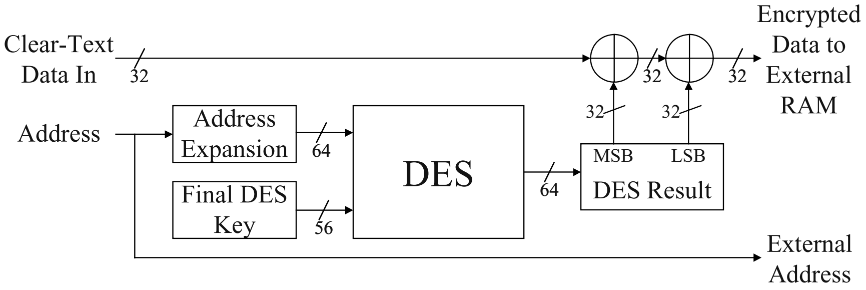


Fig. 3. EMC encrypt-and-write operation.

The read-and-decrypt operation is exactly the same process for the DES engine, but the 32-bit input data going through the two XOR operations is the encrypted data from the external RAM. The output is the clear text 32-bit data value requested by the embedded application. Figure 4 illustrates the read-and-decrypt operation.

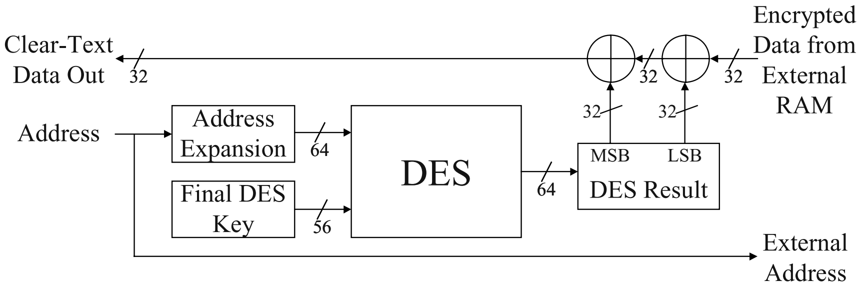


Fig. 4. EMC read-and-decrypt operation.

Note that for both the encrypt-and-write and read-and-decrypt operations, only the DES encryption function is required, since the same DES result is used both for encryption and decryption of the values written to memory at any given address. The DES decryption function is never used.

There is one layer of complexity in the prototype system that is not illustrated in Figures 3 and 4. As shown in Figure 1, there is a set of prediction registers in the

EMC that hold DES result values for memory addresses predicted to be the next accessed. The reason for having the prediction registers is that the DES engine has a latency of sixteen clock cycles between the time the first value is clocked into the DES input and the time the result is present on the DES output. However, since the DES engine is a pipeline, it can output a new result on each clock cycle after the initial sixteen-cycle latency. Keeping in mind that a memory controller needs to run at the highest possible speed, the pipelined nature of the DES engine makes it expedient to continue running predicted addresses through the DES engine after the requested address has been processed. Mathematically, the time, t_{DES}, in clock cycles, that it takes to calculate a given number of pipelined address values, n_{ADDR}, can be described as

$$t_{DES} = 16 + (n_{ADDR} - 1). \tag{7}$$

In the prototype system, we expect that the behavior of the embedded application will follow the principle of locality of reference, meaning that the next address referenced will be very near the most recent address referenced. Following this guideline, when an address is run through the DES engine, the prediction registers are filled with DES results for the sixteen addresses above and the sixteen addresses below the currently requested address. Upon the next address reference, the address is checked to see if it is in the range of addresses whose DES results already reside in the prediction registers. If it does, the pre-calculated DES result is retrieved from the correct prediction register, and the DES engine is not used. The sixteen-cycle latency is avoided. If the address does not fall in the range of the prediction registers, the DES engine is used to calculate the DES result, and the prediction registers are filled with the DES results for the addresses above and below this new address.

The prediction registers are designed to be filled without delaying the rest of the operation of the EMC. If a data is read from or written to an address not in the prediction registers, once the DES result is calculated for the requested address, the prediction registers are filled independently while the rest of the read or write cycle takes place.

In a more complex RAM system that includes a cache, the locality of reference principle is made slightly more complex. The prediction registers should hold the DES results for the data held in the cache lines. This new design would have to take into account more complex cache functions such as flushes. Similarly, in systems where the behavior of the embedded application could be easily predicted, the principle guiding the filling of the prediction registers should be modified accordingly.

5 Vulnerability Analysis

The system's vulnerability to probing and cryptanalysis has been considered in detail. An authorized user (user with an authorized iButton) can use the iButton to start the embedded application on the FPGA but can gather neither the value of their own authorized UID from the iButton nor the values of the data in RAM external to the FPGA without breaking RSA and DES, respectively. RSA's strength is in the difficulty of factoring its modulus, n, so a security analysis would need to be done prior to deploying the system to determine an appropriate size for n. It has been shown that DES is becoming increasingly vulnerable to differential cryptanalysis as computers become more adept at solving large numerical analysis problems [12]. A major advantage of the prototyped system is that any stronger keyed block cipher, such as counter-mode Advanced Encryption Standard [13] or Triple DES [14], could replace the DES engine in our current design. FPGAs add a higher degree of security by allowing cryptographic updates as needed to face new security threats.

If the physical security measures in the iButton, mentioned in Section 3, were broken and the contents of the iButton were known, an attacker would know an authorized UID, both RSA moduli, the iButton's decryption key, and the FPGA's encryption key. The attacker could then potentially negotiate an unauthorized session on the FPGA; however, the memory interface and the details of the embedded application data would still be secure due to the secret DES key's role in creating the final DES key from the UID.

Only an attack that intercepts the contents of the FPGA would unravel the entire key management scheme. If the contents of the FPGA were known, the attacker would know the proper moduli and RSA keys to setup an unauthorized session with the iButton to steal the UID. The secret DES key would be known as well, allowing the attacker to construct the final DES key and analyze the memory interface. The data for all authorized iButtons would still be safe, however, since the certificate table only stores the SHA hash of the valid iButton UIDs.

It is important to recognize that knowing the contents of the FPGA would also completely reveal the details of the embedded application. It is assumed in the scheme presented that it is not possible to know the contents of the FPGA, and much work has been done by others in this area. For example, Xilinx has implemented a feature in the Virtex-II FPGA family that allows the designer to encrypt bitstreams using Triple DES [15]. The Virtex-II has a Triple DES engine on its die that decrypts the bitstream as it programs the FPGA.

6 Implementation

A proof-of-concept design that includes a working ACU and an EMC with a prediction register system has been built. The proof-of-concept ACU is capable of retrieving a UID from a Java iButton, hashing the UID, and checking the hash digest against the certificate table to authenticate the user. The proof-of-concept EMC was implemented as described in Section 4.2. The prediction register system in the proof-of-concept design reduced the per read/write cycle encryption delay to an average of only 2 clocks for the tested embedded application. This is a significant improvement over the 16-clock delay that would be required during every read/write cycle for the DES engine alone if the prediction register system was not present.

7 Conclusion

A secure key management architecture for an encrypted external memory interface on an FPGA has been presented. The scheme is secured using modified versions of proven cryptographic algorithms. The system is not only protected from unauthorized users, but its embedded application is also protected against analysis of its memory interface. This system is currently well suited to protect embedded applications and algorithms that will be deployed publicly under the scrutiny of competitive or hostile entities, and it is extensible to encompass future advances in block cipher algorithms to keep up with new security threats.

References

1. Celoxica Limited: RC1000 Hardware Reference Manual. Version 2.3. RM-1120-0. http://www.celoxica.com, 2001.
2. Maxim/Dallas Semiconductor Corporation: Java-Powered Cryptographic iButton. http://www.ibutton.com/ibuttons/java.html, 2003.
3. National Institute of Standards and Technology (NIST): FIPS Publication 46-2. Data Encryption Standard,1993.
4. Xilinx, Incorporated: Virtex-E 1.8V Field-Programmable Gate Arrays. http://www.xilinx.com/bvdocs/publications/ds022-1.pdf, 2002.
5. Maxim Integrated Products: DS9097U Universal 1-Wire COM Port Adapter. http://www.maxim-ic.com/quick_view2.cfm/qv_pk/2983/ln/en, 2004.
6. Chappell, S., Sullivan, C., “Handel-C for co-processing & co-design of Field Programmable System on Chip,” Workshop on Reconfigurable Computing and Applications, (JCRA), September 2002.
7. Maxim/Dallas Semiconductor Corporation: iB-IDE – New IDE for the Java-powered iButton. http://www.ibutton.com/iB-IDE/, 2003.
8. Rivest, R.L., Shamir, A., Adleman, L.M., “A method for obtaining digital signatures and public-key cryptosystems,” Communications of the ACM vol. 2, no. 21, pp. 120-126, 1978.
9. National Institute of Standards and Technology (NIST): NIST PKI Program. http://csrc.nist.gov/pki/, 2001.
10. National Institute of Standards and Technology (NIST): FIPS Publication 180. Secure Hash Standard, 1993.
11. Lipmaa, H., Rogaway, P., Wagner, D., “Comments to NIST concerning AES Modes of Operations. CTR-Mode Encryption. Modes of Operation for Symmetric Key Block Ciphers.,” First Modes of Operation Workshop, online at http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/lipmaa-ctr.pdf, October 2000.
12. Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer-Verlag, 1993.
13. National Institute of Standards and Technology (NIST): FIPS Publication 197. Advanced Encryption Standard, 2001.
14. National Institute of Standards and Technology (NIST): FIPS Publication 46-3. Data Encryption Standard, 1999.
15. Xilinx, Incorporated: Virtex-II 1.5V Field-Programmable Gate Arrays. http://www.ida.ing.tu-bs.de/service/download/DigSchalt/Zusatz/xilinx_virtexII.pdf, 2001.

FPGA Implementation of Biometric Authentication System Based on Hand Geometry

Celia López-Ongil, Raul Sanchez-Reillo, Judith Liu-Jimenez, Fernando Casado, Leslie Sánchez, and Luis Entrena

Microelectronics Group, Electronic Technology Department.
University Carlos III of Madrid.
Avda. de la Universidad, 30.
28911 Leganés. Madrid
Spain
{celia,rsreillo,jliu,fcasado,lsanchez,entrena}@ing.uc3m.es

Abstract. The increasing demand for pervasive security poses a challenge in achieving robust authentication at very low cost. Identification using human biometrics is considered the most robust solution, but requires powerful computers to be performed in acceptable time. In this work a FPGA implementation of a biometric authentication system based on hand geometry is presented. The system covers all necessary steps to process a human hand pattern and verify it against a user template. This solution is able to reduce processing time by three orders of magnitude with respect to microprocessor-based solutions of similar cost, while keeping the same identification quality. On the other hand, it can be implemented in a small size FPGA, thus making it suitable for a large number of low cost applications.

1 Introduction

The increasing demand for security in our society is driving attention towards more robust and efficient solutions for user authentication and identification. Biometrics has been traditionally considered one of the best solutions for human recognition. Biometric patterns are unique for each person and are more difficult to fake than others based on passwords and secret codes or those based on physical tokens.

Existing biometric systems are based on different biological or behavioural features such as speaker verification, signature recognition, fingerprint measure, iris pattern or hand geometry [1] and [2]. Among these, hand geometry has the advantage of being user-friendly and ergonomic. The quality of these techniques is represented by two ratios, False Acceptance Ratio (FAR) and False Rejection Ratio (FRR) which try to report the number of errors the system performs when recognizing users.

Biometric authentication systems require complex pattern recognition algorithms. These algorithms can be executed in today's computers within acceptable time. However, execution time is still unacceptable in low cost microprocessors. This fact prevents the widespread use of biometric systems in an important market that requires very low cost systems and includes applications such as ubiquitous access control,

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 43–53, 2004.

smart cards, etc. In order to satisfy this demand, low cost systems with higher performance are needed.

In this work, an efficient hardware implementation of a biometric authentication system based on hand geometry is proposed. This implementation provides an improvement of three orders of magnitude in processing time and can be downloaded in a small FPGA (ASICs could be used if the product volume is large enough). In addition, it can also be used to improve performance for systems that require verification with respect to a large user database.

This work has been partially funded by Spanish Government, through the research project TIC-2003-01793 SIDECAR, (Identification and Communications Systems through Reconfigurable Hardware)

The paper is organized as follows. In section 2, main aspects of the hand geometry authentication algorithm are explained. Section 3 details architecture implementation. Results are shown in section 4 and, finally, conclusions are stated in section 5.

2 Hand Geometry Recognition

Hand geometry recognition systems are quite good for environments where medium-high security is required, and where a medium cost equipment (only a low resolution CCD camera is needed), relative low computational cost (because the algorithms to extract the features are based on basic morphological image processing [3],[4] and [5]) are needed. Finally, it implies a very low feature vector size.

A typical biometric recognition system is normally structured in two phases, namely enrollment and verification, Fig. 1. The enrollment process consists in taking various samples of a user pattern and generating a user template that it is stored for eventual comparison. The authentication process is launched every time a user wants to gain access to the secure environment, and decides whether the current user data corresponds to the user template. Currently, *user templates* are stored in smart cards which act as physical tokens.

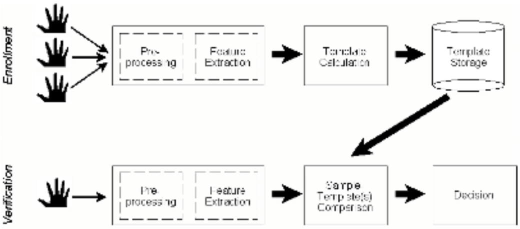


Fig. 1. Hand Geometry Recognition System

The authentication process involves three main steps. First, the user pattern must be captured and preprocessed in order to enhance the features of interest. The second step is feature extraction. Finally, the current features are compared with the template features in order to authenticate the current user. Our hand geometry recognition is based on the algorithm proposed in [6] and [7], and the main steps are described in the sequel.

Image Capture

In this step an image of the hand is taken using a low resolution CCD camera. The hand is placed on a platform guided by 6 tops, in order to position the hand in front of the camera objective. Different views of the prototype designed can be seen in Fig. 2.

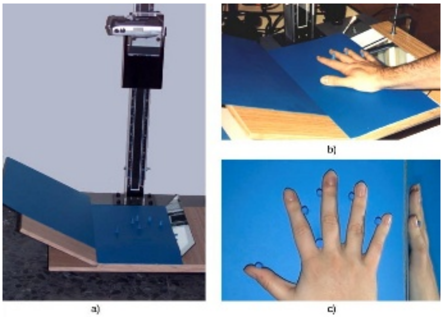


Fig. 2. Prototype developed (a. general view; b. placement of the hand; c. sample taken)

A mirror is located on one side of the platform for obtaining a side view of the hand and performing more measures. The platform is painted in blue to increase the contrast with the human skin. Images taken are RGB with 480x640 pixels (24 bits).

Data Preprocessing

This step converts input data into manageable information where biometric features could be easily extracted. The algorithm transforms a color image of the user's hand into a hand profile. First, a grey-scale image of the hand is generated in order to eliminate all unnecessary information (background). Secondly, a *black and white* image is obtained in order to generate, finally, a hand profile with an edge detection filter.

Grey Scale Conversion

The original algorithm applies the equation (1), in order to eliminate blue color (background) from the image and to obtain a grey-scale one. Normalizations are done in order that histogram can be expanded. Resulting image has 480x640 pixels (8 bits pp).

$$\left((R_n + G_n)_n - B_n\right)_n \ . \tag{1}$$

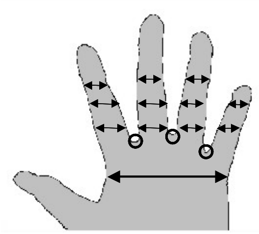
Black and White Conversion

This process converts the grey-scaled image into a *black and white* one. A threshold is applied in order to decide if the pixel is black (hand) or white (background). The result is an image of 480x640 pixels (1 bit pp).

Edge Detection
This process applies filters to generate a hand profile that enables the measuring of the main features. Sobel and Laplacian filters have been considered in this work. Both of them consist in detecting pixels surrounded by different colored pixels. A mask matrix is applied on every pixel. The results obtained by Sobel filter are better for noisy input images but the profile has wider edges.

Feature Extraction
In this step geometrical measures are performed, from the hand profile, to obtain a set of 25 features. These features have been previously defined and stated as the most representative set of measurements, [1] and [2], and include:

- Widths of the 4 fingers (thumb is excluded)
- Width of the palm
- Height of the palm, the middle finger and the small finger
- Distances between the finger joints
- Angles between the finger joints



To avoid changes due to gain or losing of weight, all the measures (except the angles) are normalized by the first width measured of the middle finger. The result is a feature vector with 25 components, each of them coded in 1 byte.

Feature Matching
Once pattern features are available, they are compared with template features in order to obtain the distance between them. Euclidean distance, Hamming distance, etc. could be applied depending on the degree of security that is considered, [6] and [7].
With the distance obtained from the comparison between template and measurements, the recognition system must decide user access to the secure environment. A threshold is set in such a way that FAR as well as FRR are minimized. This threshold must be statistically calculated during enrollment process.
In this biometric technique Continuous Hamming distance is applied on the 25 features obtained from the user's hand image. Considering r^1 and r^2, pattern features and user template features respectively, and b typical deviation of every data, Hamming distance, $d(r_1,r_2)$, is defined as shown in equation (3).

$$r^1=[K_{11},K_{12},...,K_{1n}] \cdot \qquad r^2=[K_{21},K_{22},...,K_{2n}] \cdot \qquad b=[b_1,b_2,...,b_n] \cdot \tag{2}$$

$$d(r_1,r_2)=\sum_{j=1}^{n}\delta(K_{1j},K_{2j}) \tag{3}$$

$$\delta(K_{1j},K_{2j})=0 \;\; if \;\; |K_{1j}-K_{2j}|\leq b_j \qquad \delta(K_{1j},K_{2j})=1 \;\; if \;\; |K_{1j}-K_{2j}|>b_j \tag{4}$$

3 System Architecture

In order to improve the performance of the hand geometry recognition algorithm, a hardware implementation is proposed. Fig. 3a shows the architecture of this hardware implementation. It contains four main blocks, according to the steps of the algorithm, preprocessing (*Grey Scale* and *Edge Detector*), feature measuring and feature matching. Also, the architecture has three interfaces for receiving user pattern and user template, as well as for storing temporary data in external memory banks. An internal RAM memory is used to store temporary data in the latest steps of the algorithm execution.

In the hardware implementation information storage is a key factor due, mainly, to the large amount of data to be processed. It is important how and where the information is stored, as it could be seen in Fig. 3b where storage blocks are filled. In this implementation partial and global results are stored in RAM blocks, external or internal to the circuit. The use of various banks of memory and fast memories has provided short access times and a small number of memory accesses.

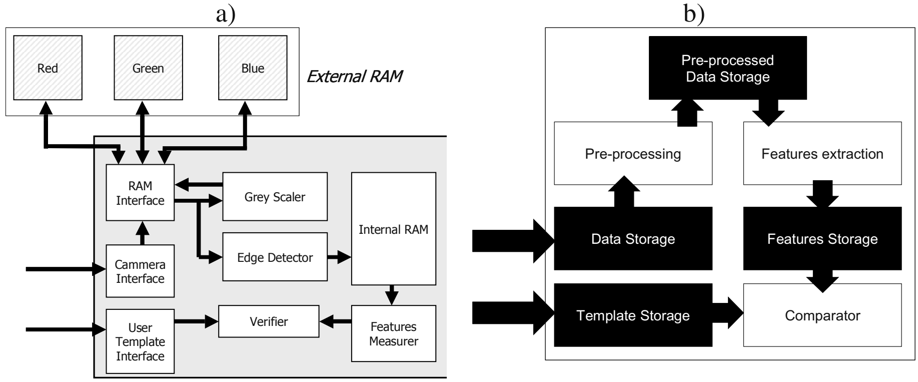


Fig. 3. a) Architecture of the proposed hardware implementation b) Storage blocks

In order to generate a high speed application, faster operators, parallelism and pipelining have been applied. Finally, it is important to set the precision required not only for achieving good results in the information processed but also for reducing resources employed. Arithmetic operations are performed with full precision. Regularly it is not possible to store the amount of information generated and results data will be truncated in order to be stored in memory.
The FPGA implementation of the hand geometry recognition technique is described in this section. The resulting system is analyzed with respect to the enhancements obtained in the time domain, in the resources used and in the cost achieved. Also, the accuracy terms FAR and FRR is taken into account. These ratios are detailed in section 4.

Data Storage

This first step in the processing receives input data, which is a hand image (RGB) of M pixels (8 bits per pixel), being M=480x640. In the following M will state for 480x640, to represent the number of pixels.

Image is coming from a digital camera and it is stored in external memories. In the process of downloading, some values are calculated that could be applied in latter processes, such as normalizations (e.g. R_n).

The use of three banks of memory reduces the number of memory accesses from 3M to 1M. On the other hand, accuracy is maintained and resources needed are a Data Path and an Address Generator.

Black and White

The original algorithm for grey scale conversion applies the equation (1). In this work, equation (5) has been implemented, in order to reduce memory accesses as well as arithmetic operations.

$$(R+G)_n - B_n \,. \tag{5}$$

Table 1 shows the complexity of the hardware and software implementations as a function of M. Second column contains cycles needed for arithmetic operations, while third and fourth are for memory readings and writings. Last column shows total cycles required in each implementation. As it can be seen, FPGA implementation reduces memory accesses as well as cycles employed in arithmetic operations, thanks to parallelism and pipelining. Pixels are processed in one clock cycle with a latency of 5 cycles. It should be noted that in FPGA implementation these arithmetic operations are performed in parallel with memory readings and writings, so as total execution times is even less.

Table 1. Cycles required in preprocessing hand image in FPGA and software implementation

Algorithm	*Implementation*	*# Arithmetic Operations*	*#Memory Readings*	*#Memory Writings*	*Total*
(1)	*HW (FPGA)*	3M	3M	3M	6M
	SW (uP)	12M	12M	7M	31M
(5)	*HW (FPGA)*	1M	1M	1M	2M
	SW (uP)	6M	6M	4M	16M

Once the grey scale image is obtained, a *black and white conversion* is performed, by the application of a threshold. This last step produces images of 480x640 pixels (1 bit pp) which are stored in an internal memory of the circuit.

With respect to the accuracy, the comparison between results given by the FPGA and the software implementation has shown slight differences. Fig. 4a shows a *black and white* image generated by the hardware block. In Fig. 4b, differences with software application image could be seen. The average error is less than 2%. This error is due to internal precision, 16 bits in hardware and 64 bits in *Matlab*®.

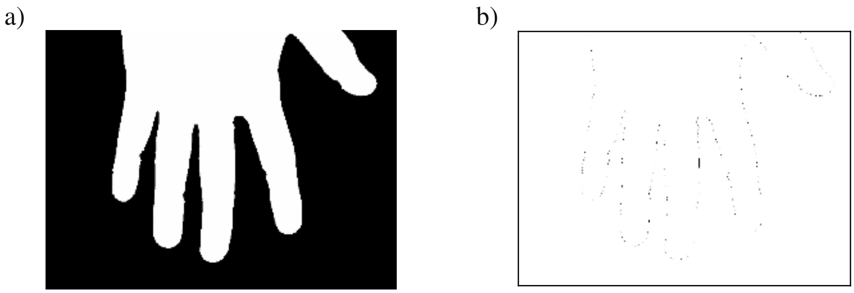


Fig. 4. a) *Black and white* image. b) Difference between results given by hardware and software implementation

Edge Detection

In this step, filter application consist in matrix multiplication (3x3) on every pixel of the *black and white* image. As masks are composed of constant values, multiplications become very simple.

The main bottleneck in this task is memory access to read the whole image. It is necessary to read 9 values per pixel, so 9M reads are needed. This is solved by using 3 banks of memory to store the image. If the *black and white* image is stored in the three banks at a time, only one cycle is needed to read three values. Also, if values used for the previous pixel processing are maintained, only three new values are needed for the next. With these two conditions, only M cycles are needed to operate pixels and M memory accesses are needed.

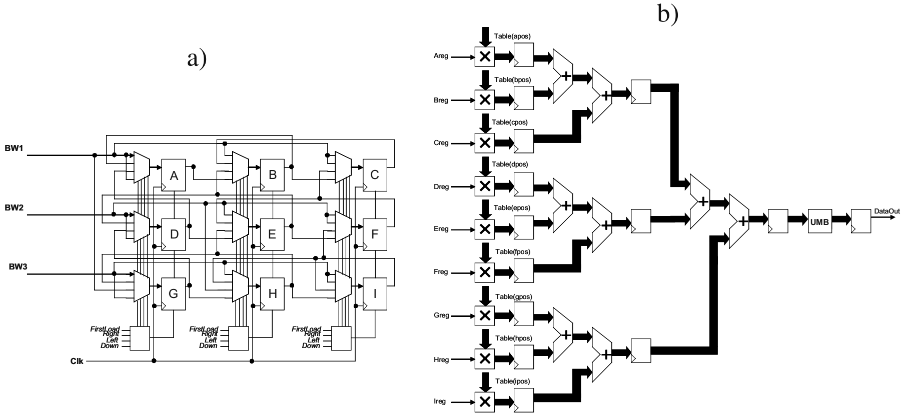


Fig. 5. Circular shift register and data path for edge detection

This solution has been implemented in the FPGA with a circular shift register which is able to store, rotate and shift values coming from the memories, shown in Fig. 5a. Parallelism and pipelining has been applied in order to reduce the time delay in the arithmetic operations. Pixels are processed in one clock cycle, with a latency of 5 cycles, within data path shown in Fig. 5b. In Table 2, cycles needed to process the image, in hardware and software implementation are compared. Second column contains cycles needed for arithmetic operations, while third and fourth are for memory readings and writings. Last column shows total cycles required in each implementation. It could be seen that software implementation uses 27M memory while FPGA uses only 1M.

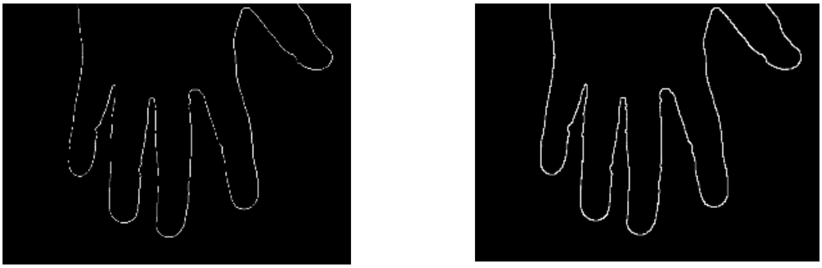


Fig. 6. Edges detected with Laplacian and Sobel filters in the hardware implementation.

Table 2. Cycles required for edge detection in FPGA and software implementation

Implementation	*# Arithmetic Operations*	*#Memory Readings*	*#Memory Writings*	*Total*
HW (FPGA)	1M	1M	1M	1M
SW (uP)	17M	9M	1M	27M

The accuracy obtained in the FPGA implementation is quite good. Currently, hardware provides results quite similar to software implementation. In Fig. 4 and Fig. 5 the superposition of both results are given for Laplacian and Sobel filters. As it could be seen, Sobel filter provides wider edges.

Biometric Feature Extraction

In this step the hand profile is read and measured in order to obtain the 25 features that characterize a human hand [1] and [2]. The architecture of the feature extraction module is shown in Fig. 7. This architecture is able to compute all features in parallel as the hand profile is being read from the memory. Thus, all features can be extracted within a single full memory read. Data path for arithmetic operations is pipelined in order to improve throughput.

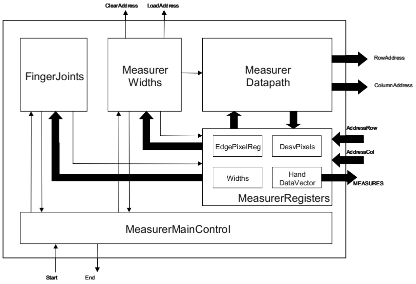


Fig. 7. Architecture of feature matching block

The architecture contains a data path (*FingerJoints)* that is in charge of reading the hand profile from memory and identifying some key points of the hand geometry, the finger joints which serve as reference points for other measures. *Measurer Widths* is a specific data path for obtaining finger widths *and Measurer Datapath* is in charge of calculating features that involve complex operations (angles, deviations, etc.). Finally, a set of register is used to store final measurements.

Execution process takes as much as M+ M/20 memory readings and the same for arithmetic operations. Table 3 presents a comparison with software implementation.

Table 3. Cycles required for edge detection in FPGA and software implementation

Implementation	*# Arithmetic Operations*	*#Memory Readings*	*#Memory Writings*	*Total*
FPGA	M/20+1M	M/20+1M	1	M/20 + 1M
uP	M/20+1M	25*1M	25	M/20 +26M

The accuracy has been checked and the results are quite similar. Differences are due to internal bit precision employed (16 bits in FPGA and 64 bits in *Matlab*®).

Feature Matching

The hardware implementation of this block is a comparator that decides if differences between user pattern and user template are less than a provided threshold (typical deviation). This block implements the Hamming distance detailed in section 2. Comparison is done in parallel, and only one cycle is required for this block. FAR and FRR for FPGA implementations is quite similar to those for software implementation, Fig. 8, therefore quality is maintained.

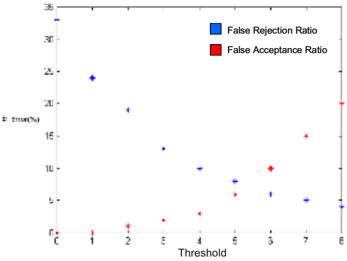


Fig. 8. FAR and FRR ratios in software implementation

4 Implementation Results

The Hand Geometry Authentication system implemented with a FPGA solution has been designed following *top-down* methodology. Hardware descriptions have been done in VHDL language at *RT level*. Synopsys®, ISE® and *Modelsim*® CAD tools have been used during the design process for *RTL* simulation, logic synthesis, *place&route* and FPGA mapping. Final prototyping has been done in a platform

FPGA from Xilinx (Virtex 2000E) included in a PCI board from Celoxica Ltd, [8] and [9]. This board provides also four banks of memory (2MB per bank) as well as PCI communication.
Comparison with respect to software application has been represented graphically, according to steps needed in the whole process. This information is shown in Fig. 9, for memory readings, memory writings and arithmetic operations.

Implementation report is shown in Table 4. There is a low FPGA occupation except for Block RAMs, used to store hand profile and to enhance memory accesses. It is possible to use smaller FPGAs storing this partial result in external memory also. With respect to maximum frequency, typical delay is 29.7 ns. Therefore this FPGA implementation works at 33 MHz.

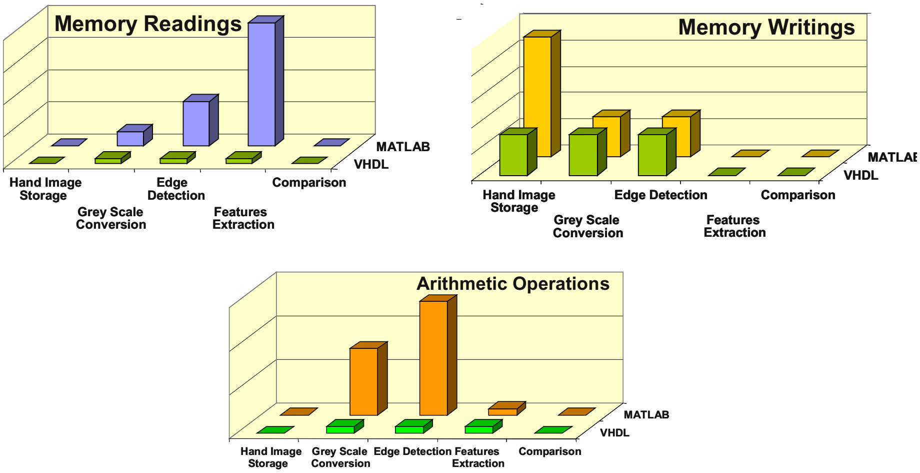


Fig. 9. Comparison of memory accesses and arithmetic operations in both implementations

Table 4.

FPGA Elements	*Used*	*Available*	*%*
Slices	1,578	19,200	8%
Slice Flip-flops	892	38,400	2%
4 input Lust	2,813	38,400	7%
Bounded IOs	263	408	64%
BRAMs	128	160	80%
GClks	1	4	25%

From tables 1, 2 and 3 the total number of cycles for algorithm execution has been obtained. FPGA implementation takes 1,551,360 operation cycles while software solution takes 22,133,760 cycles. Considering two scenarios, implementation in the selected FPGA and in a typical microprocessor (1Minst. per cycle) typical execution time will be 47ms and 20s respectively (taking into account that an instruction cycle in a microprocessor takes more than one clock cycle). On the other hand, if we consider the use of a high performance PC, times are in the same order as in FPGA but cost, resources and power consumption is much higher. Therefore, FPGA implementation has been proved as the best solution with respect to cost-effective and fast applications.

5 Conclusions

In this work, a hardware implementation for biometric recognition system based on hand geometry has been presented. Main aspects related to area, speed and accuracy have been considered. The proposed solution is able to reduce processing time by 3 orders of magnitude with respect to microprocessor based solutions, with a similar cost, size and power consumption, while keeping the same identification quality. On the other hand, it can be implemented in a small size FPGA, or an ASIC if product volume is large enough, thus making it suitable for a large number of low cost applications.

References

[1] Jain A. K., Bolle R., Pankanti S., et al., *Biometrics: Personal Identification in Networked Society*. Kluwer Academic Publishers. 1999.

[2] Jain L.C., Halici U., Hayashi I., Lee S. B., Tsutsui S., et al., *Intelligent Biometric Techniques in Fingerprint and Face Recognition.* CRC Press LLC. 1999.

[3] Schalkoff R.J. *Digital Image Processing and Computer Vision*. John Wiley & Sons. 1989

[4] Jain A. K., *Fundamentals of Digital Image Processing*. Prentice Hall. 1989.

[5] Jähne B., *Practical Handbook on Image Processing for Scientific Applications*. CRC Press LLC. 1997.

[6] Raúl Sánchez Reillo "*Mecanismos de autenticación biométrica mediante tarjeta inteligente*". Tesis doctoral. Universidad Politécnica de Madrid. Escuela Técnica Superior de Ingenieros de Telecomunicación. Madrid, 2000

[7] Sanchez-Reillo, R. Sanchez-Avila, C., Gonzalez-Marcos, A. "*Biometric Identification through hand geometry measurements*". IEEE. T. on Pattern Analysis and Matching Intelligence. Vol. 22, No 10. Oct. 200, pp. 1168-1171.

[8] Celoxica Ltd. www.celoxica.com

[9] Xilinx www.xilinx.com

SoftSONIC: A Customisable Modular Platform for Video Applications

Tero Rissa[1], Peter Y.K. Cheung[2], and Wayne Luk[1]

[1] Department of Computing, Imperial College London
[2] Department of Electrical and Electronic Engineering, Imperial College London
{tero.rissa,w.luk,p.cheung}@imperial.ac.uk

Abstract. This paper presents the Customisable Modular Platform (CMP) approach. The aim is to accelerate FPGA application development by raising the level of abstraction and facilitating design reuse. The solution is based on network of Nodes, communicating using packet-based protocol. The approach is illustrated using SoftSONIC, a CMP for video applications. Our approach promotes modularity and design reuse by having multiple interoperable layers of design abstraction, while supporting advanced development and verification methods such as mixed-abstraction execution and efficient system-level simulation based on Transaction Level Modelling. The platform provides domain-specific abstractions and customisations of various elements such as communication protocols and topology, enabling exploitation of data locality and fine- and coarse-grain parallelism. The benefits of our approach is demonstrated using SoftSONIC for development of several real-time HDTV video processing applications.

1 Introduction

Design cost has become a critical issue as a result of the exponential rise in silicon and design complexity. This paper presents the Customisable Modular Platform (CMP) approach, which aims to lower the design cost of complex digital systems. Reconfigurable platforms are utilised to alleviate some of the problems caused by *manufacturing* Non-Recurring Engineering (NRE) and *silicon* complexity, such as mask cost, probe card, signal integrity, power and clock management, manufacturing and process variability [1]. The novel aspects of the CMP approach aims to solve *system complexity* issues and those *silicon complexity* issues that are not directly solved by using reconfigurable platforms.

This area has attracted much attention during the past few years. Examples include platform-based system-level design [2,3], automated communication refinement [4] and network-on-chip paradigm [5]. The SystemC community has been the main contributor to the work on Transaction Level Modelling (TLM) [6], alongside with SpecC [7].

Our work differs from these previously proposed approaches in the following ways. (1) Exploitation of reconfigurable platforms: automated mixed simulation and hardware execution, rapid prototyping and run-time reconfigurability. (2)

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 54–63, 2004.

Domain specific design approach, which entails domain-specific information for design optimisation. (3) Platform customisation to minimise overhead. (4) Combining verification and reuse of Intellectual Property (IP) within a single framework. In addition, while previous work on TLM has been focused on bus-based embedded mircroprocessor systems. Our work offers a hardware oriented approach with a selection of communication topologies. Finally, CMP can be used for abstracting and generalising the Sonic-on-a-Chip approach [8].

The rest of the paper is organised as follows. Section 2 introduces the CMP approach. Section 3 presents SoftSONIC. Section 4 and 5 cover functional and design abstractions. Section 6 presents implementation results. Section 7 looks at current and future work, and concludes the paper.

2 Customisable Modular Platform Approach

Advanced multi-million gate FPGAs such as Xilinx Virtex II Pro and Altera Stratix and can significantly reduce the manufacturing NRE cost. However, the design NRE cost dominates the overall cost and time [9]. The CMP approach is developed to address this issue.

The CMP approach originates from the platform-based design concept [1,2, 10]. A CMP is a domain specific design methodology, capturing the underlying abstractions and design rules in the form of a virtual platform. It contains a set of design rules that characterise the target architecture. These rules are determined partly by the physical architecture and implementation technology, for instance hardware/software or FPGA/microprocessor. Many of the rules are 'virtual', enabling higher-level abstraction of architectural and physical issues. This higher-level abstraction accelerates the mapping of the application to the target architecture, and facilitates the design and verification process.

The CMP defines different *abstraction layers* for the functional abstractions. Each abstraction layer is defined in terms of information that they entail, for example numerical precision, timing behaviour, and resource usage. The CMP adopts four distinct abstractions: Algorithmic, Virtual TLM, Physical TLM, and Register Transfer Level (RTL).

The penalty in speed, area and power consumption is reduced by two factors. (1) The platforms are *application domain* specific. By narrowing the scope, platform abstractions can be chosen in accordance to known efficient implementations. Examples of such application domains are video processing [15], wireless networks [11] and communications [12]. (2) The platforms are *customisable* within the domain for a particular application and implementation, to support efficient architectural exploration and realisation. In general, architecture-level design decision have potentially much greater impact on the final performance than low level implementation optimisations.

Often, the first step in our approach is the selection of a domain specific CMP, characterised by the communication and computation. The specific scope of the chosen CMP enables efficient modelling and optimisation. Within SoftSONIC, our CMP for video applications, the customisable aspects include the level of coarse and fine grained parallelism, communication packet structures, communication protocols and topologies, packet data sequencers, and hardware/sofware partitioning. For example, the achievable parallelism can be dictated by the inherent parallelism of the algorithm or can be bound by communication bandwidth. The platform can be customised to meet the maximum performance in either case, for example with the selection of inter or intra device communication.

The main goal of a CMP is the separation of design concerns. In addition to the traditional separation of computation from communication and the functionality from architecture, the CMP approach provides the separation of verification of functionality from verification of performance. When applicable, this separation reduces the overall verification effort. The purpose is to avoid functional re-verification after design exploration and customisation for performance. High-level functional verification is also simplified, as implementation details need not be considered.

The CMP approach reduces the *observed complexity* to alleviate the overall design and verification effort in the following ways. **IP Reuse.** The platform concept enhances IP reusability by specifying an unambiguous interface between communication and computation. Furthermore, the modularity of the approach makes IP reuse possible in several levels and abstractions. **Restricted Design Space.** Application developers can focus on using the optimised facilities offered by a particular CMP, while CMP developers can focus on creating efficient CMPs by optimising and generalising designs. **High-Performance System Simulation.** The CMP approach uses TLM for the high-level block models. Simulation speed at transaction level can be 100 times faster than RTL simulation [13]. **Mixed-Abstraction Execution.** When a new block is developed, the behaviour of the rest of the system can be obtained by the fastest available model, reducing the simulation time. In addition, when using reconfigurable platforms, it is possible to use the actual hardware of available blocks in real-time execution. This can be several orders of magnitude faster than software simulation. **Co-Simulation.** Co-simulation between different design tools is difficult. The CMP approach alleviates the co-simulation problem by specifying a restricted, unambiguous platform communication mechanism. **Intrinsic Support for Run-Time Reconfiguration.** The possibility to update a product after deployment enables: (a) support for new features, (b) post-delivery design optimisation, and (c) adaptation to run-time conditions.

3 SoftSONIC CMP

SoftSONIC is a CMP for video processing applications. It is inspired by the UltraSONIC board-level architecture [14]. The main difference between SoftSONIC and UltraSONIC is that UltraSONIC was developed to support easy integration of hardware modules, whereas the aim of SoftSONIC is to reduce the observed design complexity of the system in order to speed up the design and verification.

In UltraSONIC the system is constructed from discrete PIPE hardware modules. In SoftSONIC these modules, called Nodes, are virtual, and are separated from an actual implementation. For example, one FPGA can contain several

Nodes, or one Node can span several FPGAs. UltraSONIC has a specific set of protocols to support a particular board architecture. SoftSONIC, in contrast, can be customised to target different board architectures and applications.

From the functional perspective, SoftSONIC is similar to UltraSONIC; both cover high-bandwidth video image processing applications [14,15]. SoftSONIC also adds or improves the following functionalities. **Multi-cycle pixel processing.** The processing of a pixel can take an arbitrary number of cycles. SoftSONIC supports **pixel-level parallelism** in three ways. (1) At high level, the number of Nodes that run in parallel can be varied. (2) At Node level, the data packets can be processed in parallel. In FPGA implementations of the Nodes, the packets are stored in block RAM, which supports parallel accessing. (3) At Node level, the clock frequency of each Node can be customised to meet design requirements. **Fork and join of data streams.** Forking of a stream is important, when new data are extracted from a stream while the original stream is needed later in the processing pipeline. The joining of streams is used, for example, in creating composite images, like blue/green-screen composition, logo insertion, and gradual stream transition. **Support for non-image data.** In addition to pure image data such as RGB 4:4:4[:4] and YUV (4:2:2, 4:2:0) in lines, windows and scattered pixels, SoftSONIC supports image-related information such as image metadata, compressed image data, and audio. **Shared memory random access.** Memory random access is enhanced with the use of memory-server Nodes and random-access data packets.

These improvements illustrate the additional functionalities that SoftSONIC supports. However, the main goal of SoftSONIC is to reduce application development time, which is achieved by exploiting the advances in Section 2.

4 Functional Abstraction

The purpose of functional abstraction is to specify the platform in a concise and easily adoptable manner. Functional abstractions include the model of communication and the model of computation. These functional abstractions separate the timing inside a Node from its external interfaces. This is an important factor in separating the communication from computation to obtain the benefits listed in Section 2.

Model of Communication. Communication in SoftSONIC is based on packets that contain a header and a payload. The packet payload contains *structured* data: in addition to explicit information, the type of packet implicitly indicates how the data should be interpreted. In SoftSONIC, basic packet types are pixel-based image data, structured to lines, windows or scattered pixel and address-data pairs for random memory access.

The information structure between Nodes is restricted to packets. The communication is unidirectional: once the packet is sent by the producing Node it cannot be altered, and after consumption the packet is deleted. Packet scheduling follows bounded First-In-First-Out (FIFO) buffers in a process network. Both attempts to read an empty buffer and to write to a full buffer are *blocking* operations.

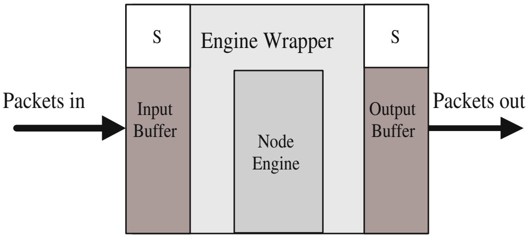


Fig. 1. A SoftSONIC node

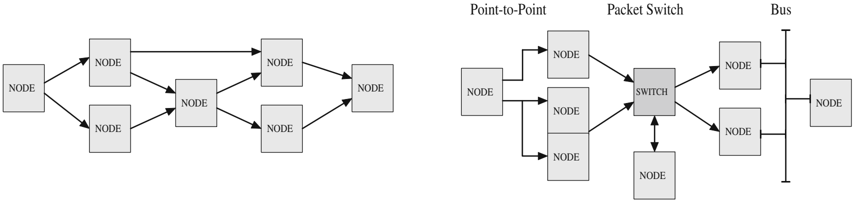
One exception to packet communication is Configuration Register (CReg) communication. An element outside the communication network, for example the host computer or on-board user interface, can access CRegs inside the Nodes without using packet-based communication. The CReg interface is intended for low-bandwidth, time-independent sporadic communication involving, for example, information about Node parameters and user control. The CReg interface implementation is application and environment specific, but must follow CReg update rules, which due to space restrictions are not explained here.

Model of Computation. User-defined functions take place inside the Nodes. The general structure of a SoftSONIC Node is depicted in Fig. 1. A Node consists of: the Input and Output Buffers, the Buffer status information 'S-box', the Node Engine Wrapper, and the Node Engine.

The Input and Output Buffers contain FIFOs for packet communication. The S-box contains the status information of the buffer and handles the arbitration of the buffer. The S-box interface handles the transition between abstractions layers, development environments, and clock domains. The Node Engine Wrapper is a customisable data sequencer for input and output buffers. It can be customised, for example, to access packets in parallel sequential streams or to have application specific scan pattern. It starts consuming the input packet when all required input buffer(s), output buffer(s) and the Node Engine(s) are available. The CReg interface is implemented within the Node Engine Wrapper and the actual registers locate inside the Node Engine.

User-defined computation is performed inside the Node Engine. The Node Engine can use only the information included in the input buffer(s), the packet header, internally stored data and the CReg data. A Node can contain one or more engines wrapped by a single Node Engine Wrapper.

The Nodes are connected to each others with channels. The channels have an identical interface to the input and output buffers, and have the same firing rules as computation Nodes. The only difference is that a channel does not change the contents of the packet in any way; it just transports a packet from one buffer to another. It is also possible to connect Nodes together in a point-to-point manner. In this case one Node's output buffer becomes another's input buffer.



(a) Virtual TLM

(b) Physical TLM

Fig. 2. Transaction Level Model Layers: Virtual and Physical

5 Design Abstraction Layers

SoftSONIC is divided into four distinct design abstraction layers. Each layer is defined in terms of timing, implementation and data accuracy, the incoming and resulting system model and the target design questions the layer aims to answer.

Algorithmic: Executable Specification. The algorithmic layer does not use SoftSONIC functional abstractions. The purpose of this layer is to provide an executable specification for the system under development. In this layer, the whole system functionality is described using high level software languages such as C/C++ or Matlab. The executable specification does not contain information about the timing and performance of the system. However, this layer must contain information about limited bitwidth effects and the maximum error bound.

Virtual TLM: Parallel Functionality. Layer 2, Virtual TLM, is the first SoftSONIC specific layer. The purpose of this layer is to re-specify the contents of the Algorithmic layer using the SoftSONIC functional abstraction. The main design effort is in mapping the functionality to individual Nodes and extraction of parallelism in the application. Node communication is modelled using unlimited communication resources in bandwidth and the number of connections, but with bounded input and output buffers. In other words, the packet always takes zero time to arrive from producer to the consumer. Bounded input and output buffers enable mixed-abstraction execution. The bounding of the FIFOs can be made arbitrarily at this point, as it does not affect the functionality of the system. An example of a Virtual TLM is depicted in Fig. 2(a).

In this layer, sending and receiving a packet is modelled as a single transaction in a queueing process network. Coarse grain parallelism at Node level can be explored, but there is limited feedback of resource usage as unbounded resources are assumed. In other words, it is possible to explore whether the algorithm can be parallelised, but it is impossible to determine whether such a system is feasible for real-life implementation. The verification task is system-to-system: the functionality of the Algorithmic layer must correspond to that of Virtual

TLM. In this layer, both communication channels and computation Nodes are described using untimed TLM models.

Physical TLM: Design Exploration. In the Physical TLM layer, the functional description of the Nodes remains unchanged. The focus is on resource allocation, design exploration, and customisation. As the functionality does not change, the verification task of this layer involves purely the *validation of the performance requirements* in speed, area, power, etc. The separation of functional and performance verification simplifies the task, and makes it easy to support a systematic approach to the problem. This can guarantee 'correct by construction' functionality when carrying out architectural design exploration.

Design exploration can be divided into two parts: communication and Node implementation analysis. Communication exploration examines the choice of different communication protocols, media and throughput versus resource usage. Choices can be made between on-chip and off-chip resources, packet sizes and types. A crucial decision is the size of the input and output buffers, as these can have significant impact on performance and resource usage. In Node implementation analysis, different levels of coarse and fine grain parallelism are explored. Also, if the decision has not been made earlier, Node-level software/hardware partitioning is carried out.

In this layer, the final packet sizes and types, communication and implementation media are allocated and buffer sizes re-examined. Thus performance analysis in terms of throughput can be estimated by modelling the individual throughput of the Nodes and communication system. These figures can be obtained from the lower RTL layer, or estimated. The communication is described using timed TLM models. As mentioned before, the computation Nodes remain unchanged; they are still modelled with untimed TLM models.

From the Virtual TLM layer description in Fig. 2(a), one possible outcome of design exploration and customisation is illustrated in Fig. 2(b). The communication of the first two Nodes is handled by point-to-point communication, such that the output buffer of the producing Node is the input buffer of the consuming Node. One of the Nodes is replicated to enhance Node-level parallelism. The second communication choice is to utilise a packet switch for the communication and finally, a bus is utilised.

RTL: Path to Implementation. The RTL layer implementation begins by generating the communication description from the Physical TLM layer. The communication acts as a wrapper for the RTL implementation. For simulation, the higher-level description of the communication can be used. This layer is focused on to the computation in the Nodes and the communication remains unchanged. The RTL description of the Node engines can be done by using a synthesisable language like RTL SystemC, HandelC, VHDL or Verilog. The verification of the Nodes is made easier by the independence of the Nodes and the ability to re-use the verification environment. The verification task is Node-to-Node when we move from Physical TLM layer to RTL layer. This is in contrast to system-to-system verification, which is the case when we move from Algorithmic layer to Virtual TLM layer. This also leads to opportunity for the use of advanced verification methods like assertion-based verification and formal

Table 1. Comparison of performance in Thermal Camouflage effect

Implementation	Clock Seed [MHz]	Throughput [Frames/s]	Speed-Up Factor (compared to SW)
Software (PC)	2400	1.6	1.00
VHDL simulation (PC)	2400	0.0006	0.0004
Mixed VHDL and SoftSONIC	2400/133	0.02	0.01
SoftSONIC	133	64	37.85
Parallel SoftSONIC	133	128	75.70
UltraSONIC	66	32	18.88
Normalised SoftSONIC	66	32	18.78

methods. The reduced scope of the verification problem increases the feasibility of these methods.

6 Results

This section compares the performance of various implementations of SoftSONIC against UltraSONIC and software simulations. We have developed an experimental hardware implementation which involves a real-time HDTV 1920 by 1080 RGB 4:4:4 10 bit/channel (SMTPE 372M – "Super2K") video effect application. The system creates a 'thermal camouflage' effect used to visualise semi-transparent objects. Similar effect has been used, for example, in the movies Predator (1987) and Hollow Man (2000). This application consists of 6 Nodes: Packet Source and Sink, 3x3 Blur Filter, 2D 3x3 Sobel Filter, Image Differentiator, and finally the 'Lens Effect' Node that produces a special effect to areas where the foreground image is different from the background image. The lens effect is created by varying the refraction according to the intensity of the edges.

One implementation of SoftSONIC is customised to the application and to the Xilinx Virtex-II Pro 50 FPGA by selecting packets of one line with buffer size of 2. This way, eight Block RAMs in 512x36 bit mode can form one buffer. With this setup, it is possible to have eight parallel pixel reads and writes. In this application, maximum throughput is determined by external memory access. Buffer size of 2 offers optimal performance/area tradeoff, as one buffer can be read while the other is being written, without significant pauses in processing. As all the functions in the application involve stream processing without non-deterministic components, point-to-point communication is the optimal selection for communication. Sometimes, the performance could be further optimised by maximising the individual clock speed of the Nodes, but in this case the maximum clock speed of 133MHz is dictated by the ZBT SRAM interface. Clock rates could also have an effect on the power consumption, but this is not currently being evaluated.

We consider seven different implementations, and the performance results are summarised in Table 1. The software implementation is the C++ Algorithmic level description of the system as a DirectShow filter running on a dual Athlon PC at 2.4GHz with 2G bytes of memory. Software simulation is based on Mod-

Table 2. SoftSONIC Node performances

Kernel	BRam	1 engine			2 engines			4 engines			8 engines		
		Slice	MHz	fps	Slice	MHz	fps	Slice	MHz	fps	Slice	MHz	fps
Invert Colours	8	160	289	139	183	295	282	228	315	604	405	296	1137
Image Diff	16	264	277	132	313	270	259	445	279	535	809	267	1025
Alfa Blend	16	364	175	83	554	167	160	951	160	307	1868	146	560
3x3 Noise Filter	24	1162	201	95	1773	202	191	3056	200	380	5525	220	836
2D 3x3 Sobel	24	1578	200	94	2736	202	191	4999	185	351	9389	140	532

elSim SE Plus 5.7f running on the above PC. In the mixed VHDL and SoftSONIC implementation, only the Lens Effect Node is simulated, and the other parts of the design is running on the FPGA. The Lens Effect Node would most likely be the only one that needs to be built from scratch, as the other Nodes are common image processing kernels. In the Parallel SoftSONIC implementation, the Packet Source and Sink produce/consume packets in double rate, and there are two Node Engines in each processing Node to enhance parallelism. The application does not have inter-line dependencies, so the upper limit of the parallelisation is bound only by the available memory or memory bandwidth. The UltraSONIC implementation contains Xilinx Virtex 1000E FPGAs using UltraSONIC protocols. Finally, in order to enable fair comparison between SoftSONIC and UltraSONIC, the Normalised SoftSONIC is an implementation on UltraSONIC hardware.

From the results it can be seen that the SoftSONIC implementations provide significant speedup compared to the software implementation. The Normalised SoftSONIC implementation is only slightly less efficient than the UltraSONIC implementation, indicating that the performance penalty is not significant. The main overhead is the higher usage of Block RAMs for input and output buffers.

Table 2 illustrates individual SoftSONIC Node performances, without concerning external memory access. The results are obtained after place and route, but without I/O buffers, as reported by Xilinx ISE 6.2.01i. For synthesis, Synplify Pro 7.2.2 is used. For each kernel, there are four separate implementations, which indicate 1, 2, 4, and 8 parallel engines inside a Node. Naturally, the faster implementations consuming more area, has more parallelism. All the implementations are automatically generated according to the level of engine parallelism. Because of the used abstractions, the engines can simply be replicated in order to accommodate the parallelism. However, as all these implementations use before mentioned 8 BRAM I/O buffers size, the data sequencers are different in each implementation in order to handle serialisation and de-serialisation of the data. Although, in the case of window processing this is non-trivial task as parallel engines have overlapping data, also these are automatically generated.

The table shows that very high frame rates can be achieved by using the SoftSONIC Nodes, and it is likely that the performance limitation will come from external memory access. In addition high performance of the kernels, the integration of kernel-Nodes to applications is greatly facilitated, as the Node interface guarantees interoperability.

7 Summary

We have described the Customisable Modular Platform approach for rapid application development and optimisation of reconfigurable designs. Our approach is illustrated using the SoftSONIC platform. Opportunities for customising SoftSONIC are discussed, and it is shown that SoftSONIC can produce many implementations rapidly without significant overheads. Current and future work includes refining the SoftSONIC model, automating the customisation process, and evaluating our approach using complex applications.

References

1. Lysaght, P.: FPGAs as meta-platforms for embedded systems. In: Proc. IEEE Int. Conf. on Field-Programmable Technology. (2002)
2. Keutzer, K., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A.: System-level design: orthogonalization of concerns and platform-based design. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems **19** (2000)
3. Ferrari, A., Sangiovanni-Vincentelli, A.: System design: traditional concepts and new paradigms. In: Proc. Int. Conf. on Computer Design: VLSI in Computer and Processors. (1999)
4. Abdi, S., Shin, D., Gajski, D.D.: Automatic communication refinement for system level design. In: Proc. Design Automation Conference. (2003)
5. Benini, L., Micheli, G.D.: Networks on chips: A new SoC paradigm. IEEE Computer (2002)
6. Grötker, T., Liao, S., Marting, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers (2002)
7. Gajski, D.D., Zhu, J., Domer, R., Gerstlauer, A., Zhao, S.: SpecC: Specification Language and Methodology. Kluwer Academic Publishers (2000)
8. Sedcole, P., Cheung, P.Y.K., Constantinides, G.A., Luk, W.: A reconfigurable platform for real-time embedded video image processing. In: Proc. Int. Conf. on Field-Programmable Logic and Applications. (2003)
9. Semiconductor Industry Association: International Technology Roadmap for Semiconductors. (1999, 2001, 2003)
10. Lysaght, P.: Future design tools for platform FPGAs. In: Proc. Symp. on Integrated Circuits and Systems Design. (2003)
11. Tuan, T., Li, S., Rabaey, J.: Reconfigurable platform design for wireless protocol processors. In: International Conf. on Acoustics, Speech, and Signal Processing. (2001)
12. Spivey, G., Bhattacharyya, S.S., Nakajima, K.: A component architecture for FPGA-based, DSP system design. In: IEEE Int. Conf. on Application-specific Systems, Architectures and Processors. (2002)
13. Kogel, T. et. al.,: Virtual architecture mapping: A systemc based methodology for architectural exploration of system-on-chip designs. In: Proc. Int. Conf. on Systems, Architectures, Modeling, and Simulation. (2003)
14. Haynes, S.D., Epsom, H.G., Cooper, R.J., McAlpine, P.L.: UltraSONIC: A reconfigurable architecture for video image processing. In: Proc. Int. Conf. on Field-Programmable Logic and Applications. (2002)
15. Haynes, S.D., Stone, J., Cheung, P.Y.K., Luk, W.: Video image processing with the SONIC architecture. IEEE Computer **33** (2000)

Deploying Hardware Platforms for SoC Validation: An Industrial Case Study

A. Bigot, F. Charpentier, H. Krupnova, and I. Sans

CMG/FMVG, ST Microelectronics, 12 rue Jules Horowitz – B.P.217 F-38019
Grenoble Cedex, France
{Anne.Bigot,Fabrice.Charpentier,Helena.Krupnova,Isabelle.Sans}@st.com

Abstract. The high complexity of the modern SoC designs ([17]) raises the serious verification challenges. The design verification becomes even more critical because of the constantly shrinking project timescales due to the time to market pressure. The hardware platform based verification is the only one that can cope with the increasing SoC complexity: only in hardware, complex test sequences exercising the complete design can run at a reasonable speed. This paper presents how both emulation and rapid FPGA-based prototyping technologies are deployed in a complementary way in a real industrial environment. Taking two latest highly complex SoC projects as an illustration (3 and 4 million ASIC gates without counting memories), we will describe the integral hardware platform based validation approach. The deployed methodology resulted in a success story for both emulation and rapid prototyping projects for both SoCs.

1 Introduction

Producing the first silicon is an important milestone for the SoC project. Getting the working first silicon allows the semiconductor company to gain the customers confidence and largely contributes to fitting the market window. Two conditions are necessary to obtain the working and “life" first silicon: (1) all vital hardware parts are functional; (2) the application and driver software is ready at the moment the chip is back from the fabrication. To reach the first condition, it is mandatory to exercise the complete chip functionality before the tape-out. To meet the second condition, the software teams should start the software development a long time before the silicon is back. The solution to satisfy both conditions is the concurrent hardware-software engineering.

The hardware-based emulation ([2], [3], [10], [16]) is the only one technology that allows to run the complex test sequences on a complete SoC. It insures a $10^5 - 10^6$ speedup compared to simulation. The simulation capabilities are limited to testing subsystems and separate IP blocks. To run the tests approaching the real life functionality or to exercise the in-circuit operation of the SoC, it is mandatory to use the emulation. Once the design was emulated, the design team gets sufficient confidence in hardware functionality. Very often, the corner case operation may be detected not by using an artificial test sequences but when trying to develop and to run the real software drivers. As a consequence, the software development should start very early.

Big emulation machines ([10]) can support the SoC software development as well as hardware debug. The disadvantage for software engineers is a relatively low speed

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 64–73, 2004.

of an emulator (100K to 1Mhz) and its extremely high cost limiting the duration of the access time. On the opposite, the rapid FPGA-based silicon prototypes are much more convenient for software debug, working 10 to 100 times faster and being less expensive ([4], [9], [14], [15]). The only problem is that the mapping process is more complicated and requires more time. To be mapped on an FPGA platform, the design hardware should be debugged and stable to reduce the number of iterations to be done for FPGA mapping. Emulation is thus an important pre-requisite to insure the FPGA-based silicon prototyping success.

Deploying both hardware-based emulation and FPGA-based rapid prototyping technologies in parallel allows to meet the hardware verification as well as software development objectives. It constitutes the basis for concurrent hardware-software engineering. This paper describes how this deployment was done using an example of two latest highly complex SoCs. These SoC designs will be called *DVD_project* and *STB_project*. Both projects extensively used emulation to debug hardware prior to tape-out and an FPGA-based silicon prototype was created for both projects to support the software development teams.

Although both emulation and rapid prototyping technologies exist for a number of years ([1], [2], [3], [12], [13]), their industrial application remains challenging and requires the breakthrough in the design teams and software teams way of thinking. The difficulties encountered during the mapping on emulator or an FPGA platform often require an additional support from the design teams, which are already heavily loaded with the design activities. From the other side, the market reality and constantly reducing project timescales push the design teams to apply the hardware-based verification technologies. The given paper describes the methodology, allowing taking the best of both emulation and rapid prototyping technologies and making them profitable to the design teams.

The paper is organized as follows. Section 2 describes the hardware-based verification cycle embedded inside the project development cycle. Section 3 presents the mapping flow for both emulation and rapid prototyping platforms putting an accent on the most critical points. Section 4 presents the statistical data about the emulation and rapid prototyping activities for *DVD_Project* and *STB_Project* followed by the Conclusions.

2 The Hardware-Based Verification Cycle

The latest SoC designs are highly complex ([17]). They contain one or more cores, ten to twenty main IP blocks, several SRAM, SDRAM or DDR memory interfaces, a number of interfaces to peripherals. The project development starts from block-by-block validation of the selected IP blocks. A big number of IPs are reused. A large number of blocks are re-designed or re-customized. The initial block validation is done in simulation or co-simulation with hardware. In parallel to the block validation, once the SoC architecture is determined, the design team becomes able to produce the first assembled backbone. This backbone usually contains a processor core, the system bus and the memory interfaces.

The emulation and the rapid prototyping activities usually follow a number of iterations. The iteration corresponds to a fixed step in the project development reflecting the degree of project maturity. The first iteration is usually done with the first assembled backbone. The emulation and the rapid prototyping projects are done in a timeframe that corresponds to a period between the creation of the first backbone and the coming back from production of the first silicon. Today, the approximate duration of this period is around six months. The hardware-based verification technologies should be extremely efficient to allow several productive iterations during this time frame. The hardware-based verification technologies require a high competence and a constant follow-up of the technology progress. That is why, hardware-based verification is done by a different team specialized in hardware emulation and prototyping. The emulation team offers the platform mapping to the design team and then both teams collaborate to debug the design.

The emulation activity starts when the design team delivers the first backbone to the emulation team (Figure 1). This backbone is mapped on the emulator to run a number of system-level tests. Numerous hardware bugs are usually cleaned out during this step. The hardware emulators offer an excellent visibility for hardware debug. They allow seeing all the design nodes with a comfortable depth and fast re-compile. On the contrary, the visibility on the commercial FPGA-based rapid prototyping platform is limited. These platforms are not suitable for hardware debug in the case of big SoC designs. The design that is supposed to be ported on an FPGA platform should be stable and clean. The emulation insures that the hardware is operational, the RTL is synthesizeable and clean. Mapping to emulator is an important condition for the silicon prototyping success.

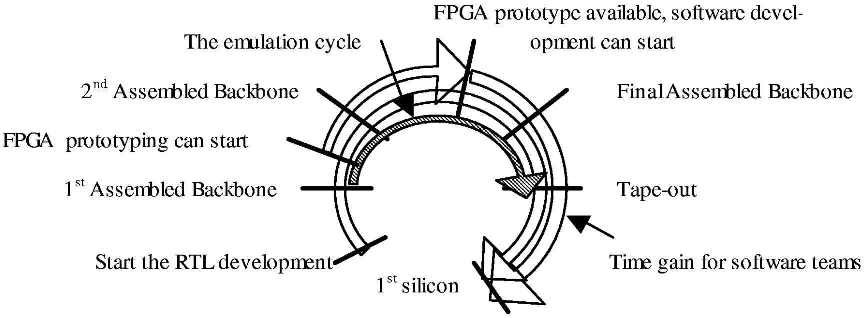

FPGA prototype available, software devel-opment can start
The emulation cycle
2nd Assembled Backbone
Final Assembled Backbone
FPGA prototyping can start
1st Assembled Backbone
Tape-out
Start the RTL development
Time gain for software teams
1st silicon

Fig. 1. Hardware-based verification life cycle

The hardware-based verification flow is depicted in Figure 2. Once the first backbone is cleaned in emulation, it can be delivered to the FPGA prototyping team to map it on the FPGA platform. This first rapid prototyping iteration is rarely directly usable for the SoC software teams. Its purpose is basically the validation of the FPGA mapping. This first iteration is followed by a number of subsequent iterations where each time new IP blocks are added to the backbone. The last, the pre-tape-out assembly has to contain the complete design. The emulation technology allows validation of a number of in-circuit interfaces, which gives to the design team a high confidence that the first silicon will respond to the external stimuli. The final FPGA-based silicon prototype should be ready early enough before the first silicon is back to give to the software teams enough time for validating the software.

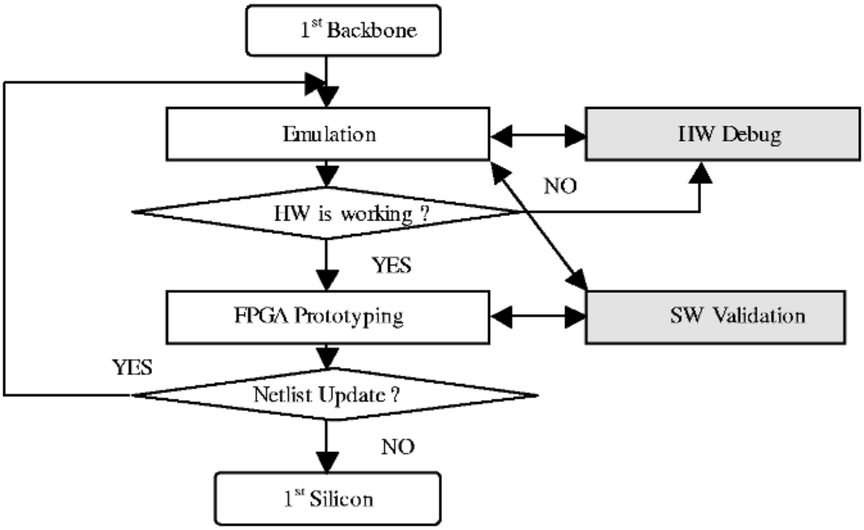


Fig. 2. HW and SW debug using hardware platforms

In practice, the software development teams first start using the emulation platform for software development. Then, they switch to the FPGA-based platform once it is working. The earlier they start using the FPGA platform, the earlier they can benefit from its higher speed and free the emulators for debugging the other projects or even let the design team to continue debug. The next section describes the mapping flow used for both the emulation platform and the FPGA platform.

3 Mapping Flow

The first prerequisite for HW emulation or prototyping mapping is to have a synthesizable RTL. The chip is then delivered in gate format to emulation team. The mapping flow is depicted in Figure 3. Among the existing commercial emulation solutions the company choice was oriented to the Mentor Graphics Celaro emulator ([10]). The Celaro emulation technology is based on custom FPGA developed explicitly for emulation. Aptix System Explorer MP4CF system ([5]) was selected as a rapid prototyping platform, targeted to the SoC software development. The targeted FPGA technology is Xilinx Virtex 2 ([6], [7], [11]).

Mapping on an emulator requires modeling the ASIC memories using the memory resources available on the emulator. In addition, the ASIC designs often directly instantiate the technology cells. These cells should be modeled to map on the emulator. The major difficulty to get the working design on the emulator is the clock domain resynchronization. Often, it requires several compile iterations. The emulation technology is developed to support the designs which are not yet mature and offers numerous facilities: clock injection at any point inside the hierarchy, description of clock domains using simple input files, easy pull-up or pull-down of the internal signals, easy fix of

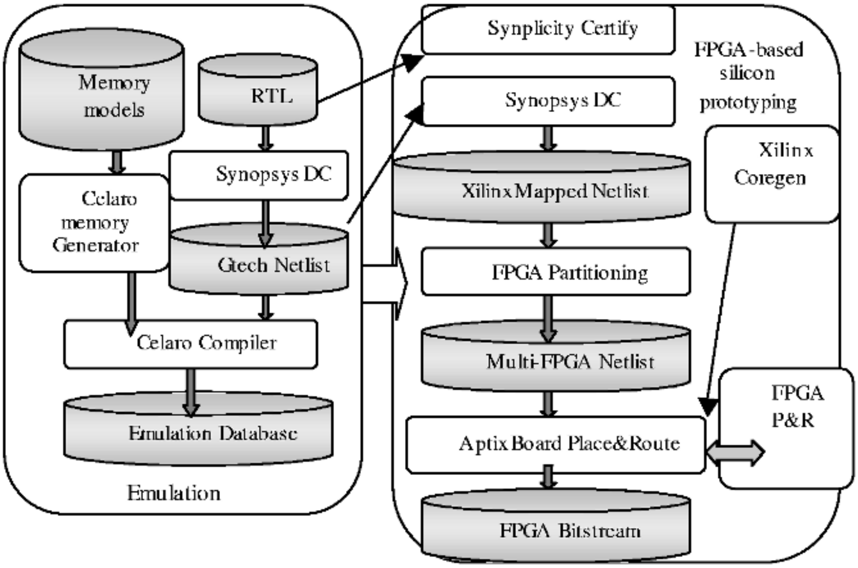


Fig. 3. The emulation and rapid prototyping flow

the internal signals to constants, etc. Exploiting all these facilities makes mapping on an emulator a matter of days. Once the emulation platform is operational, the design can be replicated on the FPGA platform.

Mapping on an FPGA platform requires much more effort. The final objective is better performance and low cost. These advantages justify the considerable mapping effort. Compared to mapping on an emulator, the design has to be partitioned by the user when mapping on an FPGA-based platform. If the design has a lot of connectivity, the partitioning step has to be followed by the pin multiplexing. The major difficulty when mapping on commercial FPGAs is clocking. Very often, it happens that the design works on the emulator but does not work when mapped on FPGA. The reasons are the gated clocks and the different non-FPGA friendly logic structures. Big emulation machines automatically transform the logic when they encounter gated clocks or other non-mappable structures. This transformation is transparent to the user. On the contrary, when mapping on commercial FPGAs, the user has to analyze the clock trees and detect all the structures that are not clean. The clock problem cleaning is very poorly supported by the FPGA mapping tools. Few tools support the gated clock transformation only for certain types of gated clocks. In any case, the user has to understand the clocking in order to guide the de-gating process and finally to clean the remaining cases manually. The latest SoCs have extremely complex clocking: twenty or more main clocks, digital clock generators with programmable clock frequencies that may change during the operation, hundreds of gated clocks, clock division, etc. In addition, clock signals may be used for example as selection of multiplexers or enable of tri-states. All these structures must be cleaned to get the working FPGA implementation. Very often, detecting these cases happens when debugging the not working design mapped on the FPGA platform.

The emulation platform is a very efficient support to debug the FPGA prototype. It is much closer to the FPGA prototype than simulation platform (test bench, memories, clocks) and is used as a reference during the debugging. In addition, it allows debugging the in-circuit functionality with external devices that are not possible to be represented as a simulatable model. Both emulation platform and FPGA platform use the common synthesizeable test bench. The FPGA platform requires slight modifications of the test bench for the clock and reset logic. Mapping on FPGA platform takes as a starting point the input files for the emulator: these files define the clock domains and specify the different fixes done during the debugging on emulator. These fixes often correspond to applying constants on the selected signals, applying the pull-ups and pull-downs and injecting clocks.

To complete the commercial tool set for emulation and rapid prototyping, the hardware prototyping team developed a number of internal tools. As an example may be cited the “emulation rule checker", the memory generators, the emulator booking system, the gated clock cleaning scripts for FPGAs, etc.

4 Project Data

The described above methodology was successfully applied to two recent SoC projects. Both projects are the consumer applications. The *DVD_project* is a DVD circuit of approximately 3 millions ASIC gates without counting memories. The *STB_project* is an STB circuit of about 4 millions ASIC gates without counting memories. The practical project data is summarized in the Table 1.

Table 1. Practical project data

	DVD_project	STB_project
Size in ASIC gates (no memory)	3 millions	4 millions
Embedded cores	ST20C104, ST220	ST20C201,MMDSP
Big Memories	4Mx16 DDR	8Mx16 DDR
Used Celaro configuration	96s	96s
Number of Celaro clock domains	2	2
Number of emulation iterations prior to tape-out	20	More than 40
Number of the rapid prototyping iterations	3	2
FPGA technology	Xilinx Virtex2	Xilinx Virtex2
Number of FPGAs	7 XC2V6000	7 XC2V6000 + 2 XC2V8000
Average FPGA filling	84%	82%
FPGA platform speed-up versus emulator	30x	20x
FPGA platform pin multiplexing ratio	4x1	4x1
Number of the emulation machines deployed	3	3
Number of the FPGA prototyping boards deployed	1	3

Both projects use a DDR memory model implemented with a wrapper around an SRAM memory. Despite the extremely complex process of development and validation, the model is compliant with Jedec DDR standard. The Celaro emulator configuration was in both cases a 96 slot machine. Both projects require an external interface for the JTAG connection between SW debugger and a processor core in order to develop and test the application software. One of the in-circuit interfaces for the *DVD_project* is represented in Figure 4. It shows the multi-core *TAPMUX* allowing reducing the number of external JTAG connections to talk to all internal cores. The functional diagram of this external interface is presented in Figure 5. It shows the multiplexed access to multiple cores inside the SoC through only one external connection. Debugging this interface on the emulator was one of the critical pre-tapeout activities. Insuring that both cores are reachable through the JTAG interface was mandatory to get the life silicon.

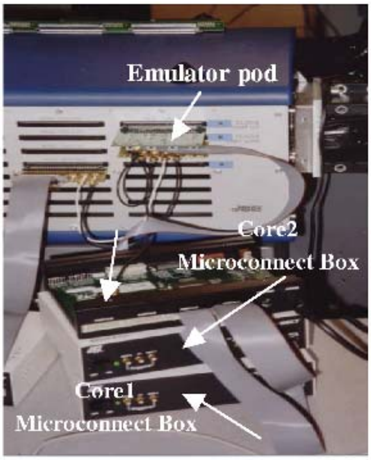


Fig. 4. In-circuit interface for multi-core TAPMUX on the emulator (DVD project)

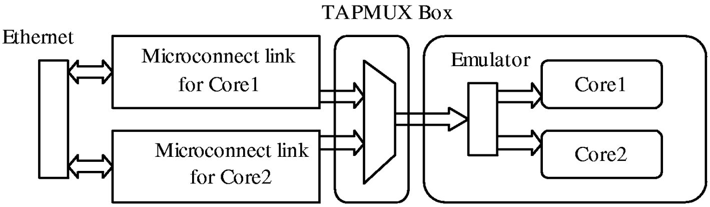


Fig. 5. TAPMUX functional diagram (DVD project)

For both projects, the replication on the FPGA platform started after the design was operational on the emulator. The FPGA implementation for the *STB_project* required 9 FPGA modules. Such complexity requires significant FPGA synthesis and place&route time. However, the major part of the mapping time was dedicated to overcoming different kind of problems. Although the mapping on the FPGA platform was a complex process, the obtained speed-up (20 and 30 times) was extremely interesting for the software development teams. While getting the test tool prompt for the *DVD_project* on an emulator required 30 minutes, on FPGA platform it required only 1 minute. The speed advantage is even more important because the software engineers often work interactively using software debuggers. Due to the overlapping schedules, the Aptix MP4CF board had a common configuration for both projects. The board contained in total 10 FPGAs. The *DVD_project* was using 7 of them. The *STB_project* was using 9 FPGAs. Both projects used 6 FPGAs in common. The platform was used in a time-shared mode for both projects. The FPGA board view for both projects is represented in Figure 6. Depending on the project that booked the FPGA platform, the FPGA board was reconfigured for this project. The software teams used the micro-connect link to access the design inside the hardware platform.

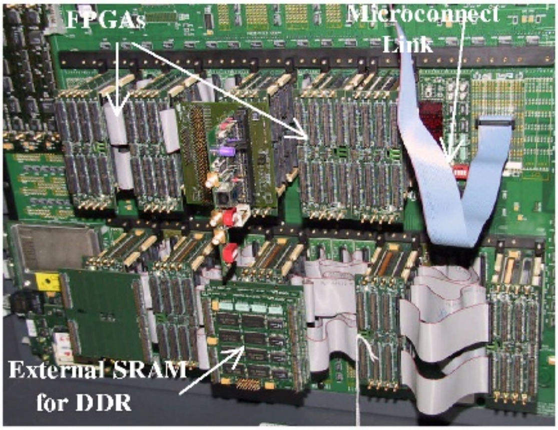


Fig. 6. FPGA board view for STB_Project and DVD_Project

The number of the emulation iterations versus the number of the rapid prototyping iterations presented in Table 1 corresponds to the difference in the mapping process productivity. Due to the longer mapping times for the FPGA platform, only 2 or 3 essential design versions were ported to the FPGAs. While the first mappings for both emulation and rapid prototyping platforms took time, the last incremental mapping on the emulator for the *DVD_Project* was done in only 1 day. The latest incremental mapping on an FPGA platform took 1 week. The comparison between the prototype speed and the platform setup time for both rapid prototyping and emulation is presented in Figure 7. The depicted numbers correspond to the results of the *DVD_Project* and *STB_Project*. Mapping on emulator requires from few days to one or two weeks and the prototype works at about 100KHz. Mapping on the FPGA platform takes several weeks and the prototype works at a speed of 1-10MHz.

The hardware verification platforms were deployed on one of the company sites. The hardware and software validation teams for both projects were geographically spread to more than five sites. All the emulation and rapid prototyping technology users accessed the hardware platforms remotely. For software debug, as shown in Figure 8, a micro-connect box was connected to each hardware platform. The software engineers used an IP address to establish the connection with the micro-connect box. The geographical time shifting contributed to increase the hardware platforms utilization ratio.

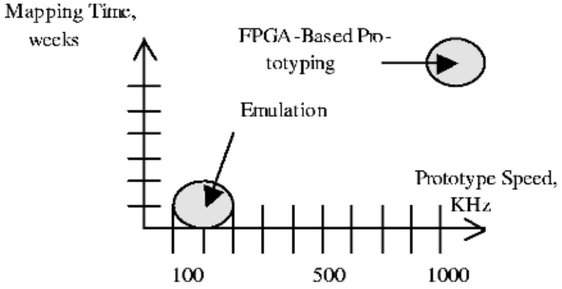


Fig. 7. Emulation versus FPGA-based prototyping

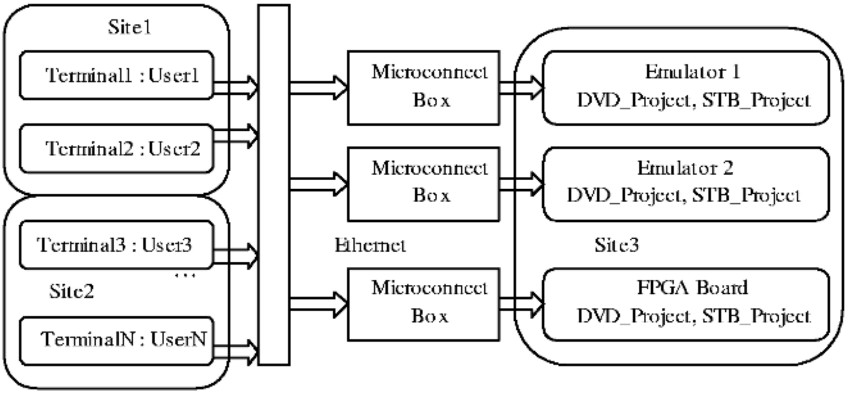


Fig. 8. Hardware platform utilization: use model

5 Conclusions

The paper presented a successful application of the emulation and rapid FPGA-based prototyping technologies to two recent SoC projects. The emulation technology is vital for the SoC hardware debug, while the rapid FPGA-based prototyping technology supports the SoC software development. The amount of practical project datas presented in the paper shows how both hardware prototyping technologies were deployed in a complementary fashion. The deployed methodology offered the SoCs hardware engineers a fast emulator porting time and good internal signal visibility. Software engineers got an opportunity to benefit from the fast speed of the commercial FPGA based platforms. Porting the design to FPGA-based hardware platform was preceded by emulation, thus guaranteeing the correct hardware functionality and faster mapping times. From the industrial viewpoint, the ideal future hardware verification platform would be one that combines the high speed of the modern FPGA technology with the fast mapping time and visibility of the big emulation machines.

References

1. S. Hauck: The Roles of FPGAs in Reprogrammable Systems. Proc. Of the IEEE, Vol. 86, No. 4 (1998): 615-639.
2. J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, A. Agrawal: Logic Emulation with Virtual Wires, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 16/6 (1997): 609-626.
3. R. Tessier, J. Babb, M. Dahl, S.Z. Hanono, A. Agrawal: The Virtual Wires Emulation System: A Gate-Efficient ASIC Prototyping Environment, Proc. ACM/SIGDA Int. Symp. On Field Programmable Gate Arrays (1994).
4. Aptix Home Page, http://www.aptix.com
5. System Explorer MP4 Reference Guide, Aptix, 1999
6. Aptix Product Datasheet: Xilinx Virtex-II FPGA Module, May 2002
7. http://www.xilinx.com
8. http://www.synplicity.com/products/certify/index.html
9. M. Pavesi: Modern FPGA capabilities made available to the FlexBench modular rapid prototyping platform. Proc. DATE 2002.
10. Mentor Graphics Accelerated Verification/Emulation page, http://www.mentor.com/celaro
11. Xilinx, “Virtex-II Platform FPGA Handbook", December 2000
12. V. Betz, J. Rose, A. Marquardt, Architecture and CAD for Deep-submicron FPGAs, Kluwer Academic Publisher, 1999
13. S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, Field-Programmable Gate Arrays, Kluwer Academic Publisher, 1992
14. V. Bhatia, S. Shtil: Rapid Prototyping Technology Accelerates Software Development of Complex Network Systems, Proc. RSP (1998): 113-115
15. M. Courtoy: Rapid System Prototyping for Real-Time Design Validation, Proc. RSP (1998): 108-112.
16. K. Harbich, J. Stohmann, E. Barke, L. Schwoerer, A Case Study: Logic Emulation – Pitfalls and Solutions, Proc. RSP (1999): 160–163.
17. H. Chang et al.: Surviving the SOC Revolution: a Guide to Platform-Based Design, Kluwer Academic Publisher, 1999.

Algorithms and Architectures for Use in FPGA Implementations of Identity Based Encryption Schemes

Tim Kerins[1], Emanuel Popovici[2], and William Marnane[1]

[1] Dept. of Electrical and Electronic Engineering,
University College Cork, Cork City, Ireland.
{timk,liam}@rennes.ucc.ie
[2] Dept. of Microelectronic Engineering,
University College Cork, Cork City, Ireland.
e.popovici@ucc.ie

Abstract. In this paper algorithms and architectures for new $GF(3^m)$ multiplier and inverter components are presented. It is described how they can be utilized as part of a hardware implementation of an Identity Based Encryption (IBE) scheme. The main computation, the Tate pairing in such a scheme in outlined and it is illustrated how it can be implemented on reconfigurable hardware using these components.

1 Introduction

Public key cryptography is the means by which two unfamiliar hosts generate a shared secret for bulk communication across an insecure channel. Typically this involves the use of each users *public* and *private* keys and *public key certificates*. In order for Alice to negotiate a shared secret with Bob both user's need to have their public key listed with some *key certificate authority* [1]. An Identity Based Encryption (IBE) scheme is a public key cryptosystem where any string (such as an email address) is a valid public key. The advantages of an email scheme based on IBE include: senders can mail recipients who have not set up a public key, and there is no need for an online lookup to obtain the recipients public key certificate. The idea of such a scheme was originally proposed by Shamir [2]. Recently a full IBE scheme (the Boneh-Franklin scheme) has appeared in the literature [3]. This system works on points on an *elliptic curve* E over an underlying *Galois field*. The principal computation in such a scheme is the *Tate pairing*, $\mathcal{T}$. It is desirable to implement this calculation in hardware as it represents the major computational bottleneck in the implementation of IBE schemes. FPGAs represent a cost-effective, versatile implementation platform.

The calculation of the Tate pairing requires a large number of multiplicative operations in the underlying Galois field. Typically extension fields of characteristic $p = 2$ or fields where p is a large prime are preferred for hardware implementation of cryptographic ciphers due to the simplicity of the underlying arithmetic. These have been well studied in the literature [4]. However, in an

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 74–83, 2004.

IBE scheme operations are implemented in $GF(p^m)$ and also $GF(p^{km})$. For maximum security it is required that k be as large as possible while satisfying certain other security criteria [5]. For fields of characteristic three, $p = 3$ the $GF(p^m)$ extension k takes its maximum possible value of $k = 6$. Using a base Galois fields of characteristic 3, $GF(3^m)$, some of the most secure IBE schemes are possible [5]. To date, few hardware architectures for $GF(3^m)$ arithmetic suitable for cryptographic purposes have appeared in the literature. However, architectures for bit serial and digit serial $GF(3^m)$ multipliers appeared in [6] and [7] respectively. A flexible combined multiplication-division $GF(3^m)$ arithmetic processor was published in [8].

In this work a new algorithm for Extended Euclidean Algorithm (EEA) based inversion in $GF(p^m)$ is described, and an efficient slice-wise architecture in the important case of $GF(3^m)$ is implemented. A literature search revealed no designs similar to this in the open literature despite the fact that $GF(3^m)$ inversion is required to implement the Tate pairing. An algorithm for *coefficient-wise* Most Significant Coefficient First (MSC) $GF(p^m)$ multiplication suggested but not discussed in [7] is also presented. A slice-wise architecture for this algorithm for $GF(3^m)$ is also described. The functionality of these multiplier and inverter algorithms was verified against a reference C++ software implementation and the Mathematica package. The aim of this contribution is to describe the basic arithmetic units necessary to perform the Tate pairing calculation and outline how they may be incorporated into a full hardware Tate pairing accelerator.

2 Identity Based Encryption and the Tate Pairing

2.1 The Boneh-Franklin IBE Scheme

IBE relies on the existence of a transformation, $\mathcal{T}$ the Tate Pairing that takes two points P, Q on E as inputs and outputs an element f of the Galois field, i.e. $\mathcal{T}(P, Q) = f$. There is a group structure on the points of E so a point P can be multiplied by an integer a to get the point aP also on E. The security of the scheme relies on the fact that $\mathcal{T}(aP, bQ) = \mathcal{T}(P, Q)^{ab}$, for elliptic curve points P,Q and integers a, b. For example for when Alice e-mails Bob at `bob@bobscompany.com` she can use his e-mail address as his public key string. The system parameters contain two public points P and xP, and an integer x known only to the *private key generator* (PKG). Bobs public key string is hashed into a point Q on E. Bobs corresponding private key is the point xQ. To encrypt, Alice picks a random number r and performs the Tate pairing $s = \mathcal{T}(Q, xP)^r$. This Galois field element s represents the session key. Alice sends the elliptic curve point rP to Bob across the insecure channel. Bob is then able to retrieve the session key by getting xQ from the PKG. He can then compute shared secret s via another calculation of the Tate pairing, $\mathcal{T}(xQ, rP) = \mathcal{T}(Q, P)^{xr} = \mathcal{T}(Q, xP)^r = s$. An evesdropper Eve sees only P, xP, rP and Q and it is difficult to compute s with only this information [3]. A flowchart of operation of the Boneh-Franklin IBE scheme is illustrated in Fig. 1. More detailed descriptions of aspects of this scheme can be found in [3] and [4].

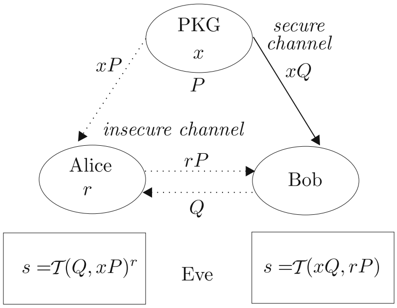


Fig. 1. Flowchart of operation of the Boneh-Franklin IBE encryption scheme

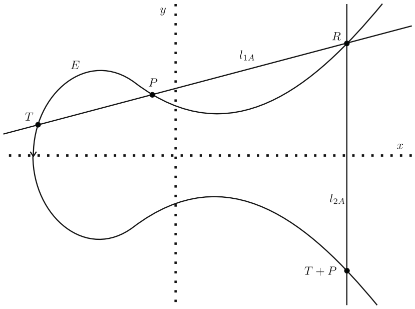


Fig. 2. Elliptic curve point addition

Table 1. Point addition and point doubling on E

PADD : $(x_{t'}, y_{t'}) = (x_t, y_t) + (x_p, y_p)$	PDBL : $(x_{t'}, y_{t'}) = [2](x_t, y_t)$
$\lambda_A = (y_p - y_t)/(x_p - x_t)$	$\lambda_D = -a/y_t$
$x_{t'} = \lambda_A^2 - (x_t + x_p)$	$x_{t'} = x_t + \lambda_D^2$
$y_{t'} = (y_t + y_p) - \lambda_A^3$	$y_{t'} = -(y_t + \lambda_D^3)$
$l_{1A}(x, y) = y - y_t - \lambda_A(x - x_t)$	$l_{1D}(x, y) = y - y_t - \lambda_D(x - x_t)$
$l_{2A}(x) = x - x_{t'}$	$l_{2D}(x) = x - x_{t'}$

2.2 Arithmetic on Supersingular Curves

A supersingular elliptic curve E over a field $GF(3^m)$ is defined as the set of pairs $(x, y), x, y \in GF(3^m)$ such that

$$E : y^2 = x^3 + ax + b, \; a \neq 0, b \in GF(3^m) \tag{1}$$

The curve in (1) also defines an elliptic curve over the field extension $GF(3^{6m})$ of $GF(3^m)$, as $GF(3^m) \subset GF(3^{6m})$. Point addition of two points $T, P \in E$ is defined by constructing the line l_{1A} through points T and P. It intersects E at a third point R. A vertical line l_{2A} is constructed through R and the point where this line intersects E is defined as $T + P$. Point doubling is defined in a similar manner except l_{1D} is defined as the tangent to E at T, and the intersection of R and l_{2D} is defined as $[2]T$. The point addition operation in outlined in Fig. 2. The lines $l_{1A}, l_{2A}, l_{1D}, l_{2D}$ can be considered as functions on the curve E. Algebraically the point addition operation (PADD) and the point doubling operation (PDBL) for supersingular curves E over curves of characteristic 3 are described in Table 1. Note the need for multiplication, inversion, addition and subtraction over $GF(3^m)$.

2.3 The Tate Pairing and Millers Algorithm

Over fields of characteristic three the Tate pairing is a transformation from two elliptic curve points $P, Q \in E(GF(3^{6m}))$ to an element $f \in GF(3^{6m})$,

$t(P, Q) \rightarrow f$. It is computed as outlined in Algorithm 1 [5]. In practice $t(P, Q')$ is calculated to guarantee a meaningful result. Here $Q' = Q + S$ by PADD and S is a random point on $E(GF(3^{6m}))$. Point P is chosen so that $P \in E(GF(3^m))$, i.e. P is an element of a *subfield curve* as this simplifies the calculation. The number l is the largest prime which divides $3^{6m} - 1$. Let $n = \lfloor \log_2(l) \rfloor - 1$ and l_i be the i^{th} bit of l. The accumulators f_1 and f_2 in Algorithm 1 are elements of $GF(3^{6m})$. Using a method described in [5] multiplications of two $GF(3^{6m})$ field elements are are carried out in 36 $GF(3^m)$ multiplications. This multiplication in the larger field is indicated by the symbol $*$ in Algorithm 1.

Algorithm 1 : Millers Algorithm

```
Initialize : $T = P$, $f_1 = 1$, $f_2 = 1$,
             $Q' = (x_{q'}, y_{q'}) = Q + S$
Calculate : for $l_i$ in $n$ downto 1 do
    $T = [2]T$ by PDBL                                   }
    $f_1 = f_1^2 * l_{1D}(x_{q'}, y_{q'}) * l_{2D}(x_s)$  } MDBL
    $f_2 = f_2 * l_{2D}(x_{q'}) * l_{1D}(x_s, y_s)$       }
    if $l_i = 1$ then
        $T = T + P$ by PADD                                  }
        $f_1 = f_1 * l_{1A}(x_{q'}, y_{q'}) * l_{2A}(x_s)$   } MADD
        $f_2 = f_2 * l_{2A}(x_{q'}) * l_{1A}(x_s, y_s)$      }
    end if
end do
Return : $f = (f_1/f_2)$
```

A flowchart of the Tate pairing calculation in terms of the underlying $GF(3^m)$ multiplier and divider cores is illustrated in Fig. 3. As illustrated the fundamental operations in this calculation are $GF(3^m)$ multiplication and inversion. A proposed hardware architecture based on 25 multipliers and an inverter is illustrated in Fig. 4. Here R, M and I represent $GF(3^m)$ registers, multipliers and inverters respectively. Using this type of architecture the updating of f_1 and f_2 from Algorithm 1 can be carried out in parallel.

3 MSC Multiplication in $GF(3^m)$

In the well studied binary field $GF(2) = \{0, 1\}$, addition and multiplication are carried out by the logical AND and XOR operations respectively and a single bit is required for storage. This situation is more complex in $GF(3) = \{0, 1, 2\}$ as two bits are now required for storage and propagation of $GF(3)$ and a non-trivial amount of combinational logic is required to implement the basic addition and multiplication operations. As the underlying logical units on many FPGAs are reconfigurable 4:1 lookup tables this makes them a suitable choice of implementation platform for $GF(3)$ arithmetic. The basic $GF(3)$ arithmetic operations can be efficiently mapped to two 4:1 reconfigurable units. We choose the encoding $0 = \{00, 01\}$, $1 = \{10\}$ and $2 = \{11\}$ as the *check if zero* operation

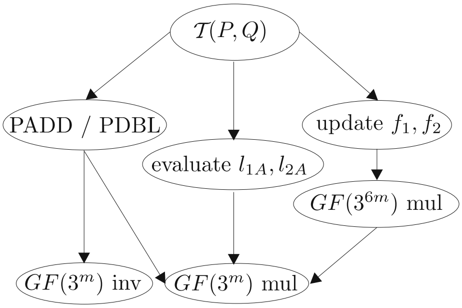


Fig. 3. Flowchart the computational operations in Millers Algorithm in terms of underlying $GF(3^m)$ arithmetic

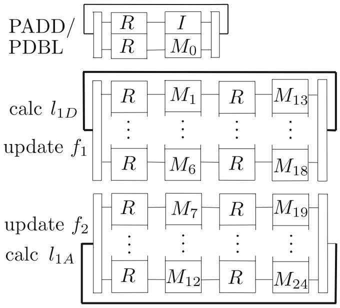


Fig. 4. Proposed hardware architecture for calculation of the Tate Pairing

is achieved by only checking the high bit of a $GF(3)$ element. In our implementations elements of $GF(3^m)$ are stored in registers of $2m$ bits. Thus in an FPGA implementation $2m$ one bit flip-flops (the fundamental storage unit on reconfigurable hardware)are required to store a $GF(3^m)$ element.

Multiplicative operations (multiplication, inversion etc.) in $GF(p^m)$ are performed modulo a specially chosen irreducible polynomial $F = x^m + \sum_{i=0}^{m-1} f_i x^i$. It is useful to define a polynomial f' from F by :

$$f' = \sum_{i=0}^{m-1} f_i x^i = f_{m-1}x^{m-1} + \ldots + f_1 x + f_0, \quad f_i \in GF(p) \tag{2}$$

Using the *most-significant-coefficient-first* (MSC) multiplication (Algorithm 2) the product $C = AB \in GF(p^m)$ is computed in m iterations. On each iteration the partial product $b_i A x \in GF(p^m)$ is calculated and accumulated. Here multiplication by x involves a coefficient-wise shift to the left. After m iterations the product AB is found in the accumulator (Z).

Algorithm 2: MSC Mul in $\mathbf{GF(p^m)}$

```
Input : $A, B \in GF(p^m)$, polynomial $f'$
Initialize : $Z = 0$
Calculate : for $i$ in $m - 1$ downto 1 do
      $Z = Z + b_i A$
      $Z = xZ$
      if $z_m \neq 0$ then
            $Z = Z - z_m f'$
      end if
Return : $C = Z + b_0 A$
```

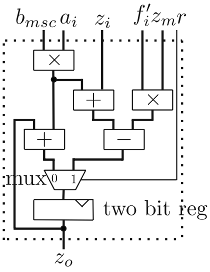


Fig. 5. Calculation slice of $GF(3^m)$ MSC Mul

The basic calculation slice for implementation in fields over $GF(3)$ is illustrated in Fig. 5. The blocks $+$,$-$ and $\times$ represent $GF(3)$ arithmetic combinational logic. The MSC Mul architecture involves a chain of m such slices (0 $\ldots m-1$) with each slice updating a coefficient (2-bits) of Z, along with a $m-1$ bit counter for control line r (r-low indicating the final iteration). All data lines are 2-bit lines, and a 1-bit control line r indicates the final iteration of the calculation in which Z updates differently. Fixed inputs to each slice are coefficients of A and f' (2-bit coefficients a_i and f'_i from (2)). The input b_{msc} represent the most significant coefficient of B (outputted from a 2-bit shift register). The feedback coefficient z_m, is the z_o output from the m^{th} slice. If the overflow coefficient z_m is nonzero the shifted value of z_i is scaled by $z_m f'_i$. The slice in Fig. 5 can be efficiently synthesized to FPGA technology and was found to occupy fourteen 4:1 lookup tables and four 1-bit flip flops. Clock frequency for a single slice is 131 MHz on Xilinx VirtexE device.

4 Inversion in $GF(3^m)$ Based on the EEA

In this section we present a new algorithm for inversion in $GF(p^m)$ based on the Extended Euclidean Algorithm, (EEA) (Algorithm 3). It is a generalization of the $GF(2^m)$ inversion algorithm found in [9]. The inverse of input A,A^{-1} is calculated after $2m$ iterations. It is useful to define the polynomial f'' from F by:

$$f'' = \sum_{i=1}^{m} \frac{f_i}{f_0} x^i = \frac{1}{f_0} x^m + \ldots + \frac{f_2}{f_0} x^2 + \frac{f_1}{f_0} x \quad f_i \in GF(p) \tag{3}$$

This is necessary to perform modulo reduction if there is an overflow coefficient in the division by x in Algorithm 3. Polynomials R and S and polynomials U and V update independently of each other (Algorithms 4 and 5) and this allows for the definition of two sperate calculation slices for our implementation in $GF(3^m)$. These are illustrated in Figs 6 and 7 respectively. The architecture for a $GF(3^m)$ inverter based on these slices includes an $(m+1)$ slice chain of RS slices $(0\ldots m)$ and an m slice chain of UV slices $(0\ldots m-1)$, along with a global control of a $2m$ bit iteration counter and a $2m$ bit bidirectional shift register for tracking the value of δ. The $\delta = 0$ condition is efficiently checked by the status of the least significant bit of this register.

The $GF(3)$ RS calculation slice, Fig 6, as described in Algorithm 4, updates coefficients (2-bits) of the R and S polynomials. Data lines are 2-bit lines and control is via the 1-bit i, r and d lines. The control i-low indicates that two bit registers for holding coefficients of R and S are initialized to values of a_i and f_i respectively. For the rest of the calculation the resisters are updated via the d and r lines. The condition d-low indicates the $\delta = 0$ condition in Fig. 6 and the r-low control indicates the condition $r_m = 0$ (Algorithm 4) and is obtained by tapping the high bit of the r_o output of the most significant RS slice. The input q is the 2-bit $GF(3)$ element $q = s_m/r_m$ which is calculated on each iteration. The slices are chained so the r_o, s_o from in the j^{th} slice connect to the r_i,s_i

Algorithm 3 : EEA Inv in $\mathbf{GF(p^m)}$

```
Input: $A \in GF(p^m)$, irreducible $F$
Initialize: $S = F$, $R = A$, $U = 1$,
            $V = 0$, $\delta = 0$, $q = 0$, $t = 0$
Calculate : for $i$ in 0 to $2m - 1$ do
    if($r_m = 0$) then
        $R = xR$
        $U = (xU)$ mod $F$
        $\delta = \delta + 1$
    else
        $q = s_m / r_m$
        $S = S - qR$
        $V = V - qU$
        $S = xS$
        if($\delta = 0$) then
            $t = R$, $R = S$, $S = t$
            $t = U$, $U = V$, $V = t$
            $U = (xU)$ mod $F$
            $\delta = \delta + 1$
        else
            $(U = U/x)$ mod $F$
            $\delta = \delta - 1$
        end if
    end if
Return : $A^{-1} = (U/r_m)$
```

Algorithm 4 : EEA Inv RS Update

```
if($r_m = 0$) then
    $R = xR$, $S = S$
else
    $S = S - qR$, $S = xS$
    if($\delta = 0$) then
        $t = R$, $R = S$, $S = t$
    end if
end if
```

Algorithm 5 : EEA Inv UV Update

```
if($r_m = 0$) then
    $(U = xU)$ mod $F$
else
    $V = V - qU$
    if($\delta = 0$) then
        $t = U$, $U = V$, $V = t$
        $U = (xU)$ mod $F$
    else
        $U = (U/x)$ mod $F$
    end if
end if
```

in the $(j+1)^{th}$ slice. A single RS slice was synthesized for the Xilinx VirtexE technology and was found to occupy twenty-two 4:1 lookup tables and four 1-bit flip flops with an estimated clock frequency of 113 MHz.

The operation of the UV calculation slice is outlined in Algorithm 5 and illustrated for $GF(3)$ in Fig. 7. Its operation is more complex than that of the RS slice due to the possible multiplication of U and $V - qU$ by x, and the division of U by x, modulo the irreducible polynomial F. The UV calculation slices are chained so that the u_o, v_o from the j^{th} slice are connected to the u_i, v_i in the $(j+1)^{th}$ slice. In order to perform the division operations the input u'_i in the j^{th} slice is driven by the u_o output in the $(j+1)^{th}$ slice. The inputs of f'_i and f''_i are constant 2-bit coefficients of the polynomials in (2) and (3). The inputs u_0 and u_{m-1} are the least and most significant coefficients of polynomial U and are common to all m slices. These are the u_o outputs of the first and last UV slices in the chain. Similarly the input v'_{m-1} is common to every slice and is given by $v'_{m-1} = v_{m-1} - qu_{m-1}$, generated from the outputs u_o and v_o from the $(m-1)^{th}$ UV slice. The MUXs indicated by stars in Fig. 7 are controlled by the high bit of u_0, u_{m-1} and v'_{m-1} respectively and control the modulo reduction for multiplication and division by x. A UV calculation slice was synthesized for Xilinx VirtexE technology and was found to occupy thirty-two 4:1 lookup tables and four 1-bit flip flops, with an estimated clock frequency of 96 MHz.

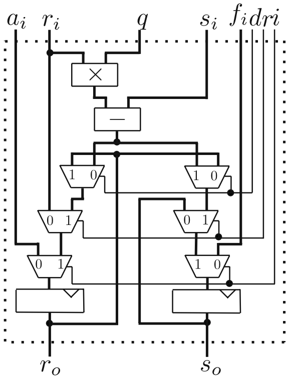


Fig. 6. RS calculation slice of $GF(3^m)$ EEA Inv

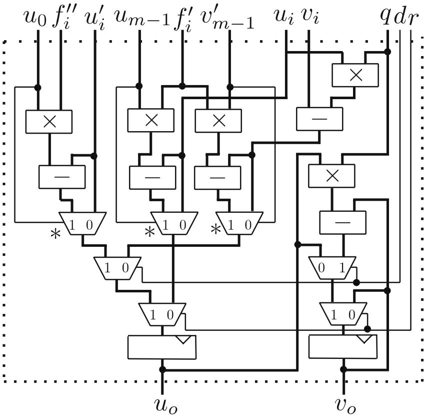

u_0 f_i'' u_i' u_{m-1} f_i' v_{m-1}' u_i v_i $q dr$
u_o
v_o

Fig. 7. UV calculation slice of $GF(3^m)$ EEA Inv

5 Results

The slice-wise $GF(3^m)$ multiplier and inverter cores were captured in the VHDL design language at the RTL level. The designs were synthesized over three different field sizes suitable for cryptography $GF(3^{97})$, $GF(3^{127})$ and $GF(3^{161})$ and on two high performance FPGA technologies, the Xilinx VirtexE, and Vitrex2Pro. Both post-synthesis (PS) and post place-and-route (PPR) timing results are given along with the slice usage (the fundamental FPGA area unit post place-and-route) and the percentage of the chip occupied (Tables 2-5). In [6] and [7] only multiplication over the field $GF(3^{97})$ was considered and a comparison of multiplication time of the MSC Mul design with these designs on a similar technology is presented in Table 6. As seen the MSC Mul has a better performance than that in [6] and a slightly decreased performance than the digit serial multiplier in [7]. However, as this design processes a number of coefficients in parallel the underlying FPGA resource usage in this design is a factor of 4 times larger then the MSC Mul.

It has been illustrated that as field size increases a high clock frequency is still obtainable for our multiplier, Tables 2-3. This is due to the simple slice-wise nature of the designs. The EEA Inv inverter design gives a higher performance than the joint multiplier divider design presented in [8]. Its speed is slower than that of the MSC Mul over similar fields due to the increased complexity of the slices, and the greater number of control signals required to propagate to each slice. A high performance is still achievable as field size increases however, particularly on the Virtex2Pro technology , Table 5. An inversion time of 3.13 μs is achievable on this technology over $GF(3^{97})$.

Both our MSC Mul and EEA Inv designs over $GF(3^m)$ are suitable for implementation on FPGA technology. The calculation slice-wise nature of these designs, which contains an amount of combinational logic followed by two bit

Table 2. MSC Mul implemented on a Xilinx XCV3200E-fg1156 device

$GF(3^m)$	PS/MHz	PPR/MHz	slices	%
97	106.4	68.7	934	2
127	104.7	59.8	1250	3
161	104.7	58.8	1607	4

Table 3. MSC Mul implemented on a Xilinx XCV2VP125-ff1704 device

$GF(3^m)$	PS/MHz	PPR/MHz	slices	%
97	213.9	119.0	985	1
127	210.7	79.9	1253	2
161	210.7	70.5	1634	2

Table 4. EEA Inv implemented on a Xilinx XCV3200E-fg1156 device

$GF(3^m)$	PS/MHz	PPR/MHz	slices	%
97	66.6	47.6	2300	7
127	64.8	41.2	3014	9
161	62.8	32.6	3810	11

Table 5. EEA Inv implemented on a Xilinx XCV2VP125-ff1704 device

$GF(3^m)$	PS/MHz	PPR/MHz	slices	%
97	117.0	61.8	2210	3
127	116.1	50.1	2897	4
161	113.8	44.5	3669	6

Table 6. Comparison of multiplication time of MSC mul design over $GF(3^{97})$ against those appearing in [6] and [7] implemented on Xilinx Virtex2Pro technology.

Design	Time for optimized multiplication / μs
Bit serial multiplier [6]	50.68 (@ 20 MHz)
Digit serial multiplier [7]	0.74 (@ 94.4 MHz)
MSC mul	0.81 (@ 119 MHz)

registers (Figs. 5-7) mirrors the internal structure of the Xilinx FPGA, which contains 4:1 configurable logic elements followed by single bit flip-flips. Excluding global control lines each calculation slice only communicates with its nearest neighbors in the slice chains hence the efficient Virtex2Pro adjacent slice routing resources can be exploited. The more efficient routing resources and more advanced processing on the Virtex2Pro over the VirtexE family of devices are the most probable causes of the increase in performance of when implemented on this technology.

The performance and area usage of our $GF(3^m)$ multiplier and inverter circuits make them suitable for use in an FPGA implementation of Miller Algorithm as outlined in section 2. Due to the low slice usage of the MSC Mul an FPGA processor for calculation of the Tate pairing over $GF(3^m)$ as outlined in Figure 4 becomes feasible (30% of a Xilinx XCV2P125 device). Using this architecture the PADD/PDBL, MDBL and MADD operations from Algorithm 1 can be calculated in $5m$, $50m$ and $38m$ clock cycles respectively. It is probable that the EEA Inv inverter (@ 62.8 MHz) represents the critical path in the design outlined in Fig. 4 as the remainder of the design consists of multipliers, memory elements, a bus architecture and a simple controller for scheduling the operations (not shown). Conservatively estimating the total design runs at 40 MHz, it is estimated that the Tate pairing can be computed on a Xilinx XC2P125 device for the base field $GF(3^{97})$ in approximately 35 ms. This represents a fivefold increase over the results reported in [5] for the same calculation on a 1 GHz Pentium 3 processor.

6 Conclusions and Further Work

In this paper efficient slice-wise multiplication and inversion architectures in $GF(3^m)$ have been described. To the authors knowledge the EEA Inv architecture is the most efficient FPGA implementation to date for inversion over fields $GF(3^m)$. Our MSC Mul multiplier compares well performance wise with recent FPGA implemented $GF(3^m)$ multipliers and utilized a relatively small amount of FPGA resources. Using these multiplier and inverter components as outlined a hardware accelerator for IBE schemes becomes viable by migrating the computation of the Tate pairing to hardware. Work is currently underway to design a full FPGA implementation of an IBE scheme using the multiplier and inverter cores described here.

Acknowledgement. This research was funded as part of an Enterprise Ireland research innovation fund project.

References

1. N. Koblitz: Algebraic Aspects of Cryptography, *Algorithms and Computation in Mathematics Vol. 3* Springer (1999).
2. A. Shamir: Identity Based Cryptosystems and Signature Schemes. *CRYPTO 1984* 47-53 (1984).
3. D. Boneh and M. Franklin: Identity-Based Encryption from the Weil Pairing. *CRYPTO 2001*, Lecture Notes in Computer Science No. 2139, Springer-Verlag, (2001) 213-229
4. I. Blake, G. Seroussi, N. Smart: Elliptic Curves in Cryptography. *London Mathematical Society Lecture Note Series 265*, Cambridge University Press (2000).
5. S. D. Galbraith, K. Harrison and D. Soldera: Implementing the Tate pairing. *Algorithmic Number Theory Syposium, ANTS-V*, Lecture notes in Computer Science No. 2369 Springer Verlag, (2002) 324-337
6. D. Page and N.P. Smart: Hardware Implementation of Finite Fields of Characteristic Three. *CHES 2002*, Lecture Notes in Computer Science No. 2523, Springer-Verlag, (2002) 529-539
7. G. Bertoni, J. Guajardo, S. Kumar, G.Orlando, C. Paar and T. Wollinger: Efficient GF(p^m) Arithmetic Architectures for Cryptographic Applications. *Topics in Crytpology-CT-RSA 2003*, Lecture Notes in Computer Science No. 2612 Springer-Verlag, (2003) 158-175
8. T. Kerins E.M.Popocici W.P. Marnane: Fully Paramaterisable Galois Field Arithmetic Processor over GF(3^m) suitable for Elliptic Curve Cryptography. *Proc. of MIEL 2004*, (2004) 739-742.
9. H. Brunner, A. Curgier and M. Hofstetter: On Computing Multiplicative Inverses in $GF(2^m)$. *IEEE Transactions on Computers* Vol. 42, No. 8. (1993) 1010-1015

Power Analysis Attacks Against FPGA Implementations of the DES

François-Xavier Standaert[1], Sıddıka Berna Örs[2],
Jean-Jacques Quisquater[1], and Bart Preneel[2]

[1] UCL Crypto Group
Laboratoire de Microélectronique
Université Catholique de Louvain
Place du Levant, 3, B-1348 Louvain-La-Neuve, Belgium
{standaert,quisquater}@dice.ucl.ac.be
[2] Katholieke Universiteit Leuven, Dept.ESAT/SCD-COSIC,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium.
{siddika.bernaors,bart.preneel}@esat.kuleuven.ac.be

Abstract. Cryptosystem designers frequently assume that secret parameters will be manipulated in tamper resistant environments. However, physical implementations can be extremely difficult to control and may result in the unintended leakage of side-channel information. In power analysis attacks, it is assumed that the power consumption is correlated to the data that is being processed. An attacker may therefore recover secret information by simply monitoring the power consumption of a device. Several articles have investigated power attacks in the context of smart card implementations. While FPGAs are becoming increasingly popular for cryptographic applications, there are only a few articles that assess their vulnerability to physical attacks. In this article, we demonstrate the specific properties of FPGAs *w.r.t.* Differential Power Analysis (DPA). First we emphasize that the original attack by Kocher *et al.* and the improvements by Brier *et al.* do not apply directly to FPGAs because their physical behavior differs substantially from that of smart cards. Then we generalize the DPA attack to FPGAs and provide strong evidence that FPGA implementations of the Data Encryption Standard (DES) are vulnerable to such attacks.

1 Introduction

Since their publication in 1998 [1], power analysis attacks have attracted significant attention within the cryptographic community. So far, they have been successfully applied to different kinds of (unprotected) implementations of symmetric and public-key encryption schemes. Most published attacks apply to smart cards and only a few articles assess the vulnerability of FPGA implementations to power analysis attacks [2,3]. In this paper, we demonstrate the specificity of this kind of platform in the context of Differential Power Analysis. First, we show that the original attack described in [1] and

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 84–94, 2004.

its most recent improvements [4] do not work properly for FPGAs because their physical behavior is different than smart cards. Then we generalize the power consumption model and apply it to FPGAs. Finally, we describe a correlation attack [4,5] in which we correlate theoretical predictions of the power consumption with real measurements in order to make an efficient use of all the collected data. The resulting attack is more efficient than the popular "multiple-bit" DPA and allows interesting theoretical predictions of the attacks with simulated data. All these techniques are successfully applied to an FPGA implementation of the DES. This is the first result on the vulnerability of an FPGA implementation of a block cipher to power attacks.

This paper is organized as follows. Section 2 presents the hypothesis used to carry out the DPA and Sect. 3 gives a short description of the DES algorithm. Section 4 describes the original DPA attack and underlines why it is not applicable to FPGAs. Sections 5 and 6 investigate a generalized power attack. Section 7 presents some theoretical predictions of the generalized attack and Sect. 8 applies it to real measurements. Finally, conclusions are presented in Sect. 9.

2 Hypothesis

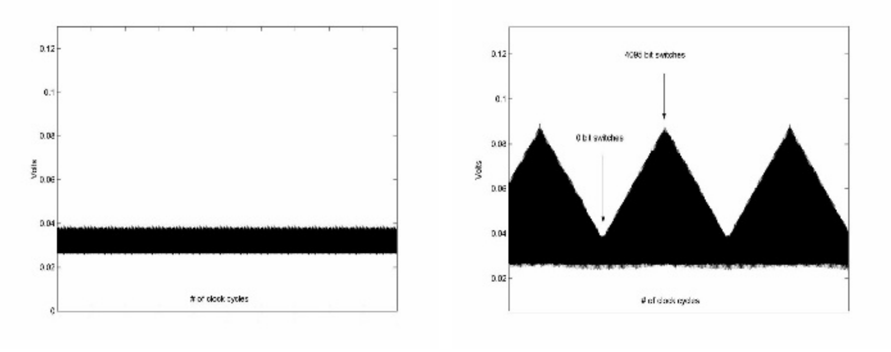
In Differential Power Analysis, an attacker uses a hypothetical model of the device under attack to predict its power consumption. These predictions are then compared to the real measured power consumption in order to recover secret information (*e.g.* secret key bits). The quality of the model has a strong impact on the effectiveness of the attack and it is therefore of primary importance.

While little information is available on the design and implementation of FPGAs (much of the information is proprietary), we can make assumptions about how commercial FPGAs behave at the transistor level. The most popular technology used to build programmable logic is static RAM[1], where the storage cells, the logic blocks and the connection blocks are made of CMOS gates. For these circuits, it is reasonable to assume that the main component of the power consumption is the dynamic power consumption. For a single CMOS gate, we can express it as follows [7]:

$$P_D = C_L V_{DD}^2 P_{0\rightarrow 1} f \ , \tag{1}$$

where C_L is the gate load capacitance, V_{DD} the supply voltage, $P_{0\rightarrow 1}$ the probability of a $0 \rightarrow 1$ output transition and f the clock frequency. Equation (1) specifies that the power consumption of CMOS circuits is data-dependent. However, for the attacker, the relevant question is to know if this data-dependent behavior is observable. This was confirmed by the following test.

[1] For all the experiments, we used a Xilinx Virtex XCV800 FPGA [6].



(a) Hamming weight

(b) Transitions

Fig. 1. Preliminary test.

Let three 4096-bit vectors be defined as follows. Initially, $a_0 = 00000...001$ and $b_0, c_0 = 00000...000$. Then:

$$\begin{aligned} a_{i+1} &= SL(a_i) \\ b_{i+1} &= b_i \oplus a_i \\ c_{i+1} &= c_i \oplus b_i \ , \end{aligned}$$

where SL is the shift left operator and consecutive values (x_i, x_{i+1}) are separated by a register. It is easy to see that:

- a is a bit-vector with a constant Hamming weight ($H(a) = 1$). The position of the 1-bit inside the vector is periodically incremented from 0 to 4095.
- b is a bit-vector for which the Hamming weight is incremented/decremented from 0 to 4095.
- c is a bit-vector for which the number of bit switches between two consecutive states is incremented/decremented from 0 to 4095.

A design that generates these three vectors was implemented in the FPGA. Figure 1(a) illustrates[2] the power consumption of the vectors a and b. Figure 1(b) illustrates the power consumption of vectors a, b and c. From this experiment, we conclude that the power consumption clearly depends on the number of transitions in registers but not on the Hamming weight of the data in the registers.

Based on these considerations, we used the following **hypothesis** to mount power attacks against FPGAs: "an estimation of a device power consumption at time t is given by the number of bit transitions inside the device registers at this time." Predicting the transitions in registers is reasonable since registers usually consume a large part of the power in a design.

[2] Measurement setups for DPA have already been intensively described in the literature. In Fig. 1, we observe the voltage variations over a small resistor inserted in the supply circuit of the FPGA. Every trace was averaged 10 times in order to remove the noise from our measurements.

3 The Data Encryption Standard

In 1977, the Data Encryption Standard (DES) algorithm [8] was adopted as a Federal Information Processing Standard for unclassified government communication. Although a new Advanced Encryption Standard (AES, [9]) was selected in October 2000, DES is still widely used, particularly in the financial sector. DES encrypts 64-bit blocks with a 56-bit key; its main operations are permutations, substitutions and XOR operations. DES is an iterated block cipher that applies 16 key-dependent transformations called rounds to the plaintext. This structure allows for very efficient hardware implementations.

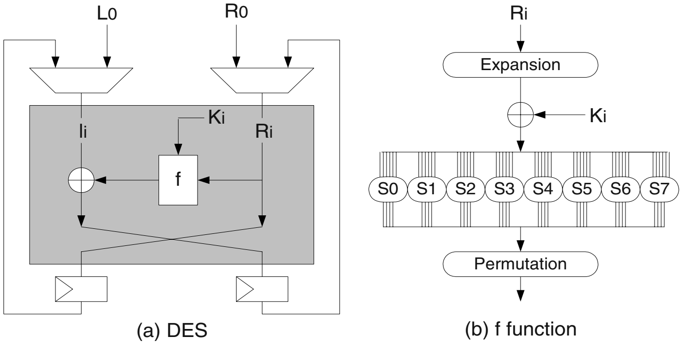
The plaintext is first permuted by a fixed permutation *IP*. Next the result is split into two 32-bit halves, denoted with L (left) and R (right) to which a round function is applied 16 times. The ciphertext is calculated by applying the inverse of the initial permutation *IP* to the result of the 16th round.

The secret key is expanded by the key schedule algorithm from 56 bits to sixteen 48-bit subkeys K_i; each round uses a different subkey K_i. The key schedule consists of bit permutations and rotations. As a consequence, if one can find any subkey, one can derive the complete key immediately (the missing 8 bits can be found by exhaustive search over 256 values).

Finally, the round function is represented in the grey part of Fig. 2(a); it can be described as follows:

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus f(R_i, K_i), \quad i = 0, \ldots 15 \ . \end{aligned}$$

Here $L_{16} \| R_{16}$ is the ciphertext. The details of the nonlinear function f are provided in Fig. 2(b): the right part R_i is first expanded to 48 bits with the E box, which duplicates some bits. Next, the 48-bit subkey K_i is added bitwise modulo 2 (XORed) to $E(R_i)$ and the result of the *XOR* function is sent to eight non-linear S-boxes (S). Each of them has six input bits and four output bits. The resulting 32 bits are permuted by the bit permutation P.



(a) DES

(b) f function

Fig. 2. Data Encryption Standard.

We have performed our experiments on the sequential DES implementation of [10] that takes one clock cycle to perform one round. It is represented in Fig. 2(a).

4 Original Attack

In its original form [1], Differential Power Analysis of DES requires a selection function $D(C, b, K_{Sb,15})$ that we define as computing the value of a bit b which is part of the intermediate vector L_{15} (Fig. 3(a)). One can write b as follows:

$$b = \text{one output bit of } Sb\left((\text{six bits of } L_{16}) \oplus K_{Sb,15}\right) \oplus \text{one bit of } R_{16} \enspace .$$

To implement the DPA attack, an attacker first observes m encryptions and captures the power traces T_i $(1 \leq i \leq m)$ and their associated ciphertexts C_i. No knowledge of the plaintext is required. By guessing six key bits $K_{Sb,15}$, the function D can be computed for each value of i and we can divide the traces into two sets: one set corresponding to $D_i = 0$ and the other one with $D_i = 1$. The traces in each set are then averaged to obtain two average traces A_0 and A_1 and we can compute the difference $\Delta = A_0 - A_1$.

If $K_{Sb,15}$ is correct, the computed value for D will be equal to the actual value of target bit b with probability 1. As the power consumption is correlated to the data, the plot of Δ will be flat, with spikes in regions where D is correlated to the values being processed. If $K_{Sb,15}$ is incorrect, Δ will be flat everywhere. Finally, in a multiple bit attack, the selection function outputs d bits with $d > 1$. It allows to improve the SNR of the attack: if a single-bit DPA attack using N traces has a signal to noise ratio SNR_1, then an *all-zeros-or-all-ones* d-bit DPA attack using N traces will have a ratio $SNR_d = d \cdot SNR_1$.

According to [1], the selection function was chosen because, at some point during a software DES implementation, the software needs to compute its value. When this occurs or any time data containing the selection bits is manipulated, there will be a slight difference in the amount of power dissipated, depending on the values of these bits. However, in the case of RAM-based FPGA implementations, this function does not correctly match the physical behavior of the devices. In a multiple-bit attack, one tries to distinguish bit vectors of different Hamming weights, although it is clear from Fig. 1 that the most significant power differences are related to the switching activity between two states. In the next section, we propose to modify the selection function in order to take into account the physical behavior of the FPGAs.

5 Modified Selection Function

In its original form, the selection function is defined as computing the value of a bit b which is part of the intermediate vector L_{15}. For multiple bit attacks, d bits are computed and we denote them by $D = L_{15}[p_0, p_1, \ldots, p_{d-1}]$, where p_i is the position of the *ith* bit guessed. Distinguishing $D =$“00…0” from

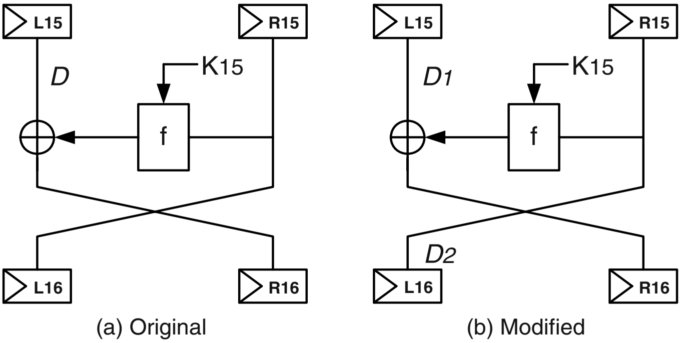


Fig. 3. Selection functions D,D'.

D =“11...1” therefore implies distinguishing vectors of different Hamming weights. A modified selection function can be defined as follows. Let D_1 be the original selection function that involves bits $L_{15}[p_0, p_1, \ldots, p_{d-1}]$. As L_{16} is part of the ciphertext, we can access it. With the notation $D_2 = L_{16}[p_0, p_1, \ldots, p_{d-1}]$, we define a new selection function correlated with the switching activity of the device: $D' = H(D_1 \oplus D_2)$. Based on this selection function, we have mounted successful 4-bit attacks against FPGA implementations of the DES. However, as a multiple bit attack only considers the texts that give rise to 0 or d switches, it is far from optimal and a lot of texts are actually not used. Next, we propose an improved attack based on the correlation between the theoretical power consumption files and the practical measurements.

Note that the same model is used implicitly for software implementations in smart cards: Brier *et al.* clearly state in [4] that the DPA model is based on the Hamming distance between the data handled and an unknown but constant reference state. This constant reference state simply corresponds to the address of an instruction. As a software implementation will load the instruction before loading the data, a DPA attack actually models the switching activity between two states, but one of these states (*i.e.* the instruction address) is constant. Our selection function (with two variable states) is therefore a generalization of the original DPA model.

6 Improved Attack

A correlation attack [4,5] against an FPGA implementation of the DES is divided into three steps. Let N be the number of plaintext/ciphertext pairs for which the power consumption measurements are available. Let K be the secret key used to encrypt. When simulating the attacks, we assume that K is known to the attacker (when the attack is implemented, K is of course unknown).

Prediction phase: For each of the N encrypted plaintexts, the attacker first selects a target S-box for the selection function D' (cf. supra). Then, he predicts the value of D' (*i.e.* the number of bit flips inside a target register between rounds 15 and 16) for the 2^6 key guesses. The result of the prediction phase is an $N \times 2^6$ **selected prediction matrix** containing integers between 0 and 4. For simulation purposes, it is also interesting to produce the **global prediction matrix** that contains the number of bit flips inside all the registers[3] of the design, for all the cycles. That is, if the encryption is performed in 16 clock cycles, we obtain an $N \times 16$ matrix, containing integers between 0 and 64. This is only feasible if the key is known. According to the hypothesis of Sect. 2, these matrices give estimations for the power consumption of the device.

Measurement phase: During the measurement phase, we let the FPGA encrypt the same N plaintexts with the same key, as we did in the prediction phase. While the chip is operating, we measure the power consumption for the 16 consecutive clock cycles. Then, the power consumption trace of each encryption is averaged 10 times in order to remove the noise from our measurements and we store the maximum value of each encryption cycle so that we produce a $N \times 16$ matrix with the power consumption values for all the texts, cycles. We denote it as the **global consumption matrix**.

Correlation phase: In the correlation phase, we compute the correlation coefficient between the 16th column of the global consumption matrix (corresponding to 16th round targeted by the prediction phase) and all the columns of the selected prediction matrix (corresponding to all the 2^6 key guesses). If the attack is successful, we expect that only one value, corresponding to the correct key guess, leads to a high correlation coefficient.

An efficient way to perform the correlation between theoretical predictions and real measurements is to use the Pearson coefficient. Let T_i denote the *ith* measurement data (*i.e.* the *ith* trace) and T the set of traces. Let P_i denote the prediction of the model for the *ith* trace and P the set of such predictions. Then we calculate:

$$C(T,P) = \frac{E(T.P) - E(T).E(P)}{\sqrt{Var(T).Var(P)}} \quad . \tag{2}$$

Here $E(T)$ denotes the mean of the set of traces T and $Var(T)$ its variance. If this correlation is high, it is usually assumed that the prediction of the model, and thus the key hypothesis, is correct.

Finally, theoretical predictions of the attack can be performed by using the global prediction matrix in place of the global consumption matrix. As the global prediction matrix contains the number of bit switches inside all the registers, it represents a theoretical noise free measurement and may help to determine the minimum number of texts needed to mount a successful attack,

[3] Note that since the same key is used for all the measurements, the power consumption of the key schedule is fixed and may be considered as a DC component that we can neglect as a first approximation.

i.e. an attack where the correct key guess leads to the highest correlation coefficient. This is investigated in the next section.

7 An Attack Using Simulated Data

Let our target for the selection function D' be the 4 bits of the register L that are affected by the 6 Most Significant Bits (MSBs) of the round key 16. It corresponds to the output bits of S-box $S0$. Let the number of measurements be $N = 4096$. A theoretical prediction of our attack can be performed by running it with simulated data.

In the first step of the simulated attack, we produce the **selected prediction matrix** and **global prediction matrix** as defined in the previous section. Thereafter, we perform the correlation phase between these two matrices. If the attack is successful, we expect that only one value, corresponding to the correct key guess, leads to a high correlation coefficient. Figure 4 shows that this expectation is fulfilled and the correct 6 MSBs of the last round key guess are $1\mathrm{E}_{hex} = 30_{dec}$.

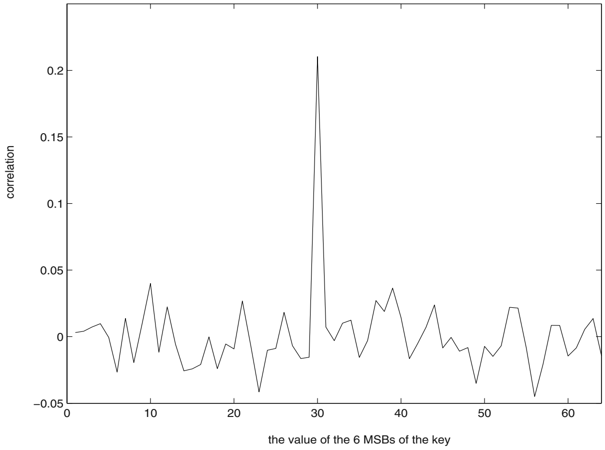


Fig. 4. A correlation attack using simulated data

As an attacker would like to learn the minimum number of plaintexts that are necessary to find the key, we have also calculated this correlation coefficient for different values of $N : 0 \leq N \leq 2\ 000$. As shown in Fig. 5, after approximately 400 plaintexts the right 6 key-bits can be distinguished from a wrong guess. We may therefore say that the attack is **theoretically successful** after about 400 texts.

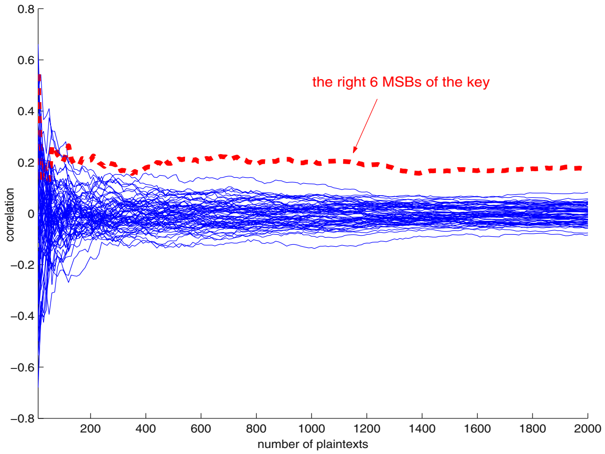


Fig. 5. A correlation attack using simulated data for different N values.

8 An Attack Using Practical Measurements

When attacking a device in practice, the selected prediction matrix stays unchanged while we replace the global prediction matrix by the measured **global consumption matrix**. Therefore, we let the FPGA encrypt the same $N = 4096$ plaintexts with the same key as we did in the previous section and produced the matrix as described in Sect. 6.

In order to identify the correct 6 MSBs of the final round key, we used the correlation coefficient again. As it is shown in Fig. 6, the highest correlation occurs when the key guess is $1\text{E}_{hex} = 30_{dec}$. This value corresponds to the correct 6 MSBs of the round key 16. As a consequence, the attack is **practically successful**, *i.e.* the selected prediction matrix is sufficiently correlated with the real measurements and we can extract the key information. Remark that comparing Fig. 4 and Fig. 6 clearly allows to evaluate the effect of the noise in our measurements.

It is important to note that more bits of the final subkey may be found using exactly the same set of measurements. The attacker only has to modify the selected prediction matrix in order to target different key bits. As every subkey consists of 48 bits and the master key of 56 bits, we can easily find the last 8 key bits by exhaustive search.

Finally, a more accurate prediction of the FPGA power consumption could allow to improve the efficiency of the attack. A notable feature of FPGAs is that they contain different components (*e.g.* logic blocks, connections) with a different power consumption because of a different effective load capacitance. As a consequence, the power consumption of FPGA designs does not only depend on their switching activity but also on the internal components used. Recent works [11] tried to identify these important resources in the FPGA architecture and to characterize their power consumption. This could be used to improve the power consumption predictions.

In practice, more accurate estimations about the most power hungry components of an FPGA design can be derived from the delay information that is generated by most implementation tools [12]. As an input delay represents the delay seen by a signal driving that input due to the capacitance along the wire, large (*resp.* small) delay values indicate that the wire has a large (*resp.* small) capacitance. Based on the reports automatically generated by implementation tools, one may expect to recover a very accurate information about the signals that are driving high capacitances. The knowledge of the implementation netlists with delay information is therefore relevant. It will allow an attacker to improve the attack.

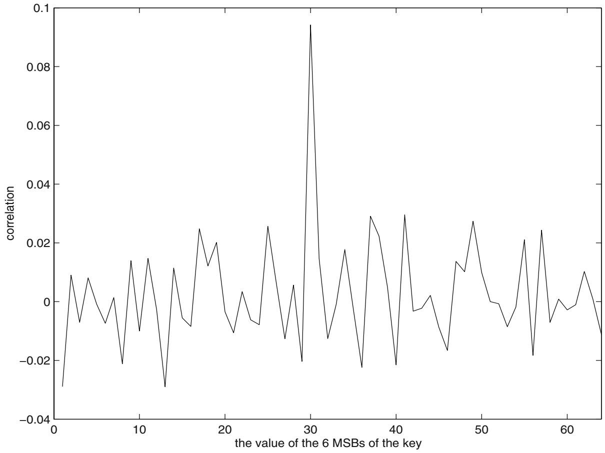


Fig. 6. A correlation attack with real measurements

9 Conclusion

This paper demonstrated the specific properties of SRAM-based FPGAs in the context of Differential Power Analysis. As the original attack of [1] does not apply 'as it is' to these reconfigurable devices, we generalized the model of power attacks in order to take into account the physical behavior of FPGAs. The resulting attack is effective with a reasonable number of measurements. It is more efficient than the popular "multiple-bit" DPA and allows interesting theoretical predictions of the attacks with simulated data. Moreover, the power consumption model and therefore the efficiency of the attack could be improved in different ways, for example by taking advantage of implementation netlists and delay information as we suggest in Sect. 8. Other block ciphers (*e.g.* AES Rijndael) are also vulnerable to our methods. These results confirm that power analysis presents a realistic threat for FPGA implementations of block ciphers.

References

1. P.Kocher, J.Jaffe, B.Jun, *Differential Power Analysis*, in the proceedings of CRYPTO 99, Lecture Notes in Computer Science 1666, pp 398-412, Springer-Verlag.
2. S.B.Ors, E.Oswald, B.Preneel, *Power-Analysis Attacks on an FPGA – First Experimental Results*, in the proceedings of CHES 2003, Lecture Notes in Computer Science, vol 2279, pp 35-50, Springer-Verlag.
3. F.X. Standaert, L.van Oldeneel, D.Samyde, J.J. Quisquater, *Power Analysis of FPGAs, How Practical is the Attack?*, in the proceedings of FPL 2003, Lecture Notes in Computer Science, vol 2278, pp 701-711, Springer-Verlag.
4. E.Brier, C.Clavier, F.Olivier, *Optimal Statistical Power Analysis*, IACR e-print archive 2003/152.
5. S.B.Ors, F.Gurkaynak, E. Oswald, B. Preneel *Power-Analysis Attack on an ASIC AES implementation*, in the proceedings of ITCC 2004, Las Vegas, April 5-7 2004.
6. Xilinx: *Virtex 2.5V Field Programmable Gate Arrays Data Sheet*, http://www.xilinx.com.
7. J.M.Rabaey, *Digital Integrated Circuits*, Prentice Hall International, 1996.
8. National Bureau of Standards. *FIPS PUB 46*, The Data Encryption Standard, Jan 1977.
9. NIST Home page, http://csrc.nist.gov/CryptoToolkit/aes/.
10. G.Rouvroy, F.X.Standaert, J.J.Quisquater, J.D.Legat, *Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES*, in the proceedings of FPL 2003, LNCS 2778, pp 181-193, Springer-Verlag, 2003.
11. L.Shang, A.Kaviani, K.Bathala, *Dynamic Power Consumption in Virtex2 FPGA Family*, FPGA 2002, Monterey, California, 2002.
12. L.T. Mc Daniel, *An Investigation of Differential Power Analysis Attacks on FPGA-based Encryption Systems*, Master Thesis, Virginia Polytechnic Insitute, May 29, 2003.

Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer

Maya Gokhale, Janette Frigo, Christine Ahrens, Justin L. Tripp, and Ron Minnich

Los Alamos National Laboratory

Abstract. Recently, the appearance of very large (3 – 10M gate) FPGAs with embedded arithmetic units has opened the door to the possibility of floating point computation on these devices. While previous researchers have described peak performance or kernel matrix operations, there is as yet relatively little experience with mapping an application-specific floating point loop onto FPGAs. In this work, we port a supercomputer application benchmark onto Xilinx Virtex II and Virtex II Pro FPGAs and compare performance with three Pentium IV Xeon microprocessors. Our results show that this application-specific pipeline, with 12 multiply, 10 add/subtract, one divide, and two compare modules of single precision floating point data type, shows speed up of 10.37×. We analyze the tradeoffs between hardware and software to characterize the algorithms that will perform well on current and future FPGA architectures.

1 Introduction

Over the past decade, Reconfigurable Computing (RCC) using Field-Programmable Gate Arrays (FPGAs) has demonstrated speed-ups of one to two orders of magnitude on data- and compute-intensive processing tasks involving fixed point computation on small integers, typically in signal and image processing applications. Floating point computation was not mapped to FPGAs due to the large operand size (32- or 64-bit) and excessive area consumed by floating point arithmetic units on configurable logic cells. Recently, that limitation of FPGAs appears to be receding: 3–10 million gate FPGAs with embedded processors, memories, and arithmetic units have become available, making it feasible to consider a broader range of applications than traditional signal and image processing, including those requiring floating point operations. Studies comparing floating point performance of FPGAs vs. high performance microprocessors [1] suggest that peak FPGA floating-point performance is growing significantly faster than peak floating-point performance for a CPU. Other studies [2,3] also suggest that modern FPGAs may be competitive with microprocessors on dense matrix operations such as matrix multiply and LU decomposition.

However, it is well-known in the supercomputing community that peak performance and dense matrix kernel operations are far from accurate predictors of realized performance of a complete application. Memory access patterns, cache

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 95–104, 2004.

behavior, control flow, and inter-processor communication result in actual performance that is well below peak. For example, applications run on a cluster supercomputer often realize no more than 50–80% of theoretical peak [4], reducing a 30 TFLOP machine to 15 TFLOPs.

The purpose of the work described below is to better quantify the performance of FPGA-based floating point computation on real applications by mapping a portion of an application (as opposed to a kernel) onto an FPGA. We compare the performance of an application-specific (single precision) floating point pipeline mapped to the Virtex family of FPGAs to execution on comparable microprocessors.

Reconfigurable Computing using FPGAs exploits "spatial parallelism", the ability, for example, to unroll a computational block directly onto hardware, executing the entire block in parallel. This ability is not available on a CPU, which depends on a fast clock rate to increase performance. FPGAs use a significant amount of spatial parallelism to compensate for having a clock speed that is an order of magnitude slower than that of a CPU.

In this paper we describe our FPGA implementation of a floating-point intensive supercomputing application called "radiative heat transfer" [5]. First, we describe other floating-point applications implemented on FPGAs and discuss floating-point libraries for FPGAs. Next, we give an overview of the radiative heat transfer application. We describe how we parallelized the inner loop of the application, which is the most computationally intensive portion of the program. We present performance results of the inner loop on three Intel Pentium IV Xeon workstations and compare that to the performance of our implementation on Xilinx Virtex II and Virtex II Pro FPGAs. Finally, we provide our conclusions.

2 Related Work

Using FPGAs for floating-point operations is not new. Past efforts exploring floating point include exploration by Virginia Tech[6], a re-evaluation at Clemson[7] and a library produced at Northeastern[8]. These efforts demonstrate the viability of using floating-point on FPGAs. FPGAs are viable targets because they can be programmed to include many concurrent floating-point operations[1]. Earlier work [9] found that FPGAs were not fast enough to be competitive with general purpose processors for floating point. However, current generations have increased performance with faster logic and embedded multipliers [10]. This increased performance may allow FPGAs to be used for floating-point in areas normally reserved for supercomputers.

FPGAs offer several advantages when used to calculate floating-point operations. First, FPGAs offer a high degree of flexibility, where they can provide a customized solution for a given floating-point algorithm. Second, due to the available concurrency, an FPGA can provide a floating-point solution that is faster than a general purpose processor. Third, FPGAs are based on SRAM, and thus they track trends in transistors (e.g. "Moore's Law") more closely than general purpose processors. FPGAs take advantage of transistor density to provide high levels of concurrency. Offsetting those advantages are the slow clock speed relative to microprocessors and the relatively large area required by floating point operands and operations, which limits spatial parallelism opportunities.

Several commercial [11,12] and open source [8,10] libraries are available for creating floating-point circuits. For our implementation of the radiative heat transfer algorithm, we chose the FPLibrary, a VHDL library of hardware operators for floating-point computation, developed by the Arénaire project [13]. The FPLibrary meets three important qualifications. First, it is written in VHDL in a platform-independent manner. This allows designs to be easily targeted to different FPGA architectures. Second, the library implements add, multiply and divide floating point operations which are required for this algorithm. Third, the modules and floating-point types have parameterizable bitwidths, so that we can easily program the library for single, double or arbitrary sized floating point types. FPLibrary is used to leverage the advantages of FPGAs to implement the core of a supercomputing application.

3 Description of the Monte Carlo Radiative Heat Transfer Simulation

Monte Carlo radiative heat transfer simulation was chosen for implementation on an FPGA, because it contains computationally intensive floating-point operations. It has been run on a SPARCStation computer cluster [14] as well the Cyber 205 supercomputer [5]. It is a real world problem, because it models the geometry of a laser isotope separation (LIS) unit to accurately determine the radiant exchange factors among the surfaces. This is an important component of the isotope separation process simulation.

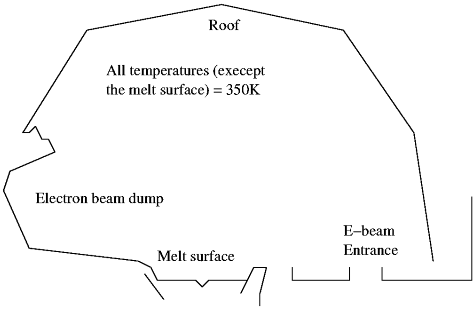


Fig. 1. Test Geometry for Radiative Heat Transfer

The radiative heat transfer simulation is a Monte Carlo application that traces a large number of photons emitted from the surfaces of a 2-D enclosure (Figure 1). The simulation records how many photons emitted from each surface i were absorbed at surface j. This information is used to compute a heat transfer coefficient between each pair of surfaces, i and j. It is a Monte Carlo application because it uses random values to determine characteristics of an emitted photon's path and because it traces a large number of photons.

In the algorithm, **N** photons are emitted (with randomly chosen characteristics) from each surface of an **m**-sided polygon. The algorithm follows the path of each emitted photon. It identifies the surface of intersection, which is the most computationally intensive portion of the algorithm. Next, a random number determines whether the photon is absorbed into the surface, reflected off of it, or transmitted through it. The photon is followed until it is transmitted, absorbed or lost. This algorithm is designed to calculate intersections assuming a convex chamber. There is also a more sophisticated version which works with both convex and concave surfaces, and is the subject of future work.

```
for each surface in the m−surface polygon
    for each of N photons emitted from this surface
        Emit a photon with random characteristics from this surface
        while the photon is not absorbed, transmitted or lost
            for each side in the polygon
                if the current side is not the emitted side
a   b   c   d
                    Check if the photon intersected with this side
                end if
            end for
            Randomly determine if the photon is absorbed, transmitted, reflected or lost
        end while
    end for
end for
```

Fig. 2. Radiative Heat Transfer algorithm loop structure. Loop "d" is implemented on the FPGA.

The parallel version of the algorithm distributes at the "task" level. The pseudo-code for each task is summarized in Figure 2. In loop "a", a task iterates through the **m** surfaces of the polygon and traces the **N** photons emitted from each surface. For each surface, a **for** loop ("b") iterates through each photon emitted, then an inner **while** loop ("c") checks if the photon is still active before following it to its next surface intersection. Inside the **while** loop, an inner **for** loop ("d") computes the surface intersection, then the random number generator determines if the photon is absorbed, reflected, transmitted or lost.

When considering which part of the algorithm to implement on the FPGA, we decided that parallelism at the task or surface level was too coarse, and would not fit on currently available FPGAs. At the **while** loop level, tracing one photon's path until it is not active may be possible in terms of fitting on an FPGA, but there are dependencies carried between loop iterations that make the implementation more complex and limit parallelism. At the inner **for** loop level, where the algorithm checks for the surface of intersection, the code is straightforward to realize on an FPGA, since the loop iterations are independent of each other and can be spatially replicated on the FPGA.

```
float x1[NSM], x2[NSM], y1[NSM], y2[NSM], delx[NSM], dely[NSM], sqln[NSM], rhs[NSM];

delxs = delx[s];  delys = dely[s];  rhss = rhs[s];
x1s = x1[s];  y1s = y1[s];  x2s = x2[s];  y2s = y2[s];  sqlns = sqln[s];

/*  compute intersection points*/
det = ex*delys - ey*delxs;
absdt = fabs(det);
if(absdt <= epsdet0) det = epsdet0;
dtinv = 1.0/det;
xi = dtinv * (delxi*rhse - ex*rhss);
yi = dtinv * (delyi*rhse - ey*rhss);

/*  test for intersection between surface endpoints*/
ssq  = (xi - x1s)*(xi - x1s) + (xi - x2s)*(xi - x2s)
  + (yi - y1s)*(yi - y1s) + (yi - y2s)*(yi - y2s);
if(ssq <= sqlns) {
  intersect_side[s] = 1;  /* s is the intersected side */
  else intersect_side[s] = 0; /* break here in the software version */
}
```

Fig. 3. Radiative Heat Transfer code implemented on the FPGA

In addition, this inner **for** loop is the most computationally intensive portion of the program. Using a timer described in Section 5.1, with **N**=5000 and **m**=37, we found that a Pentium IV Xeon 3 GHz workstation spends 86% of the algorithm time executing the inner **for** loop. The C code inside this loop is included in Figure 3. All the variables used in the arithmetic computations are floating-point.

Originally the program was written for double precision floating-point. In this work, we evaluate single and double precision floating-point. We found that there is not a significant difference in the scientific results from the algorithm when using single versus double precision. The number of photons absorbed differed by only .0025% in the single precision version as compared to the double precision version.

4 Hardware Implementation

We target the hardware implementation to the Virtex II and Virtex II Pro FPGAs. These devices have small embedded memories called Block RAMs as well as embedded 18-bit multipliers. An initial approximation of the pipeline was generated from the Streams-C compiler [15] on an integer version of the code. The generated pipeline was then converted to use floating point modules, and manually optimized to maximize pipelining.

Figure 3 shows the C code for the compute-intensive **for** loop of the radiative heat transfer algorithm. In each iteration of this loop, the calculations are performed relative to one of the surfaces of the convex shape. Some variables are invariant across loop iterations (e.g., **epsdet0**) while others assume unique values for each loop iteration, as shown by the array index **s**, for example, **delxs**, **delys**, and **rhss**. The latter variables are assumed to reside in Block RAMs.

Figure 4 shows the pipelined hardware implementation of the loop. The design is an 11 stage pipeline utilizing the floating point libraries from [13]. It consists of 12 multiply, 3 addition, 7 subtraction, 1 divide and 2 comparison

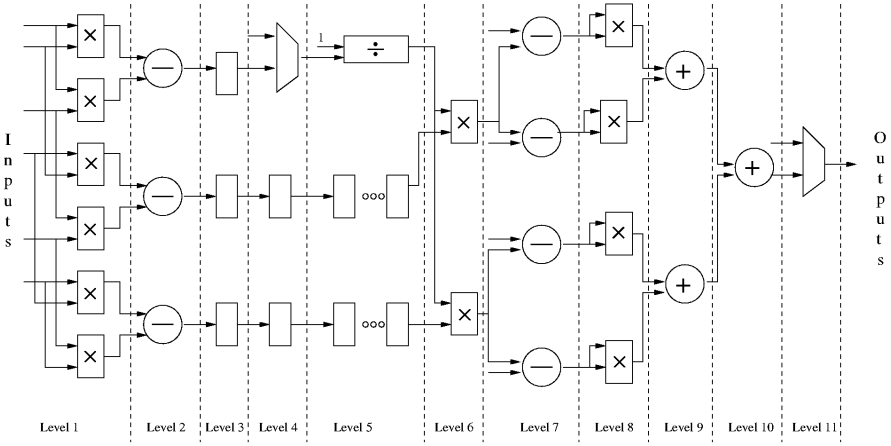


Fig. 4. FPGA Implementation

modules. The breakdown of the latency is as follows: 4 cycles for multiplication, 3 cycles for addition or subtraction, 15 cycles for division, and 1 cycle for comparison. The total latency of the 11 stage pipeline is 41 cycles. There are 2 intermediate registers that need pipelining from Level 4 through Level 5. This data synchronization requirement introduces 32 additional 34-bit registers into the design.[1] For clarity, only two registers are shown in in the Figure 4 in Level 5, but there are 15 registers for each operand, for a total of thirty 34-bit registers at Level 5.

For this implementation there are eleven inputs to the pipeline – six inputs are consumed in Level 1, four at Level 7 and one at Level 10. The data is stored in two 204-bit by 512 deep, dual-port Block RAMs. Memory reads are scheduled so that values arrive at Level 7 and at Level 10 at exactly the cycle they are consumed. By scheduling the reads in this way, we avoid the overhead of fully pipelining the 5 inputs that are needed at Level 7 and Level 10. The latter approach introduces an extra 27 cycles × 4 registers (Level 7) plus 40 cycles × 1 register (Level 10), or $112 + 40 = 152$ 34-bit registers into the design. These 152 registers correlate to a 1% increase in area utilization on the Virtex II.

5 Performance

This section analyzes the performance of the application running on several Pentium IV Xeon (P4) systems versus the Virtex II (V2-6k) and Virtex II Pro (V2p100,V2p125)[2] hardware platforms. Note that the V2p125 is not yet available.

[1] The FPLibrary adds a 2-bit tag to each floating point register.

[2] The V2-6k is speed grade –4 and the V2p100 and V2p124 are speed grade –6.

5.1 Workstation Performance

For performance comparisons with the FPGA we examined the innermost loop of the application, which is the iteration over **m** surfaces for a single photon, searching for an intersection. The static instruction count of this loop count is 130 instructions: 61 floating point instructions, 9 branches, 73 instructions which reference the stack (including floating load/store to stack for locals), and only one integer instruction (the loop counter). All the instructions and data for this loop fit into the Level 1 cache (the fastest cache level), and hence could be expected to run at maximum speed on the CPU.

Timing measurements of the inner loop are easily perturbed due to the small instruction count of 130 instructions. Obtaining an accurate measure of this loop represents a challenge, since traditional profiling tools such as gprof are only acceptable for function-level timing, and do not provide an extremely accurate measure of the inner loop. However, on the Pentium and later processors there is a timer register, called the Time Stamp Counter (TSC), which measures processor ticks at the processor clock rate. This 64-bit read-only counter is extremely accurate, as it is implemented as a Model-Specific Register inside the CPU. The overhead of using this register is extremely low. On a 1.7 GHz P4 the TSC runs at 1.68 GHz and has a resolution of 595 picoseconds; on a 3 GHz P4 the TSC has a resolution of 333 picoseconds.

We used the TSC to measure the inner loop of the application. C code was added using the gcc asm statement, which produces inline assembly code to read the TSC at the start and end of the loop code. We performed measurements both in the application itself, and by extracting the inner loop and running it many times. As expected for this loop, the performance varied with the CPU being used, with the fastest CPU being the 3 GHz P4.

We tested both the Intel compiler v7.0 and gcc v3.2. The gcc compiler provided the best performance results with –O3 optimization level. Timing for one iteration of the inner loop, shown in Figure 5, ranges from 60ns to 104ns. It is important to note that the time is an average, since in the sequential version of the loop body, there is opportunity for early exit from the loop.

5.2 FPGA Performance

Synplicity was used to synthesize the inner loop to Xilinx FPGAs. Placement and area results were obtained using Xilinx ISE 6.1. The results for one iteration of the inner loop on the Virtex II and Virtex II Pro FPGAs are shown in Figure 5. On the V2-6k, only 20% of the Look Up Tables (LUT) are used by the loop body. However, all the multipliers are used, and therefore only one instance of the loop body can fit on this part. In contrast, the larger Virtex II Pro parts can fit three pipelines of the inner loop, resulting in a higher degree of spatial parallelism. The speed up row calculates the speed up relative to the 3 GHz P4. The hardware calculation assumes a steady state pipeline in which a result is delivered every clock cycle. With three pipelines three results are delivered every clock cycle, effectively reduce the execution time by one third. These results do not include the time to write the parameters into Block RAM.

	V2-6k	V2p100			V2p125			P4 1.7	P4 2.4	P4 3
# Pipelines	1	1	2	3	1	2	3			
Execution Time (ns)	29.9	16.7	7.89	5.78	15.7	7.72	6.12	104	74	60
%Area (LUTs)	20	15	33	50	12	26	40			
%Multiplers	100	32	64	97	25	51	77			
Latency (cycles)	41	41	41	41	41	41	41			
Speed up	2.01	3.59	7.61	**10.37**	3.82	7.77	9.81	0.58	0.81	1

Fig. 5. FPGA vs. Workstation performance for Inner Loop. Speed up compared to the P4 3 GHz System.

In terms of technology generation, the V2-6k and P4 1.7 GHz are comparable. The V2-6k hardware implementation outperforms the 1.7 GHz Pentium by a factor of 3.48. For the newer generations of FPGA and microprocessor (V2p100 and 3 GHz), the single pipeline speed up is slightly better – 3.59×. However, with this newer generation Virtex Pro it is possible to fit three pipelines on the V2p100 which allows a speed up of 10.37.

As noted above, the hardware design is highly pipelined. The pipelining allows a relatively high clock frequency for the design, at the cost of high latency – 41 clock cycles before the first result appears. For a large number of surfaces, the effect of pipeline latency diminishes. For example, with 10,000 surfaces the speedup for three pipelines is 10.25×. 150,000 surfaces are desirable for this particular simulation, so the pipeline latency effect is negligible.

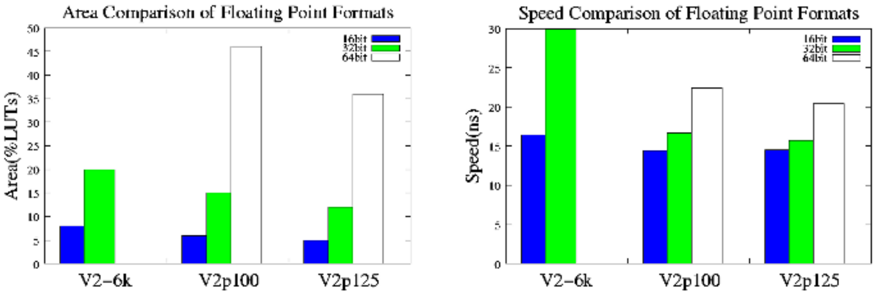


Fig. 6. Placement results for a single pipeline 16, 32, and 64-bit implementations of the inner loop.

Lastly, if we analyze the granularity of the input data width as shown in Figure 6, the placement results show that for a 16-bit word width, the area utilization across the Virtex Family is 5% to 8% which allows 10 to 20 instantiations of the inner loop to run concurrently on the FPGA. For larger bit widths, fewer parallel versions of the loop can fit onto hardware, for example with 32-bits 3 pipelines fit. As expected, the run-time clock speeds are faster for smaller bit widths. The results show that on the Virtex II Pro family, 32-bit operations are only slightly more expensive than 16-bit, while 64-bit incur a much higher penalty both in area and clock speed. As the graphs show, the 64-bit version of the application does not fit on the V2-6k.

5.3 Discussion

Our results show that the FPGA hardware outperforms a comparable generation of microprocessor by up to 10.37× on an application-specific single-precision floating point pipeline. There are several points to note.

First, the FPGA implementation must execute all loop iterations of the inner **for** loop. The software timing is an average number: many times the software breaks out of the loop without completing all iterations, as the last **if** statement of Figure 3 contains a **break** in the software version of the loop. If all loop iterations were executed, the FPGA speed up would be much greater.

Second, this application fits well in the L1 cache of the microprocessor. A more data-intensive application would better use the strengths of the FPGA (greater memory bandwidth and better performance on data-intensive computation).

Third, the tractability of an application kernel to pipelining, especially long pipelines, is crucial to get performance. The highest performance floating point operators are heavily pipelined, so there is substantial cost in starting up and breaking up the pipeline. Like vector processors, the application-specific pipeline on the FPGA shows the best performance when the algorithm has many iterations with minimal data-dependent branching. In this application, the vector length is very large, and thus the latency is negligible. This application also has the advantage of little data-dependent branching. Although predication can be used to reduce the impact of branching, area costs increase by having both the **then** and **else** bodies instantiated on the chip.

Fourth, the floating point library we used in this experiment is technology-independent. In fact, we were able to synthesize it to several different families, including the Altera Stratix. Technology-specific floating point cores such as Quixilica yield smaller area and faster clock rate. On the minus side, other floating point libraries, including Quixilica, have even higher operation latencies. For best performance, embedded hard floating point units in a fabric of reconfigurable logic would, of course, be desirable.

Finally, it is important to compare the performance of the application-specific pipeline, with a mix of different floating point operators and branching constructs, to peak performance results cited by others. While theoretical peak numbers are useful to gauge feasibility, a floating point intensive supercomputing application gives us more accurate performance results.

6 Conclusions

We have presented hardware implementation of a floating point Monte Carlo radiative heat transfer simulation application on the Virtex II and Virtex II Pro families of FPGA. In contrast to previous work that presented peak performance or performance results on small kernels, we have implemented an application-specific pipeline on the FPGA. We have presented detailed timing results comparing FPGA speed to high performance workstations, realizing a 10.37× speed up with three single precision floating point pipelines running on a Virtex II Pro hardware platform versus running the application on a 3 GHz workstation.

Acknowledgments. Thanks to Jérémie Detrey for his open source floating point library and his help with simulation and synthesis of the library modules, and to Sung-Eun Choi for her assistance in obtaining workstation timings.

References

1. K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA 2004)*, 2004.
2. Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003), *Area and Power Performance Analysis of Floating-point based Application on FPGAs*, (Lexington, MA), September 2003.
3. S. Choi and V. Prasanna, "Time and energy efficient matrix factorization using fpgas," in *FPL 03: 13th International Conference on Field Programmable Logic and Applications*, Sept. 2003.
4. Top 500, "Top 500 supercomputer sites." http://www.top500.org, 2004.
5. P. J. Burns and D. V. Pryor, "Vector and parallel monte carlo radiative heat transfer simulation," *Numerical Heat Transfer*, vol. 16, 1989.
6. N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic of FPGA based custom computing machines," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), pp. 155–162, IEEE Computer Society Press, 1995.
7. Walter B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on fpgas," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), pp. 206–215, IEEE Computer Society Press, April 1998.
8. P. Belanović and M. Leeser, "A library of parameterized floating-point modules and their use," in *FPL 2002: The 12th International Conference on Field-Programmable Logic and Applications*, pp. 657–666, Springer-Verlag, 2002.
9. K. R. Nichols, M. A. Moussa, and S. M. Areibi, "Feasibility of floating-point arithmetic in FPGA based artificial neural networks," CAINE02, Nov. 2002.
10. E. Roesler and B. Nelson, "Novel optimizations for hardware floating-point units," in *FPL 2002: The 12th International Conference on Field-Programmable Logic and Applications*, pp. 637–646, Springer-Verlag, 2002.
11. QinetiQ Holdings Ltd., "Real time systems lab." http://www.quixilica.com/ products.htm, 2002.
12. Nallatech, "Floating point IP cores for virtex-II." http: //www.nallatech.com/ solutions/products/ software_fpga_ip/fpga_ip/fpc/, 2003.
13. J. Detrey and F. de Dinechin, "FPLibrary, a VHDL library of parametrisable floating-point and LNS operators for FPGA." http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/, 2004.
14. R. Minnich and D. V. Pryor, "A radiative heat transfer simulation on a SPARC-Station farm," in *First International Symposium on High Performance Distributed Computing (HPDC '92)*, 1992.
15. M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa, CA), 2000.

Stochastic Simulation for Biochemical Reactions on FPGA

M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano

Dept. of Computer Science, Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama #223-8522, JAPAN
bio@am.ics.keio.ac.jp

Abstract. Biological cell simulations generally require high-powered computer resources. A reconfigurable system is a possible solution to the problem as an alternative approach against PC/WS clusters. A stochastic simulation algorithm proposed by Gillespie is implemented on a reconfigurable platform called ReCSiP, and the performance is evaluated. The implemented Lotka system outperforms the software implementation on AthlonXP2800+ by 105.13 times.

1 Introduction

A number of sophisticated methods of biological experiments were developed in '90s, and they enabled quantitative modeling of cellular systems with massive amount of experimental data. Such modeling and simulation on computers are now indispensable to understand the cellular processes in detail. Some software simulators including The Virtual Cell[1], E-Cell[2] have been already utilized by researchers, and their execution time becomes a problem even by using recent high speed PCs. For example, 2 seconds simulation of a nerve cell on The Virtual Cell required 2 days using a workstation with MIPS R8000[1]. To address this problem, Bio-grid project[3] has been developed to use a huge power of parallel processing through the network. As more economical solution, parallel processing on PC/WS clusters have been widely used.

Unfortunately, problems in biological researches often generate a lot of fine grain processes frequently communicating with each other. Such problems are difficult to be treated with network-based parallel processing used in PC/WS clusters or Bio-grid. Instead, reconfigurable systems[4] which execute the algorithm directly on programmable logic devices are suitable for such problems, since they can make the best use of the inherent fine grain parallelism. The flexibility of reconfigurable systems is also useful for biological application which will be changed depending on the research results.

We developed a reconfigurable accelerator called "ReCSiP" for biocomputing, and ODEs(Ordinary Differential Equations)-based simulation has been already implemented on it[5].

In this paper, the design and evaluation of another way of FPGA-based simulator which is based on Gillespie's stochastic simulation algorithm is shown. By making the best use of fine grain parallelism, the implemented module for the Lotka system achieved 105.13 times performance compared with the software executed on AthlonXP 2800+.

The proposed simulator can be easily extended for other reaction model, since the common simulation part and reaction system specific part are separately implemented.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 105–114, 2004.

2 Overview of ReCSiP

The major goal of ReCSiP is acceleration of metabolic simulations to reasonable responsive time for researchers. It is designed to achieve high-throughput computation by the co-operation of the host CPU and reconfigurable platform containing FPGA. ReCSiP consists of the hardware (ReCSiP board), and the software (driver and API). The hardware layer of the ReCSiP includes an FPGA, four SSRAM sets (which can be accessed simultaneously), an SDRAM set, and a PCI interface. It can be inserted into the PCI bus slot on common PCs. Users can easily build up their own acceleration modules coded in HDL for their own simulation tasks. Figure 1 shows the hardware part of the ReCSiP. On the other hand, the software layer provides the device driver and the API. The API can be easily customized to help the development of hardware-accelerated simulators.

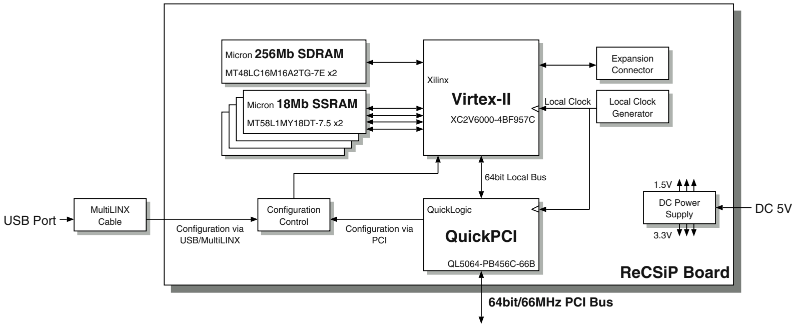


Fig. 1. Concept of ReCSiP

3 Gillespie's Algorithm

3.1 Features of the Algorithm

There are two approaches in mathematical modeling of metabolic reaction systems: deterministic and stochastic. The deterministic approach describes the time evolution of chemical systems in the form of ODEs. Now, many software biological simulators with deterministic approach have been researched and developed. Although various useful knowledge has been obtained by this approach, it has a problem of parameter sensitivity. That is, the equations become stiff according to their parameters, and the solution would not be stable.

The stochastic approach regards the time evolution of the target system as a kind of random-walk process which is governed by a single differential-difference equation. It is difficult to systematically solve the equation, but there is a solution to analyze the behavior of a system without dealing the equation directly by using random numbers.

Gillespie's algorithm[6] is a well-known stochastic simulation algorithm for cellular processes. Since it is a kind of Monte-Carlo simulation, simulation results of the same target will be different by each trial. By running the simulation a number of times, the average of the results will be close to the actual system behavior. That is, the stochastic algorithm is stable to stiff equations, and does not too much sensitive to parameters.

The problem of this approach is computation time. STOCKS[7] is the software simulator using Gillespie's algorithm. At worst, the execution time of the behavior of prokaryotic gene expression in 2,100 seconds, is 22 hours (executed on PentiumIII 800[MHz]). With increasing the complexity of the target system structure and the behavior, more time will be required for simulation.

3.2 Summary of the Algorithm

Gillespie's algorithm is consisting of the iteration of following steps.

1. Calculate τ; time to the next reaction
2. and, μ; type of the next reaction
3. Increase or decrease the molecule numbers by the determined μ

The pair of two values (τ and μ) is calculated from the molecular numbers in the current system, and the molecular numbers in the next iteration are determined by the output of the previous reaction and two random numbers.

τ is the time between the present and the proceeding reaction. τ is a value obtained by multiplying the inverse of the sum of all the reaction possibilities which are in the target system by logarithmic-distributed random number which is derived from a uniform random number r_1 $(0 < r_1 \leq 1)$.

The other variable, μ denotes the next reaction, which is determined by selecting one reaction from all the occurable reactions in the current conditions of the target system. The selection is done by using a uniform random number, r_2 $(0 < r_2 \leq 1)$.

τ and μ are calculated with Equation (1) and (2).

$$\tau = \frac{1}{a_0} \ln \left(\frac{1}{r_1} \right) \tag{1}$$

$$\sum_{\nu=1}^{\mu-1} a_\nu < r_2 a_0 \leqslant \sum_{\nu=1}^{\mu} a_\nu \tag{2}$$

a_ν in Equation (2) is the reaction probability of the corresponding reaction R_μ in the target system. a_ν is a value multiplied combination of molecule numbers related to R_ν by the stochastic reaction rate constant c_ν. c_ν is related to rate constant k_ν, which forms the basis for the deterministic approach to chemical kinetics. c_ν is a value obtained by dividing k_ν by V (volume of target system). If V is not changed during the simulation, c_ν is constant. a_0 implies the sum of all the a_ν.

3.3 Lotka System

The Lotka system is a mathematical model of the predator-prey ecosystem. The following system is an example of simulation of the Lotka system, described in Gillespie's paper[6].

It has 4 species of molecule and 4 reactions among them. Each molecule number is represented with $X_1, \cdots, X_4$, and the reactions are represented with $R_1, \cdots R_4$, as follows:

$$R_1 : \quad X_2 + 1\ ,\ (\bar{X_1} - 1) \tag{3}$$

$$R_2 : \quad X_2 - 1\ ,\ X_3 + 1 \tag{4}$$

$$R_3 : \quad X_3 - 1\ ,\ X_4 + 1 \tag{5}$$

$$R_4 : \quad X_2 - 1\ ,\ X_4 + 1 \tag{6}$$

The Lotka system is well suited for evaluation of Gillespie's algorithm implemented on FPGA. Reaction R_1 is the reproduction of specie X_2 by consuming X_1. In the other words, R_1 implies that X_2 feeds on the foodstuff X_1 and breeds. Reaction R_2 represents that the specie X_3 feeds on the specie X_2 and breeds. R_3 and R_4 are isomerization, which describe the death of X_2 and X_3. The bar over X_1 means that the number is assumed to be constant. In this system, reproduction of X_1 is not performed, but it does not decrease, since it is assumed to be a constant, as in Gillespie's work. By nature, X_1 and X_4 are regarded as input and output of the system, respectively. Numbers of X_2 and X_3 draw waved trajectory by the time series.

Followings show a cycle of process in the Lotka system. It assumes that initialization has been done.

Step 1. h_ν is stored in a value which is combination of molecule numbers related to R_ν $(\nu = 1, \cdots, 4)$ in the current system

$$h_1 = X_1 \cdot X_2, \quad h_2 = X_1 \cdot X_3, \quad h_3 = X_3, \quad h_4 = X_2$$

Step 2. a_ν and a_0 are substituted with multiplication of h_ν and c_ν, and the sum of a_ν

$$a_\nu = h_\nu \cdot c_\nu \ , \qquad a_0 = \textstyle\sum_{i=1}^{4} a_i$$

Step 3. New random numbers r_1 and r_2 are generated. And $1/\tau$ is calculated with multiplication of $1/\ln(1/r_1)$ and a_0. μ is determined by comparing a_ν with $r_2 a_0$

$$\tfrac{1}{\tau} = \tfrac{a_0}{1/\ln(1/r_1)} \qquad \textstyle\sum_{\nu=1}^{\mu-1} a_\nu < r_2 a_0 \leqslant \sum_{\nu=1}^{\mu} a_\nu$$

Step 4. $X_1, \cdots, X_4$ in step 1 are modified by R_μ

Step 5. Modified molecule numbers $X_1', \cdots, X_4'$ and τ are generated. $X_1', \cdots, X_4'$ are referred by the next cycle of step 1

Various kinds of chemical reactions can be modeled like equations (3), (4), (5), and (6), to simulate other systems.

4 Implementation

4.1 Overview of the Simulator

In this section, design and implementation of the Lotka system with the stochastic simulator on ReCSiP are introduced.

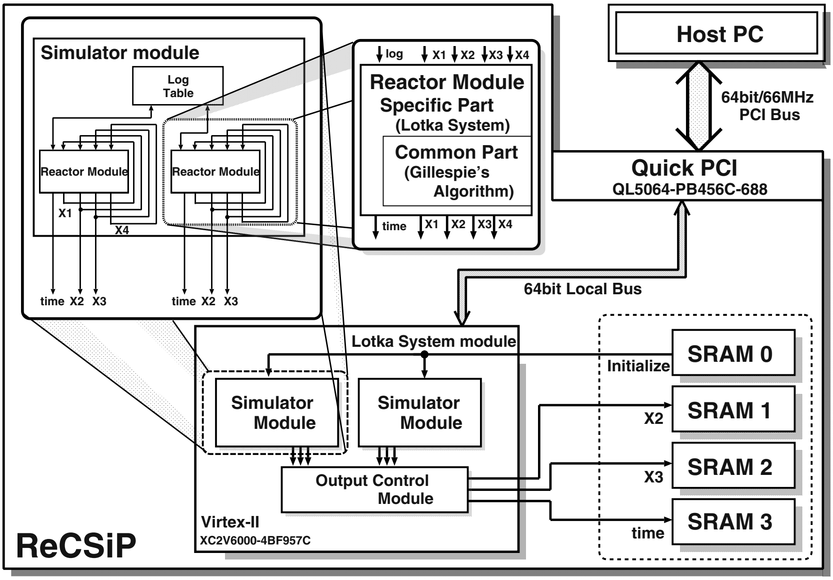


Fig. 2. Structure of the Lotka System Module

The reactor module, which performs step 1, 2, 3, and 4 in the previous section, is parallelized and controlled by the other modules as shown in Figure 2. The Lotka system module consists of two simulator modules and an output control module. Each simulator module has two reactor modules as the core of simulator.

A simulator module starts calculation when it receives the random seed and initial value of molecule numbers stored in the SRAM module. The results from 2 simulator modules (4 reactor modules) are stored into the SRAM modules through the output control module.

4.2 Simulator Module and Reactor Module

Each simulator module simply consists of two reactor modules, which share a logarithmic table.

A reactor module processes a cycle of the Lotka system with Gillespie's algorithm. In the first step of the process, the module receives the molecule numbers, $X_1, \cdots, X_4$. Then, it processes a cycle of the Lotka system with Gillespie's algorithm. Finally, it outputs τ and molecule numbers $X'_1, \cdots X'_4$ after the reaction. Molecule numbers $X'_1, \cdots X'_4$ are used in the next input of reactor module. As X'_2, X'_3 and τ are required for the evaluation of the simulation result, they are transferred to the output control module.

The reactor module consists of two parts; the common part for Gellespie's algorithm, and the specific part for the Lotka system. By separating them, the common part can be used for the other reactor modules. That is, by replacing the reactor module in Verilog-HDL code level, other reactions can be simulated.

The common part calculates τ and μ, and also generates random numbers. The target-specific part processes the combinations of reactions, then increases or decreases numbers of molecules according to the result of calculated μ.

A reactor module has 4 kinds of major functional units, which are two 32bit integer multipliers, five Single-precision FP adders, six Single-precision multipliers, and a Single-precision FP divider.

- Single-precision FP multiplier includes 18 × 18 bit dedicated multiplier blocks (distinct features of Virtex-II)
- Single-precision FP divider using 2bit-base subtract-and-shift divider
- 32 bit LFSRs (Linear Feedback Shift Registers) for the random number generation with M-sequences

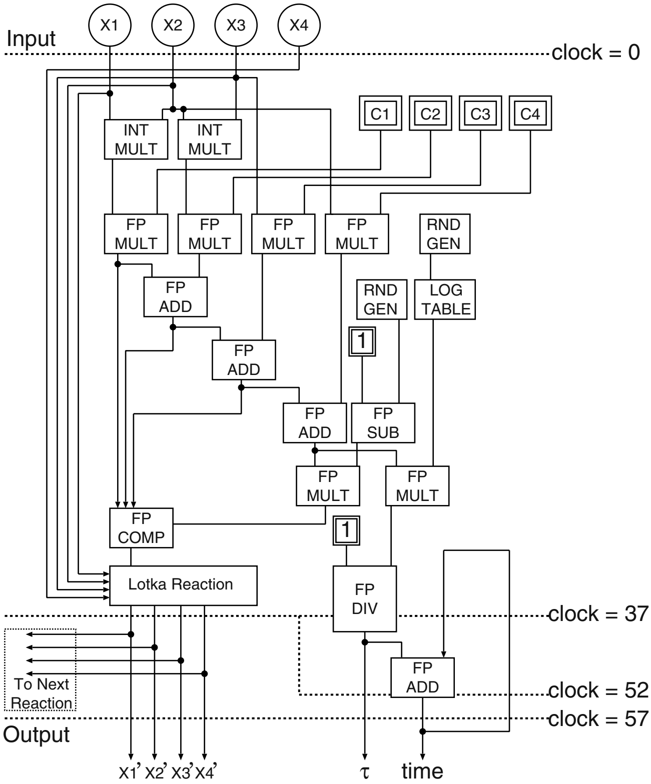


Fig. 3. Data-flows in Reactor Module

Figure 3 shows the flow of calculation in the implemented reactor module. The reactor module has 37 pipeline stages. It takes $X_1, \cdots, X_4$ as inputs, then outputs molecule numbers $X'_1, \cdots X'_4$ and τ after a reaction through step 1, 2, 3, and 4 described before. The output takes 37 clocks for $X'_1, \cdots X'_4$, and 52 clocks for τ. So, it is possible to execute 37 independent simulation processes at a time. These multiple concurrent executions are advantageous in this kind of stochastic simulation which returns an average of the iterative executions.

Values of $1/\ln(1/r_1)$, which are required to calculate τ, are stored in 32bit width tables whose depth are 2^{15}. The tables are implemented on Block RAM on Virtex-II. τ is derived with multiplication of the fetched number and a_0 (the sum of the reaction probability of each reaction a_ν).

The reactor module has 32bit-width 37 entries shift registers which store the total simulation times t. Each register updates value by adding previous t and τ at the end of a cycle.

4.3 Output Control Module

The Lotka system module has two simulation modules each of which has two reactor modules, and in a reactor module, 37 simulation processes are executable in parallel. Therefore, maximum 148 simulation processes are "on-the-fly" in the whole the Lotka system module. Total simulation time t, number of prey species X_2, and number of predator species X_3 are generated with the simulation of the Lotka system. These values are represented 32bit integer or single-precision floating-point. As a result, four sets of three 32bit data is generated from the Lotka system module with each cycle.

Output control module manages data transfer from the simulator module to SRAM modules. The module works as follows.

- A set of three 32 bit data is transferred from reactor modules to SRAM modules with an arbitrary cycle interval
- FIFO buffer for temporary storage of output data is provided. It begins to store data from the reactor module in the specified cycle

Inputs of the output control module are four sets of three 32 bit data which are t, X_2 and X_3 per clock. The module transfers them to SRAM modules with an arbitrary interval (that is, interval must be nothing less than 5). Figure 4 shows the structure of the output control module.

5 Evaluation

5.1 Result of Synthesis and Place and Route

Table 1 and 2 show required resources and maximum operation frequency for the Lotka system module.

Modules are described in Verilog-HDL. Synthesis and place & route are done with Xilinx ISE6.1i. The target device is Xilinx's XC2V6000-4BF957C, which is equipped on the ReCSiP board.

In order to optimize the clock frequency, this implementation is somehow specialized to the Lotka system. So, it can not be extended larger scale systems without modifying the Lotka system directory in the current implementation.

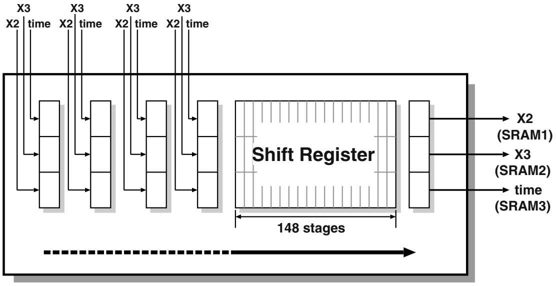


Fig. 4. Structure of Output Control Module

Table 1. Resource Utilization

Slices	18x18 Multipliers	18kbit BlockRAM
26091 (77.21%)	120 (83.33%)	132 (91.67%)

Table 2. Performance

Frequency [MHz]	Throughput [cycle/sec]
76.25	304.88M

Some simulation results are shown in Figure 5 and 6. These are results after 500,000 cycles of executions with 10 output intervals. The difference between them is caused only by the random seeds, and the conclusive simulation result of the Lotka system is obtained with taking an average of them by appropriate time interval.

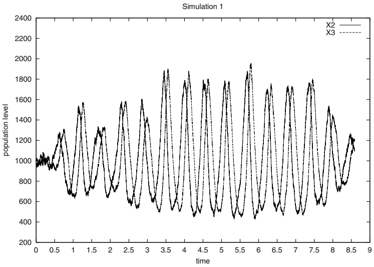


Fig. 5. Example of Result

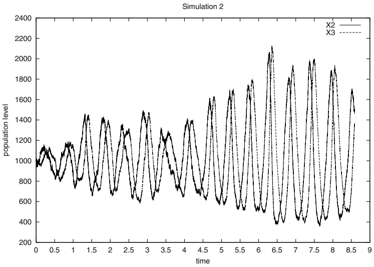


Fig. 6. Example of Another Result

5.2 Accuracy Verification

The accuracy and performance of the simulator on ReCSiP is compared with a software simulator on common PCs.

Since the floating-point number arithmetic units on ReCSiP are not based on rounding algorithm in the IEEE standard, its influence must be examined. In calculation of τ, there is no error propagation because the output isn't used as the input of the next cycle. However, the result of μ is used to determine what the next reaction occurs. In this case, the accumulation of rounding errors may cause a problem.

To examine the effect of rounding errors, the same simulator including an M-sequence generator written in C is executed with the same random seed. Figure 7 and 8 show the result of software execution and result from ReCSiP board. They are measured after 100,000 cycles of execution, and the output interval is 10. Trajectories of molecule number X_2 and X_3 are similar both in software and in hardware execution. This shows that useful simulation results can be obtained from the hardware execution on ReCSiP board.

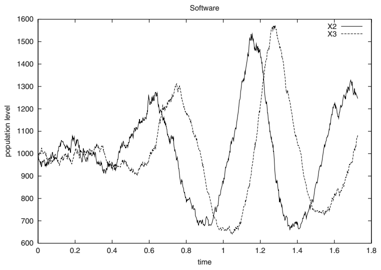


Fig. 7. Exec by C Code

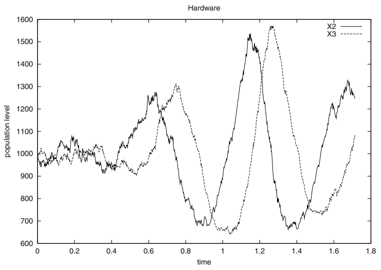


Fig. 8. Exec on Hardware

5.3 Performance Evaluation

Table 3 shows response times and performance for 500,000 cycles of the software simulation of the Lotka system. In the software execution, the program is compiled with -O3 option. Throughput S is derived from the equation $S = 0.5[\text{Mcycle}]/\text{run-time }[\mu\text{sec}]$.

Table 3. Throughput of the Software

Processor	Memory	Environment	Run-time [μsec]	Throughput [cycle/sec]
AthlonXP2800+ 2.08GHz	2GB	Free BSD 4.8 +gcc2.95.3	172,226	2.90M
Xeon 2.8GHz Dual(HT off)	4GB	Linux2.4.21 +gcc2.95.3	214,219	2.33M
UltraSPARCIIIcu 1.2GHz	4GB	Solaris8 +gcc2.95.3	555,907	0.90M

The implemented Lotka system simulator can generate a result at intervals of 37 clocks, and the maximum operation frequency is 76.25 MHz. Accordingly, by equation 76.25 [MHz]/37 = 2.06 [Mcycle/sec], the throughput of the single simulator is 2.06[Mcycle/sec].

The implemented reactor module is 37-stage pipelined, and four reactor modules can be mounted (on a Lotka system module). Thus, this simulator is able to execute 148 simulations simultaneously in a clock. As the result, the maximum throughput of the Lotka system simulator is 304.88 [Mcycle/sec]. It is about 105 times faster than the software implementation on AthlonXP 2800+ (operating at 2.08GHz).

ReCSiP is connected to host PC with 64bit/66MHz PCI bus. If the Lotka system module transfers to SRAM modules by 10 output intervals, total throughput of output data is 365.856 [MByte/sec]. That is, each simulator outputs three 32bit data, X_2, X_3, and t in every cycle, and 148 simulations are executing at a cycle. The performance of the Lotka system module is 304.88[Mcycle/sec]. If the output interval is assumed to be 10, data throughput becomes 365.856 [MByte/sec]. 64bit/66MHz PCI bus has the enough bandwidth to transfer the amount of data derived above.

6 Conclusion

A stochastic cell simulator is implemented on ReCSiP, a reconfigurable platform for bioinformatics. A hardware simulator of the Lotka system with Gillespie exemplifying is implemented and evaluated. This module can execute simulations 105.13 times faster than that of software implementation on the AthlonXP2800+.

Acknowledgement. This work was supported by the MEXT, the Ministry of Education, Culture, Sports, Science and Technology of the Japanese Government (Special Coordination Funds for Promoting Science and Technology).

References

1. J. Schaff et al. A General Computational Framework for Modeling Cellular Struc ture and Function. *Biophysical Journal*, 73:1135–1146, Sep. 1997.
2. M. Tomita et al. E-Cell: software environment for whole-cell simulation. *Bioinformatics*, 15(1):72–84, Jan. 1999.
3. H.Matsuda. Development of Bio-information Environment of the Grid. In *Grobus World*, Jan. 2003.
4. W.M.Smith et al. Seeking Solutions in Configurable Computing. *IEEE Computer*, 30(12), Dec. 1997.
5. Y.Osana et al. Implementation of ReCSiP: a Reconfigurable Cell Simulation Platform. In *The 13th International Conference on Field Programmable Logic and Applications*, volume 2778 of *Lecture Notes in Computer Science*, pages 766–775. Springer, Sep. 2003.
6. D. T. Gillespie et al. Exact Stochastic Simulation of Coupled Chemical Reactions. *The Journal of Physical Chemistry*, 81(25):2340–2461, Dec. 1977.
7. A. M. Kierzek. STOCKS: STOChastic Kinetic Simulations of biochemical systems with Gillespie algorithm. *Bioinformatics*, 18(3):470–481, Mar. 2002.

Dynamic Adaptive Runtime Routing Techniques in Multigrain Reconfigurable Hardware Architectures

Alexander Thomas and Juergen Becker

Institut für Technik der Informationsverarbeitung (ITIV), Universität Karlsruhe,
76128 Karlsruhe, Germany
{thomas, becker}@itiv.uni-karlsruhe.de
http://www.itiv.uni-karlsruhe.de

Abstract. Modern application scenarios out of the multimedia and mobile communication domains demand more and more performant data processing architectures, which cannot be reached by using actual DSP or microprocessor approaches. This contribution describes a new architecture approach out of the reconfigurable array field which offers a set of new features to increase the flexibility and usability of reconfigurable array architectures by increasing the performance benefit concurrently. The main focus of this publication is the communication topology where the authors will discuss the concepts in detail.

1 Introduction

The increasing complexity and performance requirements of actual applications e.g. within multimedia and mobile communication domains with their control and data oriented characteristics needs more and more sophisticated architecture solutions for providing efficient and rapid application execution. By far the most systems today consist of a set of DSPs, microprocessors and ASICs, which compose the main data processing core. Beyond this approaches there are only a few exceptions where new concepts are realized [NEC1], [XP01].

The concepts of microprocessors and DSPs base on sequential execution of instructions in combination with load and store procedures of data and instruction words in and out of the memory. By this approach the architecture gets concept specific advantages and disadvantages. Basically, in combination with random access memories sequential execution processors are predestinated for processing of control oriented application. The disadvantages of this kind of approaches lie in the frequently memory accesses during instruction/data reading and writing. Thereby this strategy leads inevitable to a bandwidth bottleneck which results because of a brisk data exchange between the memory and data processing unit. Modern microprocessors reduce this problem by deploying fast clocked caches. By working on streaming data the caches loose their potency and the throughput of the data processing unit converges to maximum memory throughput, which limits the system performance.

Similar to ASIC concepts but more flexible approach follow the concepts of dynamic reconfigurable array architectures. In this architecture domain we differ two

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 115–124, 2004.

architecture types: fine grained architectures [XI01], [AL01] and coarse grained architectures [Kr01], [XP01], [XP02], [Be01], [Be02], [NEC1]. Both architecture types use multiple functional unit implementations in combination with a flexible communication network. This network is used to connect the functional units dynamically during the runtime. The main difference between both architecture types is the data word width granularity of the architectures. Those approaches are able to perform arithmetic data manipulation directly on input data. Whereas the fine grain approaches are working on single bit data by realizing low level Boolean functions. The advantage of this approach is clear: by mapping a function into the area and using the functional units in parallel the architecture reaches similar properties like ASICs. High utilization of functional units, parallel data processing and lowered I/O-bandwidth without the almost needed administration overhead of a processor lead to high performance. At the same time the architecture does not need high frequencies like DSPs and in this connection it is also possible to lower the core voltage which results in high reduction of power consumption.

Naturally dynamic reconfigurable architectures possess several not neglectable disadvantages in comparison to microprocessors or ASICs. Because of structural reasons and mostly easy composition of functional units with their rudimental set of operations till a few exceptions [Be01] it is not easy to perform a control flow based application on such kind of architectures. At this point one more flexible approach is needed, which allows the execution of control flow based application. The first step in this direction personates a flexible communication network, which allows the efficient transportation of control data between the functional units and other system components beyond the array architecture. Therefore this paper introduces in detail a new adaptive multigrain dynamic reconfigurable communication network topology, which offers new ways for dynamic communication within such kind of architectures.

The next chapter gives an overview of the target architecture, the HoneyComb array, which is used to demonstrate the feasibility and performance capability of the new communication approach. The third chapter introduces the new communication topology approach. In chapter 4 an application scenario for using the new architecture is shown. This contribution is finalized with conclusion in chapter 5.

2 HoneyComb: Adaptive Dynamic Reconfigurable Architecture

The HoneyComb (HC) architecture belongs to the class of multi grained reconfigurable array architectures. It offers a lot of new features, which extend the usability of this architecture. With new dynamic functions, like dynamic adaptive routing, intelligent I/O-units and multigrain data paths, the architecture has been enabled to support data flow based applications as well as a set of control flow based applications without loosing the performance benefit caused by great reconfiguration overhead. The main granularity of the HoneyComb is 32 bits. Figure 1 shows the logical and technological structure of the HoneyComb array. The array is based on the hexagonal geometrical shape, wherefrom the architecture name is derived. This structure offers six direct neighbors to each cell. All neighbors are connected through on RTL configurable set of bidirectional links, typical configuration contains six links. The six directions of the hexagonal structure increase the routing overhead, but offer a good tradeoff in respect of reach ability and communication latency between cells. Dependant on the relative position of two cells the latency gain can be up to 50% in comparison to 4 neighbored architectures, where the data needs half of the time as before. The reachability can be increased by raising the number of direct neighbors, but this proceeding is expensive because of fast raising complexity of needed multiplexers.

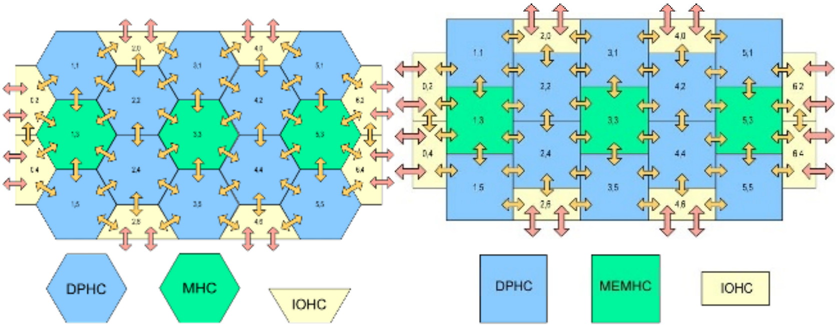


Fig. 1. The Overview of the HoneyComb array: The logical structure (on the left hand side) is based on hexagonal cells with six direct neighbors. The technology mapping changes the structure to quadratic form.

The HoneyComb array consists of three cell types. Each cell contains a routing unit and a specialized module which defines the type of the cell. The routing unit is the controller unit of the communication network, where the sum of all routing units in the array composes the communication network. All neighbor cells are connected through routing units. The first and main type cell in the HoneyComb array is the data processing honey comb (DPHC). This cell is responsible for data manipulations and calculations. The idea behind those data paths is to integrate more intelligent logic into the data paths to enable the data path to respond to control focused application flows. The second cell type of the HoneyComb is the memory honey comb (MEMHC). This cell offers storage functionality within the HC array. The data can be stored in a FIFO, LIFO or linear memory of defined size. Integration of additional rudimental functional units into the MEMHC is also planned. The last cell type of the HoneyComb is the input/output honey comb (IOHC). The main job of this cell is the interfacing of the HoneyComb array to the main on-chip bus. Basically this cell is responsible for transportation of data and configurations into the array and data out of the array. Therefore IOHC possesses AMBA AHB master/slave interfaces on the system side. The clue of the IOHC is the integrated distributed stack pointing and instruction controller (DiSPIC). This controller enables the IOHC to read any data out of the memory in any irregular sequences. The idea behind the DiSPIC was to create small programs and associate those programs with logical data streams which connect the IOHC and any port within the array. The direction of those streams is not relevant, whereby the software developer can create input DiSPIC programs as well as output routines.

The communication network of the HoneyComb array offers support for dynamically routing during runtime in hardware. The architecture provides functionality which undertakes the routing in the array. Beyond this feature it offers multigrain channels for transportation of the below inducted control information or fine grain data for logical calculations. Those fine grain channels can be routed by the architecture as well as the standard 32 bit data channels. The automated routing connects always one dedicated data output port to one dedicated input port. Those ports can be part of the data path within the DPHC, memory module within the MEMHC or DiSPIC within the IOHC. Logically there is no difference between all named ports except the location and address within a specified cell.

3 Adaptive Dynamic Runtime Routing Topology in Hardware

The following sections describe the communication protocol, multigrain hardware links, the special defined instruction set, the adaptive routing and the build-up of the routing unit.

3.1 Communication Protocol and Register Buffering

The network topology of the HoneyComb architecture is based on bidirectional point-to-point links between all routing units within neighbor cells. Each bidirectional link consists of two single opposite unidirectional handshake protocol based connections. The exact protocol diagram is shown in Figure 2. If the transmitter cell wants to send a data word it asserts the RDY signal line at time point t. The receiver cell detects the readiness of the sender and generates - if ready - an asynchronous response within the same time clock t by asserting the ACK signal. This is done by generating the response by integrating the necessary logic in a combinational net. If both control signals RDY and ACK are asserted at a given positive clock edge, both communication partners accept the valid transfer. In this way it is possible to transfer a data word in each clock cycle with data buffering in each cell.

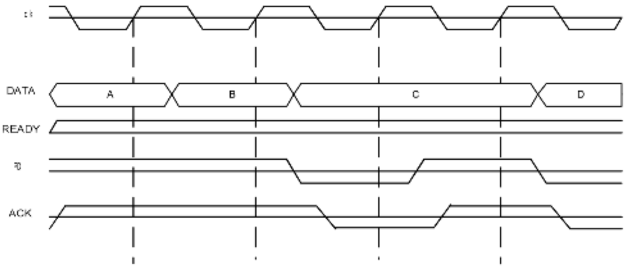


Fig. 2. Waveform .of the handshake protocol used for data transmission

But the described communication protocol causes a critical path. If there is a chain of several communication participants each one with a single register buffer, one single delay at the end of the chain can cause data loss. The problem is the propagation time backward through the chain. Because of asynchronous ACK signal generation the whole chain has to be passed in one cycle. It is surely clear understandable, that this can not work because of dynamic number of the chain length. The remedy at this point is the extension of each participant with double buffers. The strategy is to use in normal operation mode just one single register and fill the second register just in case that the successor participant is not able to receive data. In the next clock cycle this participant blocks the receiving till the point the second buffer register is empty again. The InReg module within the routing unit in each cell of the HoneyComb has to take care of the described problem. Therefore this module consists of two word registers and one control FSM.

3.2 Multigrain Hardware Links

The multigrain communication support in the HoneyComb is realized with two link types. The first type is the coarse grain link, which is fixed to 32 bit for data transmission and can not be splitted in smaller channels. An additional instruction flag is used to tag the data type, which is used to differ between instruction and data words. All instructions are carrying asserted instruction flag. Each coarse grain link contains a RDY signal as well as an ACK signal. An additional ROUTACK signal, which runs from receiver to sender, is implemented as well. It is needed to indicate if the routing is successful or not. Figure 3 shows the protocol signals of two connected communication participants.

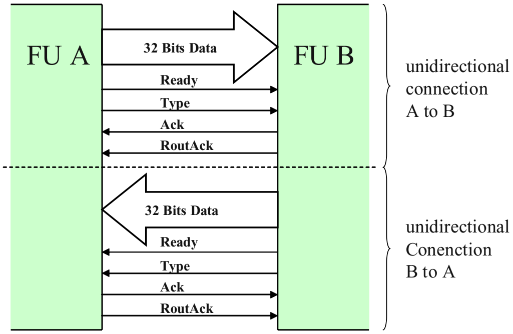


Fig. 3. Signal overview of the bidirectional hardware links between two neighbor cells compose of two separated unidirectional links.

The second link type is the fine grain link (see figure 4). This link has the size of 32 bits but can be splitted into smaller channels. Those channels have the size of 1 bit or 8 bit. The sum of eight 1 bit channels and three 8 bits channels adds up to the whole fine grain link of 32 bits. Each 1 bit or 8 bits channel implies RDY and ACK signals and an additional PATHDELETE signal, which can be used to release this channel. The realization of multigrain streams can now be done by logical combination of a set of this channel to an interrelated stream of desired size. An example transition of an input fine grain link to an output fine grain link is shown in figure 4. Figure 5 shows all involved signal within a fine grain channel.

3.3 Instruction Set

The implementation of adaptive runtime routing is realized by executing an instruction set of a total of seven instructions which is defined for this purpose. The first instruction in this context is the start configuration instruction. This instruction initiates the transmission of new configuration instructions with a header-data field method. The main purpose of the start configuration instruction is the routing, because it is not possible to transmit data on not routed links. Before transmitting any data words on the array it is necessary to build up a logical connection, so called logic stream, between start and target cells. Within this logic stream all involved physical links are fix assigned to this stream and can not be used by other streams. If a logical stream is not longer needed, the involved links can be released and the logical stream is dissolved. For doing this job, the start configuration instruction contains several data fields. Beside the coordinate information of the target cell, this instruction contains information about type of wished logical stream. A priority bit within the instruction control influences the handling of this request in each routing unit. Routing instructions with set priority bit will be handled privileged. Additional a speed path bit can be used to bypass register buffer in long pipeline chains and to reduce the transmission latency. The exactly routing techniques are described in section 3.5.

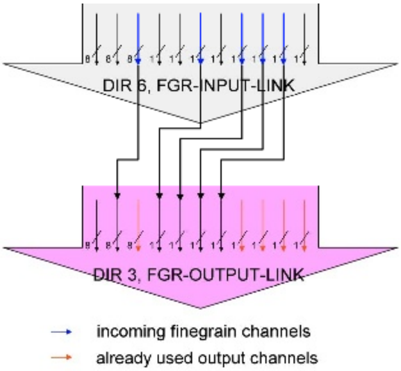


Fig. 4. Routing example of incoming fine grain link with 12 bits data width to an output link, where 12 bits are already occupied.

The next instruction type is the configuration header instruction. This instruction covers the configuration data for the target cell. It contains information about the amount of configuration data which should be stream into the configuration interface of the data path or memory module. After interpretation of this instruction the routing unit transmits the specified amount of followed data to the configuration interface.

The routing instruction is similar to the start configuration instruction. The difference of those instructions is the purpose of the routing instruction to connect two ports within the HoneyComb. Therefore this instruction establishes a logical stream from specified output port to specified input port. The routing works similar to the routing caused by the start configuration instruction.

The special issue of the routing instruction is to route also fine grain logical streams, too. Therefore the routing units just forwards the copy of the routing instruction over the coarse grain links in parallel to the mentioned fine grain logical stream. The stream end instruction releases a routed logical stream. Therefore this instruction should be injected into the data or configuration stream. All passed routing units identify the stream end instruction and delete before written multiplexer control information in the routing memory.

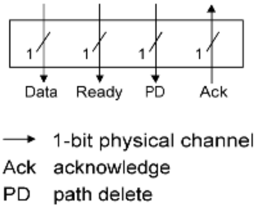


Fig. 5. Involved signal of a single bit physical channel; a set of this channels in addition to three 8 bits channels with same control signals composes the whole fine grain link.

3.4 Routing Unit Structure

The routing unit consists of a set of modules which realizes the above described functions. The first unit is the InReg module. This module contains two registers for data buffering in case if any delay is happening. The InReg is additional controlled by the InCtrl. The InCtrl module detects weather the incoming data is an instruction or not and decides what has to be done next. InCtrl forwards routing requests to the Request Dispatcher. This unit distributes the requests to multiple intermediate registers. Those registers contain all information about actual routing operations which are in progress. The multiple implementations of intermediate registers allow parallel routings at the same time.

The intermediate registers communicate with the routing controller and induces routing operations. The routing controller analyzes the routing memory data, which describes the states of the output link und multiplexer assignments. Based on this information the controller calculates the next routing direction and switches the necessary multiplexers. After this the Routing Controller forwards this information to the intermediate register and waits till the routing operation successes or fails. If the routing successes the routing controller just writes the routing data into the routing memory and forwards this to the InCtrl, which switches to normal transmission mode. If the routing fails, the routing controller checks alternative directions if any available else forwards the information of failing to the InCtrl, which uses the InReg to inform the predecessor cell of failed routing operation.

In case of fine grain routing the strip matcher checks additionally the fitting fine grain output links, which offer enough free capacity to transport the requested granularity. Those links are included by the routing controller into the next direction calculation.

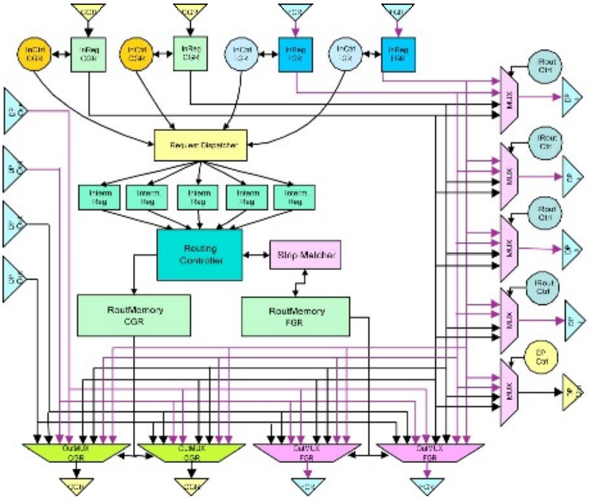


Fig. 6. Structure of the routing unit; left: output ports of the cell internal module like data path; top: input links from neighbor cells; right: input ports to the cell internal module; bottom: output links to neighbor cells.

Configuration instructions induce the InCtrl to rout the following data of specified size to the configuration port of this cell, where e.g. the data path controller interprets the forwarded data. Figure 6 shows the whole routing unit structure.

The IRoutCtrl at the entrance of the data path or memory module has the job to block the local stream end instruction. The global stream end instructions will be propagated through the data path and release all configurations.

3.5 Adaptive Routing Techniques

The adaptive routing technique is used to route the logically stream for data transportation. Therefore the physical links between HoneyComb cells get logic assignment to a specified logic stream. For better understanding how this technique is working exemplary routing process for start configuration instruction is described.

The start cell of the start configuration instruction within the HoneyComb architecture is the IOHC. The DiSPIC in the IOHC transmits the start configuration instruction to the routing unit in the same cell. The routing unit calculates the direction to the target cell and forwards a copy of the instruction to the selected cell and waits till the routing is finished or failed. If the routing was successful the ROUTACK signal will be asserted during the successor cell signals with the ACK signal the absorption of actual applied data. Hereupon the routing unit establishes the logical stream by writing the output multiplexer control data into the routing memory. If the routing fails the routing unit tries to route over the other directions. If those actions fail too, the routing unit signals this nuisance to the DiSPIC which has to handle the problem.

The calculation of the direction is based on the coordinate information within the instruction and the cell position. The following cells on the way to the target cell operate in similar manner till the target cell is reached. The target cell just consumes the start configuration instruction and acknowledges the target achievement to previous cell. From now on the logical stream is established and the configuration can be received by the target cell.

The expected advantages of this new routing method are the saving of needed memory space, smaller mapper tool complexity and fault tolerance. The memory space can be saved because the architecture does not need several geometrical configuration IPs. The mapper software can generate template configurations, which can be used by system controller to adapt the geometrical property of this configuration to the current free available cell constellation in the array. The template should contain informations about the suggestive configuration splitting by providing data about routing load within this configuration for the purpose of calculation minimizing during runtime. Of course, this template should be usable by the HoneyComb array directly.

By making available the hardware routing in runtime the configuration design software like configuration mapper should result in reduced complexity. It is not longer necessary to generate the configuration data for the wiring. The main focus is to map the functions onto the data paths and memory modules. The routing in this context is done by generating the necessary routing instructions. For the purpose of optimization it will be additionally necessary to estimate the routing load within the configuration to prevent high latency signals between HoneyComb cells.

Well, in combination of template configurations and dynamic hardware routing it is possible to prevent the HoneyComb of using faulty cells. If a faulty cell is detected, direct after fabrication or later, however it is possible to deactivate this cell. The deactivation effects that the hardware routing avoids deactivated cells and routes a bypass over the neighbor cells. The deactivated cell can use the implemented link protocol to signalize failed routing. This cause the previous cell just routes over other available directions like described above.

4 Conclusion

This paper introduced the innovative concepts of the dynamic runtime routing in hardware. The resulting technique helps to accelerate the dynamic reconfiguration activity by supporting multiple reconfiguration channels within every configuration. Furthermore, the adaptive routing offers a lot of advantages, like flexible configuration programming, flexible configuration patterns and fault tolerant array usability. The flexible DiSPIC interface composes a flexible control front-end, which enable the HoneyComb architecture to operate without system controller interferences.

The first VHDL-version of this architecture is already developed, but needs more optimization effort. The current area of a DPHC with six coarse grain and 2 find grain links per side and the data path configuration of 4 adders, 1 multiplier, 1 divider and LUTs is about 1,5 mm² by using the Synopsys Design Compiler and UMC 0.13 µm standard cell technology. The ambition at this point is to reach area size of about 0.7-1.0 mm². Reachable clock rates are at about 250 MHz.

Furthermore, some programming support is needed. Therefore the definition of a suitable assembler language is planned. Based on this language, development of an application for demonstration purposes will be developed. The selected demonstration application is H.264, which offers a lot of challenges because of strong control orientation to reconfigurable array architectures.

References

[Kr01] R. Kress: A fast reconfigurable ALU for Xputers; Ph. D. dissertation, Kaiserslautern University, 1996

[Be01] Juergen Becker, Thilo Pionteck, Manfred Glesner: DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications; in: 10th International Conference on Field Programmable Logic and Applications, Villach, Österreich, 2000.

[Be02] Becker, J.; Hartenstein, R. Configware and Morphware going Mainstream In: *Elsevier Journal of Systems Architecture JSA (Special Issue on Reconfigurable Systems)* , Oktober 2003.

[Be03] Becker, J.;Thomas, A.; Vorbach, M.; Baumgarte, V.: An Industrial/Academic Configurable System-on-Chip Project (CSoC): Coarse-grain XPP-/Leon-based Architecture Integration; Kongressbericht Design, Automation and Test in Europe Conference (DATE 2003) , München, März 2003

[Be04] Becker, J.; Thomas, A.; Scheer, M.: Datapath and Compiler Integration of Coarse-grain Reconfigurable XPP-Arrays into Pipelined RISC Processors In: *Kongressbericht Proceedings of IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2003)* , Darmstadt, Germany, Dezember 1-3, 2003

[XP01] PACT XPP Technologies Corporation: http://www.pactcorp.com

[XP02] V. Baumgarte, F. Mayr, A. Nückel, M. Vorbach, M. Weinhardt: PACT XPP Technologies - A Self-Reconfigurable Data Processing Architecture; The 1st Int´l. Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA´01), Las Vegas, NV, June 2001.

[XI01] Xilinx, Inc.; http://www.xilinx.com

[AL01] Altera Corp.: http://www.altera.com

[TR02] Triscend A7 Configurable System-on-Chip Platform - Data Sheet http://www.triscend.com/products/ dsa7csoc_summary.pdf

[AT01] Atmel Corp.: http://www.atmel.com

[QU01] QuickSilver Technology, Inc.; http://www.quicksilvertech.com

[Go01] S. Copen Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer: "PipeRench: a Coprocessor for Streaming Multimedia Acceleration" in ISCA 1999. http://www.ece.cmu.edu/research/piperench/

[Ba01] N. Bagherzadeh, F. J. Kurdahi, H. Singh, G. Lu, M. Lee: "Design and Implementation of the MorphoSys Reconfigurable Computing Processor"; J. of VLSI and Signal Processing-Systems for Signal, Image and Video Technology, 3/ 2000

[Zh01] Hui Zhang, Vandana Prabhu, Varghese George, Marlene Wan, Martin Benes, Arthur Abnous, "A 1V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications", Proc. of ISSCC2000

[NEC1] DRLE, dynamic reconfigurable logic engine, http://www.nec.co.jp/press/en/9902/1502-01.html

Interconnecting Heterogeneous Nodes in an Adaptive Computing Machine

Frederick Furtek, Eugene Hogenauer, and James Scheuermann

QuickSilver Technology, San Jose, CA 95119 USA
fred@calculus.com

Abstract. A distinguishing characteristic of field-programmable logic is the ability to route wires in the field, but previous authors have made compelling arguments for routing *packets*, not wires, between major system components. The present paper outlines the packet-switched network for interconnecting heterogeneous nodes in QuickSilver Technology's Adaptive Computing Machine (ACM). Special attention is paid to two truly innovative aspects of the ACM architecture: (1) the *Point-to-Point* (*PTP*) *protocol* for transferring real-time, streaming data and (2) the *node wrapper* which makes all nodes appear homogeneous regardless of their internal structure or functionality. The wrapper also provides a single, uniform and consistent mechanism for *task management*, *flow control* and *load balancing* across all node types. With the PTP protocol and the node wrapper, nodes as diverse as digital signal processors, reduced-instruction-set processors, domain-specific processors, reconfigurable fabrics, on-chip and off-chip bulk memories and input/output ports can communicate seamlessly. Moreover, once a node (including wrapper) has been configured, or reconfigured, by a supervisory node, it is able to operate autonomously without the need for global control.

1 Introduction

A distinguishing characteristic of field-programmable logic is the ability to route wires in the field, but previous authors have made compelling arguments for routing *packets*, not wires, between major system elements. Seitz[1] and Dally[2] argue that a dedicated packet-switched network offers several advantages in structure, performance and modularity:

- Electrical properties are optimized and well controlled
- Controlled electrical parameters enable aggressive signaling circuits
- Aggressive signaling circuits reduce power and increase propagation velocity
- Sharing wires among multiple communication flows makes more efficient use of wires
- A standard interface facilitates modularity

QuickSilver Technology's Adaptive Computing Machine (ACM) [3,4] extends the previous work by introducing network protocols and hardware mechanisms designed to make the transfer of real-time streaming data among heterogeneous *nodes* as seamless as possible, and does so without the need for global control. Two key as-

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 125–134, 2004.

pects of the ACM architecture – and the focus of this paper – are (1) the *Point-to-Point* (*PTP*) *protocol* for transferring real-time streaming data over the ACM network and (2) the *node wrapper* which makes all nodes appear homogeneous regardless of their internal structure or functionality. With the PTP protocol and the node wrapper, nodes as diverse as digital signal processors, reduced-instruction-set processors, domain-specific processors, reconfigurable fabrics, on-chip and off-chip bulk memories and input/output ports can communicate seamlessly.

The present paper introduces the ACM architecture including network topology and streaming protocols and describes how the node wrapper provides a single, uniform and consistent mechanism for task management, flow control and load balancing across all node types. The hardware task manager – which is inspired by Dennis's pioneering work in data-flow computing [5,6] – is a key component of the node wrapper and is discussed in detail.

2 The Adaptive Computing Machine

Adaptive Computing Machines are targeted at satisfying the signal and image processing needs of low-power, handheld, mobile, wireless devices and other forms of consumer electronics.

2.1 The ACM Network

The Adaptive Computing Machine consists of a collection of heterogeneous nodes interconnected by a scalable, fractal-based network (Figure 1). The network has a single *root* to which are connected:

- Network input and output ports
- System port (optional)
- Internal and external bulk memory (optional)
- K-Node (Supervisor Node)
- One or more descending *quadtrees* with heterogeneous leaf nodes

The quadtrees are implemented using 5-ported *switch elements*, each connected to a single parent and up to four children. The switch elements implement a fair, round-robin arbitration scheme and provide pipelining with multi-level look-ahead for enhanced performance. At present, the width of all paths is constant (51 bits), but the option is available to widen pathways as a tree is ascended, in the style of Leiserson's *fat trees* [7], in order to increase network bandwidth.

2.2 Network Words

All traffic on the ACM network is in the form of 51-bit *network words* (Figure 2) where the fields are defined as follows:

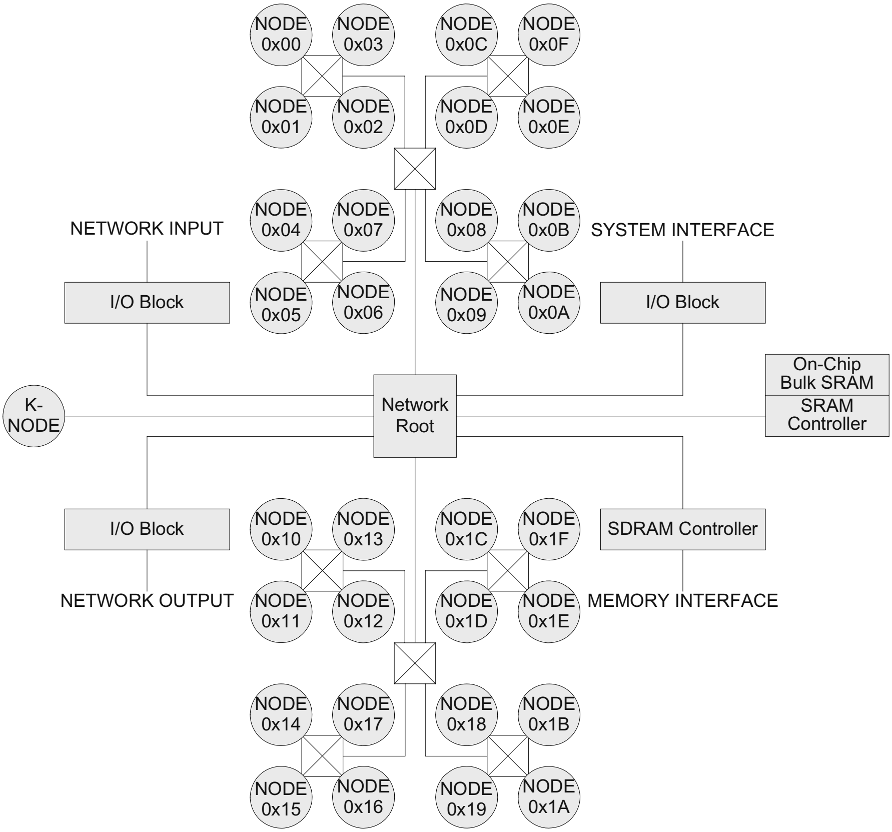


Fig. 1. ACM Network with 32 Leaf Nodes, K-Node (Supervisor Node), Off-Chip-SDRAM Controller and On-Chip SRAM with Controller

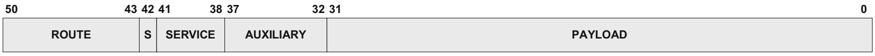


Fig. 2. A Network Word

Route – Destination address of the network word; The two high-order bits are the chip ID

S (Security Bit) – Bit allowing *peeks* (reads) and *pokes* (writes) to configuration memory; Set only for words sent by the K-Node

Service – Type of service

Auxiliary – Dependent on service type

Payload – Payload

The service field defines one of sixteen service types, two of which are of interest to us here:

Point-to-Point (PTP) – Streaming data

PTP Acknowledgement – Supports flow control for PTP data; Causes a *Consumer* or *Producer Count* at the destination node to be incremented or decremented

2.3 Nodes

Each node in the network has three elements as illustrated in Figure 3: a *node wrapper*, an *execution unit* (*EU*) and *memory* (*nodal memory*).

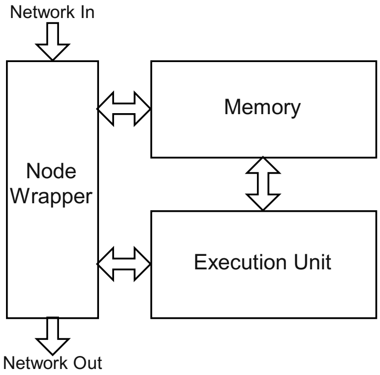


Fig. 3. A Cell

The wrapper makes the node identical in outward appearance to all other nodes regardless of its internal structure or functionality. The wrapper also relieves the execution unit from having to deal with myriad activities associated with task management and network interactions. Among other things, the wrapper is responsible for disposing of each incoming network word in an appropriate fashion – in one clock cycle.

The execution unit is responsible for executing *tasks*. It may take a wide variety of forms:

- Digital signal processor
- Reduced-instruction-set processor
- Domain-specific processor
- ASIC (application-specific integrated circuit)
- Reconfigurable (FPGA) fabric

But regardless of its form, the EU interacts with the node wrapper through a standard interface.

Nodal memory is accessible to both the node wrapper and the execution unit. It is where the node wrapper deposits incoming streaming data and where the EU accesses that data. A node's own memory, however, is typically *not* where the EU sends *output data*. To minimize memory accesses, output data is usually sent directly to the node(s) requiring that data: the *consumer node*(*s*). Nodal memory is also used to store task parameters and is available to tasks for temporary (scratchpad) storage.

3 Transferring Streaming Data

In a multi-node system where nodes are both consumers and producers of streaming data, matching production and consumption rates is a fundamental problem. A producer task on one node may produce data at a rate that is either greater than or less than the rate at which a consuming task on another node can handle. If the producer is sending data at a *greater* rate than the consumer can handle, then data is eventually lost. If the producer is sending data at a *lesser* rate than the consumer can handle, then the consumer may be starved for data, thereby potentially causing the consumer to sit idle waiting for additional data.

To address these issues, the ACM provides – via the Point-to-Point protocol and the node wrapper – a single, uniform and consistent mechanism for *task management*, *flow control* and *load balancing*. Task management ensures that a task is placed in execution only when it has sufficient input data and when there is sufficient space in the consumer node(s) to accommodate the data produced by the task. Flow control guarantees that a producer task will never overwhelm a consumer task with too much data in too short a time. Load balancing permits a producer task to distribute data among several alternate consumer nodes, thus allowing the producer task to operate at a potentially higher rate.

3.1 Point-to-Point Channels

Streaming data is transferred between two nodes (*points*) via a **Point-to-Point channel** (Figure 4). Associated with each PTP channel are:

- A **Producer Node** (Node A in Figure 4)
- A **Producer Task** running on the Producer Node's execution unit that produces a finite-sized block of PTP data per task activation, that block of data being sent over the PTP channel as a sequence of PTP words (Task 1 in Figure 4)
- An **Output Port** on the Producer Node that is associated with the Producer Task (Output Port j in Figure 4)
- A **Consumer Node** (Node B in Figure 4)
- An **Input Port** on the Consumer Node via which the Consumer Task receives PTP data from the PTP channel (Input Port k in Figure 4)
- A circular **Input Buffer** in the Consumer Node's nodal memory into which the incoming PTP data is deposited (Input Buffer k in Figure 4)
- A **Consumer Task** running on the Consumer Node's execution unit that *consumes* a finite amount of the PTP data residing in the circular input buffer per task activation (Task 2 in Figure 4)

Data is conveyed over a PTP channel when the Producer Task transfers a 50-bit **Point-to-Point word** (Figure 5) to the node wrapper in the Producer Node. (The 51^{st} bit, the Security Bit, is added later by the network.) The node wrapper, in turn, hands the PTP word over to the packet-switched network for transfer to the Consumer Node. The 8-bit Route Field of the PTP word provides the address of the Consumer Node, while the low-order 5 bits of the Auxiliary Field indicate to which of the Consumer

Node's input ports the data is directed. When the PTP word arrives at the Consumer Node, the node wrapper deposits the 32-bit payload into the circular input buffer associated with the indicated input port. The transfer is then complete.

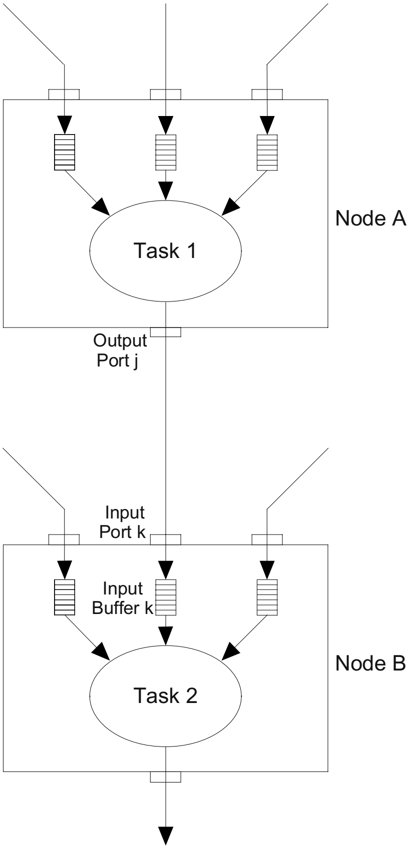


Fig. 4. A Point-to-Point Channel

50 43	42	41 38	37	36 32	31 0
Node		0 0 0 0	M	Port	Data

Fig. 5. A Point-to-Point Word

4 Task Management, Flow Control, and Load Balancing

Having described a simple mechanism for moving streaming data between two points in the ACM, we now turn our attention to the mechanisms for task management, flow control and load balancing.

4.1 Consumer Counts and Producer Counts

As already noted, there is an input buffer associated with each input port. There is also a two's-complement signed count associated with each port, both input and output.

For an input port, the count is referred to as a **consumer count** since it reflects the amount of data in that port's input buffer that is available to be *consumed* by the associated task. A consumer count is **enabled** when its value is non-negative – that is, when its sign bit is 0. An enabled consumer count indicates that the associated input buffer has the minimum amount of data required by an activation of the associated task. At system initialization, or upon reconfiguration, a consumer count is typically reset to –C, where C is the minimum number of 32-bit words required per task activation.

For an output port, the count is referred to as a **producer count** since it reflects the amount of available space in the downstream input buffer to accept the data that is *produced* by the associated task. A producer count is **enabled** when its value is negative – that is, when its sign bit is 1. An enabled producer count indicates that the downstream input buffer has space available to accommodate the maximum amount of data produced per activation of the associated task. At system initialization, or upon reconfiguration, a producer count is typically reset to P – S – 1, where P is the maximum number of 32-bit words produced per task activation and S is the size of the downstream input buffer in 32-bit words.

Notice that both consumer counts and producer counts are typically initialized to negative values, which means that consumer counts start out *disabled* while producer counts start out *enabled*. This initial state reflects the fact that input buffers are usually empty at system initialization/reconfiguration.

4.2 PTP Acknowledgements

Consumer and Producer Counts are updated by a system of *credits* and *debits* in the form of **forward acknowledgements** and **backward acknowledgements**. Both types of acknowledgements are network words (Figure 6) sent by a task as the last steps in a task activation. In both cases, the payload contains four fields: (1) a bit indicating the type of acknowledgement, (2) a port, (3) a task and (4) an Ack Value.

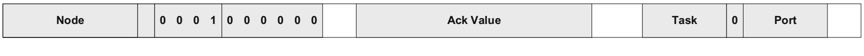


Fig. 6. A PTP Acknowledgement

The sequence of acknowledgements that a task performs at the end of each activation is as follows:

A. For each output port of the task:

1. Send a forward acknowledgement to the consumer node specifying the consumer input port and the consumer task; Ack Value is the number of PTP words the task just sent to the consumer input port

2. Send a backward acknowledgement (a *self ack*) to the node on which the task resides specifying the output port and the task; Ack Value is the number of PTP words the task just sent via the output port

B. For each input port of the task:

1. Send a backward acknowledgement to the producer node specifying the producer output port and producer task; Ack Value is *minus* the number of 32-bit words the task just consumed from the input port's buffer
2. Send a forward acknowledgement (a *self ack*) to the node on which the task resides indicating the input port and the task; Ack Value is *minus* the number of 32-bit words the task just consumed from the input port's buffer

4.3 The Hardware Task Manager

The hardware task manager is the part of the node wrapper responsible for updating consumer and producer counts in response to incoming acknowledgements. It also monitors the sign bits of those counts and launches a task when an appropriate set of counts is enabled. This last responsibility is met using two signed counts that are associated not with a port but with a task: a **task input count** and a **task output count**. A task's input (output) count reflects the number of task consumer (producer) counts that are enabled. A task count is said to be **enabled** when its value is non-negative. A task is **enabled** – and available for execution – when both its input count and its output count are enabled.

Incoming acknowledgements update various counts and cause tasks to be launched as follows:

A. If a *forward* acknowledgement is received:

1. Interpret the specified port as an *input* port, and add Ack Value to the corresponding consumer count
2. If the consumer count makes a transition from disabled to enabled (enabled to disabled), then increment (decrement) the input count of the specified task by 1

B. Else if a *backward* acknowledgement is received:

1. Interpret the specified port as an *output* port, and add Ack Value to the corresponding producer count
2. If the producer count makes a transition from disabled to enabled (enabled to disabled), then increment (decrement) the output count of the specified task by 1

C. If after Step A or B the specified task's input and output counts are both enabled, then place the task on the *ready-to-run queue* if it is not already on the queue; Launch the task when it reaches the head of the queue

These actions, in effect, embody the *firing rule* for tasks. They cause a task to be placed on the ready-to-run queue and ultimately executed when a sufficient number of consumer counts and a sufficient number of producer counts are enabled. What those *sufficient numbers* are is determined by the initial values of a task's input count and output count. If I (O) is the number of input (output) ports associated with a task and $IC_{Initial}$ ($OC_{Initial}$) is the initial value of the task's input (output) count, and if we assume that all consumer counts are initially disabled and all producer counts are initially enabled as discussed above, then a task *fires* when

$$-IC_{Initial} \text{ out of I consumer counts are enabled}$$
$$\text{AND}$$
$$(O - OC_{Initial}) \text{ out of O producer counts are enabled}$$

For example, for I = 4,

If $IC_{Initial} = -1$, then 1 out of 4 consumer counts must be enabled
If $IC_{Initial} = -2$, then 2 out of 4 consumer counts must be enabled
If $IC_{Initial} = -3$, then 3 out of 4 consumer counts must be enabled
If $IC_{Initial} = -4$, then 4 out of 4 consumer counts must be enabled

For O = 4,

If $OC_{Initial} = 3$, then 1 out of 4 producer counts must be enabled
If $OC_{Initial} = 2$, then 2 out of 4 producer counts must be enabled
If $OC_{Initial} = 1$, then 3 out of 4 producer counts must be enabled
If $OC_{Initial} = 0$, then 4 out of 4 producer counts must be enabled

4.4 Flow Control

Earlier, we said that flow control guarantees that a producer task will never overwhelm a consumer task with too much data in too short a time. In the context of the ACM, that means that a producer task will never overflow an input buffer of a consumer task. The mechanism that guarantees this property has been spelled out above:

1. The producer count associated with the output port of the producer task is initialized to P – S – 1 as described in Section 4.1.
2. The producer and consumer tasks perform the sequence of acknowledgments outlined in Section 4.2 upon the completion of each activation.

4.5 Load Balancing

The mechanism described in Section 4.3 – that launches a task when *just one* of its output ports is enabled – permits a producer task, usually a high-throughput task, to send the output of each activation to one of several alternate, usually lower-throughput, consumer tasks. The downstream processing load is thus distributed (balanced) among the several consumer tasks. To support this capability, the node wrapper makes available to the execution unit the identities of enabled input and output ports.

5 Conclusions

The node wrapper and the point-to-point protocol provide an asynchronous, distributed mechanism for handling streaming data in a system with heterogeneous nodes. This *distributed intelligence* supports task management, flow control and load balancing across a wide range of node types.

This work is connected to prior research in several related areas. Petri nets [8] were the first mathematical model to truly capture the notion of *concurrency*, and the firing rule described in Section 4.3 above can be seen as a generalization of Petri's original firing rule. Dennis's work on data-flow computing [5,6] – which inspired our work and was in turn inspired, at least in part, by Petri's work – is based on the principle that computing should be *data-driven*, that an operator (task) should fire (execute) when input data and output buffers are available. Seitz[1] and Dally[2] recognized the importance of making a transition from routing wires – the realm of field-programmable logic – to routing packets (tokens) – the realm of data-flow computing.

There is also a connection to Lysaght's work [9] on *logic caching* since the hardware task manager provides a mechanism for distributed, rather than centralized, control of logic caching. Finally, there is a connection to threshold logic which can be appreciated when one realizes that consumer and producer counts act as *threshold gates* with the output sign bit indicating whether a threshold has been reached.

References

1. Seitz, C. L.: Let's Route Packets Instead of Wires. In: Dally, W. J. (ed.): Proceedings of the 6th MIT Conference on Advanced Research in VLSI. MIT Press (1990) 133–37
2. Dally, W., Towles, B.: Route Packets, Not Wires: On-Chip Interconnection Networks. In: Proc. Design Automation Conf. (June 2001) 684–689
3. Master, P.: The Age of Adaptive Computing is Here. In: Glesner, M., Zipf, P., Renovell, M. (eds.): Proceedings of the 12th International Conference on Field Programmable Logic (FPL 2002). Lecture Notes in Computer Science, Vol. 2438. Springer-Verlag, Berlin Heidelberg New York (2002) 1-3
4. Guccione, S. A.: The Adaptive Computing Machine Combines DSP Programmability with the Power and Performance of Custom Hardware. In: Global Signal Processing Expo and Conference (GSPx). Dallas, TX (2003)
5. Dennis, J. B.: First Version of a Data Flow Procedure Language. In: Programming Symposium. Lecture Notes in Computer Science, Vol. 19. Springer-Verlag, Berlin, New York (1974) 362–376
6. Dennis, J. B.: The Evolution of 'Static' Data-Flow Architecture. In: Gaudiot, J.-L., Bic, L. (eds.): Advanced Topics in Data-Flow Computing. Prentice-Hall, Englewood Cliffs (1991) 35–91
7. Leiserson, C. E.: Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. In: IEEE Transactions on Computers, C-34(10) (October 1985) 892–901
8. Petri, C.A.: Kommunikation mit Automaten: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, Bonn (1962)
9. Lysaght, P., Dunlop, J.: Dynamic Reconfiguration of Field Programmable Gate Arrays. In: Moore, W., Luk, W. (eds.): Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications, Oxford (1993)

Improving FPGA Performance and Area Using an Adaptive Logic Module

Mike Hutton[1], Jay Schleicher[1], David Lewis[2], Bruce Pedersen[1], Richard Yuan[1],
Sinan Kaptanoglu[1], Gregg Baeckler[1], Boris Ratchev[1], Ketan Padalia[2],
Mark Bourgeault[2], Andy Lee[1], Henry Kim[1] and Rahul Saini[1]

[1] Altera San Jose, 101 Innovation Dr., San Jose CA, USA, 95134.
mhutton@altera.com
[2] Altera Toronto, 151 Bloor St. W., Toronto, Ontario, Canada, M5S 1S4

Abstract. This paper proposes a new adaptable FPGA logic element based on fracturable 6-LUTs, which fundamentally alters the longstanding belief that a 4-LUT is the most efficient area/delay tradeoff. We will describe theory and benchmarking results showing a 15% performance increase with 12% area decrease vs. a standard BLE4. The ALM structure is one of a number of architectural improvements giving Altera's 90nm Stratix II architecture a 50% performance advantage over its 130nm Stratix predecessor.

1 Introduction

Previous research on LUT-size for FPGAs [12][14][15][10] has consistently shown that a 4-LUT provides the best area-delay product. This is based on the fact that larger LUTs can absorb more logic and decrease the critical path length, but require increasing resources for LUT-mask and input muxing. Mainstream Altera and Xilinx SRAM-based FPGAs use a 4-LUT, though this sometimes comes with additional hardware to compose base logic elements.

Here we show a new view of this tradeoff, illustrated in Figure 1. By novel use of input sharing and fracturability we are able to get the advantages of larger LUT sizes without paying the high price of the additional inputs required to build 5 or 6-LUTs.

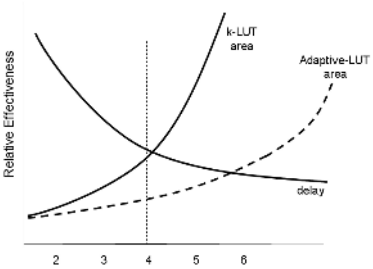


Fig. 1. Area/delay Tradeoff

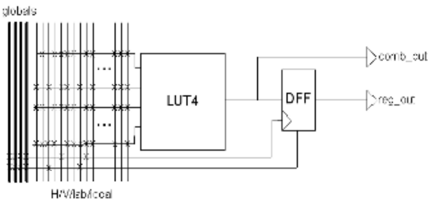


Fig. 2. BLE4 logic element

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 135–144, 2004.

The adaptive logic module developed in this paper allows us to decrease the critical path depth by 20% on average, but because the structure can be used either as a 6-LUT, two 5-LUTs with sharing, or other combinations without the need for expensive additional input muxing, we are able to achieve this without area penalty. With further improvements built on the ALM we can actually show an area benefit.

2 Logic Element Architecture and Adaptive Logic Modules

In this paper the generic BLE4 of Figure 2 and it's BLE5 and BLE6 analogs form the base comparison for the new logic structure. Empirically as we map for larger values of k nominal area and delay decrease. For BLE5 an average netlist uses 15% fewer LUTs and has 25% shorter unit delay; for BLE6 this is (-22%, -36%) and for BLE7 (-28%, -46%). However, as we move from BLE4 to BLE5 we add 16-bits of SRAM for the LUT-mask, and a new input mux to sample the neighboring connection block (see [5] for terminology). The true chip-area metric of #LEs * sizeof(LE) is minimized at about k=4 as shown in Figure 1. Related previous work also involves the use of heterogeneous LUT sizes [8], and hybrid PTERM/LUT architectures [9].

An interesting property of tech-mapping for larger LUTs is a decrease in efficiency. In a 6-LUT mapping, for example, only about 1/4 of LUTs end up as 6-input functions, the rest are underutilized. A design mapping to 100 LUTs with k=4 will map to 78 6-LUTs with a distribution of {23,32,17,9,13} LUT-{6,5,4,3,2} functions, based on experiments with RASP/FlowMap [7][6] and confirmed with Altera tech-mappers.

The cost of larger k is not just the LUT mask, though that is significant. Most dominant is the input mux, which is roughly 30:1 in a Stratix LAB. Though CLB and LAB routing structures are rather different this is also roughly 30:1 for VirtexII, based on quotes in [2]. The register, adder and other logic is unchanged.

Two reasons why a 6-LUT might be more preferable for depth than previously seen are that both the area devoted to routing and the relative delay contribution of interconnect to the critical path have been increasing consistently with new process generations. Also, the concept of LAB hierarchy and routing flexibility introduced in Altera's FLEX8K architecture, discussed in [3][1] and improved upon with depopulation [11] has minimized the effect. All modern FPGAs utilize some degree of cluster-based hierarchy. However, to use a 6-LUT effectively, we need to deal efficiently with the increase in input muxing, and the wastage involved in building a 6-LUT which is often underutilized.

If LUT-mask were the only concern, than we could compose multiple base LEs into larger LUTs, as shown in Figure 3.

The drawback with this approach is the ensuing area cost. This structure has a total of 19 input muxes, and 4 registers. When used as a 6-LUT, multiple signals have to be routed repeatedly through the connection block, and 3 of the 4 registers and sets of output muxing are wasted. The netlist of 100 BLE4 LUTs quoted above will re-map on average into 78 6-LUTs with a distribution of 23 LUT6, 32 LUT5, and 39 other. To implement this directly in the structure of Figure 3 would cost well more than 100

BLE4s in equivalent area. Using BLE5 as the base (meaning you can't use the BLE4 outputs) would help, but would still be more expensive than the BLE4 base.

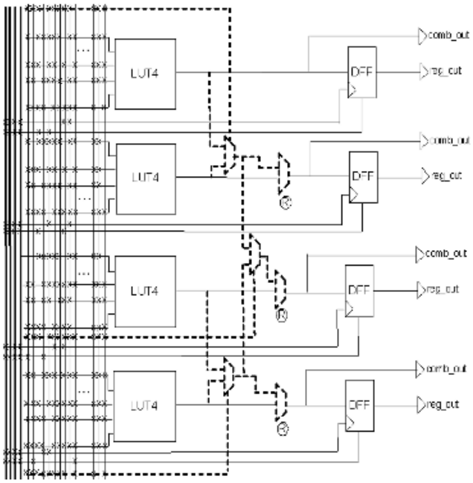


Fig. 3. Composing BLE4 to build a LUT6.

Our solution to this problem is the fracturable logic module, shown abstractly in Figure 4. The logic structure has a total of 8 input-muxes, and provides 4 functional outputs (2 comb, 2 reg), uses 64 bits of LUT-mask (6-input complete) and 2 bits of arithmetic and registers. We denote this a 6,2 fracturable LE, because it is a 6-LUT with 2 additional inputs for use when fracturing to smaller LUTs. These extra inputs are key to facilitating packing of non 6-input functions, without the overkill of adding a complete set of 10 or more inputs. Variants of this structure such as 4,2 are possible, but beyond the scope of this paper. Each of the two outputs (top and bottom) are denoted as an ALUT. This terminology is necessary in order to account for area later.

The ALM can implement one 6-LUT, two 5-LUTs which share 2 inputs, or two independent 4-LUTs, among other combinations.

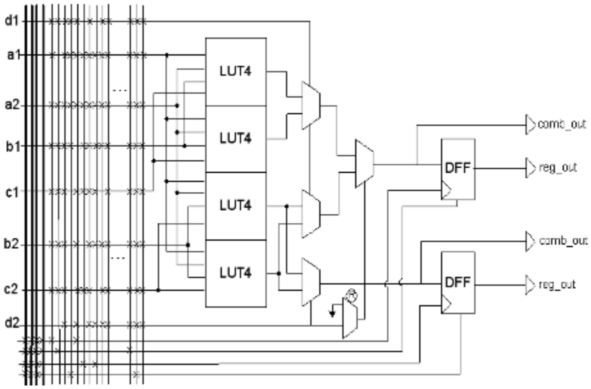


Fig. 4. Fracturable 6,2 adaptable logic module (first version).

Observe that the logic module of Figure 4 is more comparable in area with two BLE4 logic elements than with the four shown in Figure 3, because it has the same number of input mux, FF, output mux and arithmetic cost; only the proportion of the logic element devoted to the LUT-mask is increased. In functional terms, it is closer to the composition of two BLE5 logic elements.

Though the difference between composing two independent LEs and fracturing one compound structure is subtle, the most important issue to understand is the difference in the area cost of the two approaches.

2.1 Outputs, LUT-Mask Sharing, and 7-Input Functions

We can make a number of improvements to this first version of the ALM. One issue with Figure 4 is the number of outputs. We have partially addressed this by pushing the output merging back one stage in order to incur a speed hit only on the 6th stage input d2. When in fractured mode we set the SRAM-bit to 0 to disconnect the upper ALM-half from the lower.

However we can do better with the following transformation to Figure 5. First we duplicate the 2nd level muxes controlled by the 5th stage (d1 input), and add a new mux which choose c1 or GND on the top ALUT and c2 or VCC on the bottom. The effect of the transformation is to remove the additional output muxing from the critical path of the LE for all speed-paths, pushing it to the middle inputs only. Routing interfaces are identical to Figure 4 and are not shown.

As a further transformation, we introduce swap muxes controlled by R and T, for reasons which will become clear shortly.

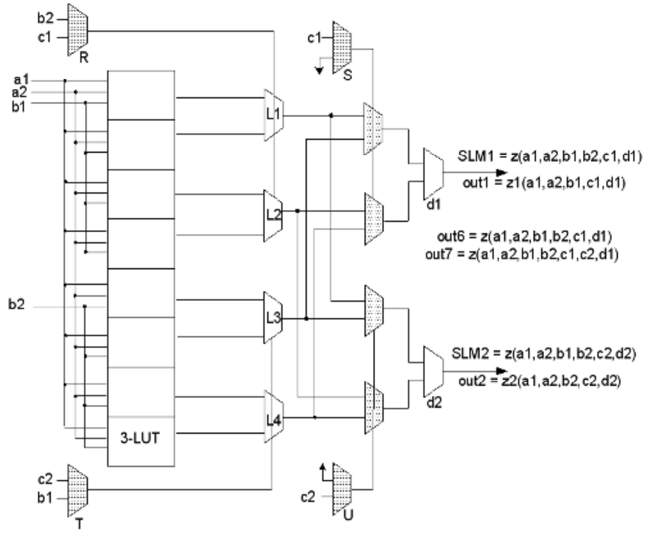

b2
c1
R
c1
S
a1
a2
b1
L1
L2
L3
L4
3-LUT
b2
c2
b1
T
c2
U
d1
d2
SLM1 = z(a1,a2,b1,b2,c1,d1)
out1 = z1(a1,a2,b1,c1,d1)
out6 = z(a1,a2,b1,b2,c1,d1)
out7 = z(a1,a2,b1,b2,c1,c2,d1)
SLM2 = z(a1,a2,b1,b2,c2,d2)
out2 = z2(a1,a2,b2,c2,d2)

Fig. 5. 6,2 ALM with 2 outputs and shared LUT-mask.

The operation of the logic module of Figure 5 is now a bit more complex. We still have two outputs and 8 inputs, and all previous properties. However there is an additional benefit from the latter transformation that makes it particularly clever. Note that when a 6,2 ALM is used in 6-LUT mode, there are two outputs unused (wasted). With the additional circuitry, which we call "shared LUT-mask" or SLM, we are now able to configure the logic module to implement two 6-input functions that share 4 inputs as long as they share the identical LUT-mask function. By setting R=1 and T=1, S=0 and U=1, and reorganizing the LUT mask appropriately, SLM1 becomes a 6-LUT function of (a1,a2,b1,b2,c1,d1) and SLM2 becomes the same function, only of (a1,a2,b1,b2,c2,d2).

This seemingly obscure property is incredibly useful in practice. Designs which contain multiple barrel shifters and crossbars will synthesize into many 4:1 muxes with common data and different select lines, which fit perfectly into the SLM structure. For example, a benchmark SPI-4 (posphy level 4) core is able to implement about 12% of all ALMs as packed pairs of 6-LUTs implementing 4:1 muxes, meaning a 12% overall savings in ALM area.

A further side-effect of this transformation is that the ALM of Figure 5 can also implement a restricted set of 7-input functions.

Setting R=0 the upper two 4-LUTs are arbitrary functions of (a1,a2,b1,b2). Setting T=1 the bottom 4-LUTs are arbitrary functions of (a1,a2,b2,c2). Setting S=1 makes the upper shaded muxes controlled by c1 and the results of these controlled by d1. When c1=0 out7 is driven by the L1 and L3 outputs chosen by d1. When c1=0 out7 is driven by the L2 and L4 outputs chosen by d1. The result is that we can compute a class of 7-input functions using all the inputs except for d2.

Specifically, we can implement any 7-input function that can be expressed as

F1 = fn(a1,a2,**b1**,b2,d1), F2 = fn(a1,a2,b2,**c2**,d1), Out = mux(F1,F2,c1)

Thus we can compute the c1-controlled mux of two 5-input functions which share 4 of their inputs, differing only in b1 and c2. The reason that the output of the functional template is controlled by c1 rather than d1 as shown in the physical diagram comes from the LUT-mask changes performed by synthesis to rotate the c1 and d1 effects (this does require some thought to see completely).

Figure 6 shows how to build an 8:1 mux in 2 ALMs (4 ALUTs) using this property. We first compute sub-functions y0 and y1. Since y0 and y1 are 5 input functions with two shared inputs they pack into a single ALM and generate the two outputs y0 and y1. In the second ALM we compute the output of the 8:1 mux using F1=fn(s0,s1,d3,y0,y1) and F2=fn(s0,s1,d7,y0,y1) and the 2:1 mux controlled by s2. In both sub-functions one of the bridged inputs is unused, but nonetheless the result is a partial 7-input function matching the above template.

An 8:1 mux implemented with simple BLE4 requires 5 BLE4 logic elements vs. 2 ALMs, which saves roughly the area of a BLE4.

It is worth noting that though the composable BLE4 structure of Figure 3 is not efficient for making 6-LUTs or 5-LUTs, it is quite useful for building muxes – it can build an 8:1 mux in 4 composable BLE4s, which is comparable in area to the 2 ALM solution.

As a final comment on the choice of 6-LUTs as the basis for the ALM, we note that in addition to 4:1 muxes, 6-input functions are natural implementations for many other logical functions. One such class of functions is DES encryption.

The core operation of DES is an array of 8 sboxes or substitution tables. Each sbox has 6 inputs and produces 4 outputs. In a parallel implementation when targeting speed, it is usual to produce 128 SBOXes, each of which needs 6 BLE4 for each of 4 outputs (3072 LEs).

The sbox has a natural implementation in 4 6-LUTs. Due to the complex nature of the function each of the 6-LUTs would otherwise require the worst-case 6 4-LUTs shown in Figure 7. This behavior is typical of other encryption functions such as Rijndael also, and is a further justification that 6-LUTs are a "natural" building block for combinational functions.

For an area-optimized DES core, the ALM described in this paper uses 239 ALMs vs. 736 in BLE4. For the speed-optimized version, we use 1465 ALMs vs. 5352 BLE4 logic elements. This represents a roughly 35% and 45% overall area improvement, respectively.

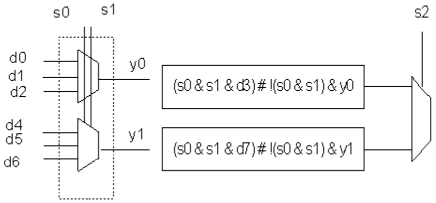

s0
s1
s2
d0
d1
d2
d4
d5
d6
y0
y1
(s0 & s1 & d3) # !(s0 & s1) & y0
(s0 & s1 & d7) # !(s0 & s1) & y1

Fig. 6. 7-input function for 8:1 mux

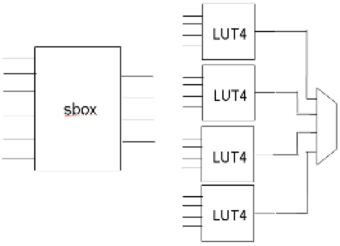


Fig. 7. DES sbox with BLE4

In a naïve implementation, the total area of the Figure 5 ALM is a little larger than two base BLE4s, roughly 15-20%. This comes from the additional LUT-mask SRAM bits and extra 2:1 muxes and configuration. However, since layout of the ALM is done as a pair, much of this can be clawed back with intelligent layout sharing. In overall chip area the physical implementation of Figure 5 is roughly area-neutral with the BLE4 architecture yet achieves the 36% decrease in logic depth.

3 Balanced Technology Mapping

It is critical to balance the distribution of LUT-sizes away from the natural distribution of tech-mapping to one which is more packable and facilitates SLM.

Consider Figure 8. The network on the left can be covered by three ALUTs – two 6-LUTs and one 4-LUT (upper solution), and this is the solution that FlowMap will generate. After packing, the solution has depth 2 and 2 ½ ALMs. The solution to the bottom, though it generates 4 ALUTs (all 5-LUTs), can be efficiently packed into just 2 ALMs while maintaining the depth-2 solution.

We refer to modifying the distribution of LUT-sizes to improve packing as balancing. The goal of balanced mapping is to maintain optimal critical path depth (unit delay) while producing a more packable LUT distribution.

The primary issue to tech-mapping for a good distribution is to modify cost functions to avoid 6-LUTs when they are not necessary for delay minimization. Further discussion is beyond the scope of this paper. However, Figure 9 shows empirical results from our prototype tools using default, balanced and aggressive balancing.

In the prototype we captured 7-LUT functions not by mapping to k=7, but rather by specifically recognizing 8:1 muxes in RTL synthesis, and then post-processing the netlist after tech-mapping. On average, we find that 7% of all ALMs are able to implement a 7-input function, a very significant area benefit.

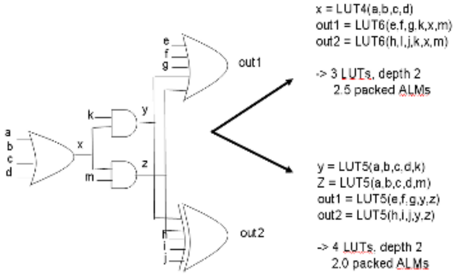


Fig. 8. Better with balanced mapping.

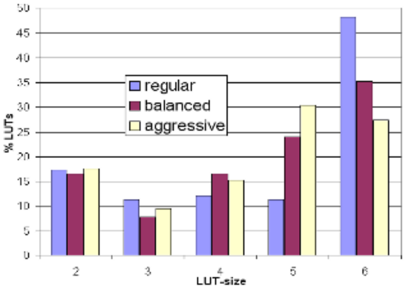


Fig. 9. LUT-size distribution

4 Experimental Results

This section shows results on 80 large VHDL/Verilog industrial designs, using prototype architecture development tools. Each design is synthesized and mapped by the Quartus II architecture evaluation flow, then clustered, placed and routed by our parameterized architecture evaluation toll PMT. Architectures are generated automatically by PMT based on the size of the design, to emulate as-full-as-possible chips.

For evaluating the ALM, the routing architecture of the LIM and global network outside the LAB is held fixed. However optimization sweeps for connectivity of the input mux is performed for both the BLE4 and ALM 6,2 cases, so that each receives its optimal layout and connectivity while maintaining routability. Timing delays for the different delays through the BLE4 and ALM are obtained by Spice simulation based on a preliminary layout on a common 130nm process. Routing delays are also obtained by Spice, and are common between the two architectures. Area models are computed using preliminary layout estimation, also using common 130nm process design rules. Layout optimizations and rough transistor sizing is done to optimize the BLE4 and ALM independently. As a caveat, design efforts were biased towards performance over area, so results could change slightly with different emphasis.

Figure 10 shows performance results (as a ratio), overall a 15% geomean improvement. Figure 11 shows chip area, with a 12% geomean improvement. For area, we capture the effect of clustering into LABs and routing architecture, by using the metric “labsize * sizeof(lab)” as this is the fairest comparison. Both architectures route all designs.

The ALM structure introduced in this paper is one of a number of architectural changes introduced in the Stratix II family of FPGAs recently announced by Altera, further architectural changes were also made to the LAB and routing structure, and these are in general additive.

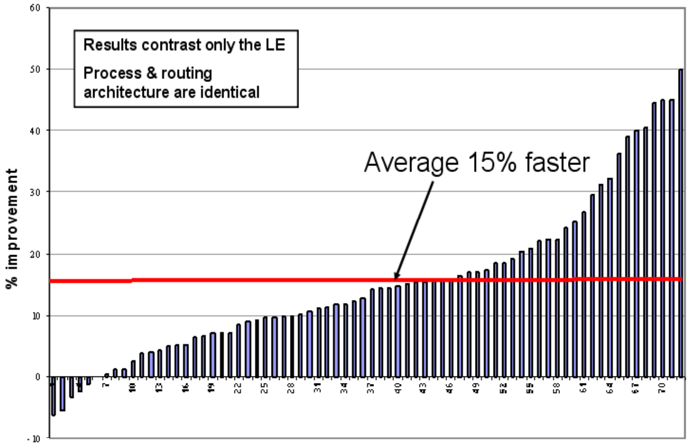


Fig. 10. Performance improvement of 6,2 ALM vs. BLE4

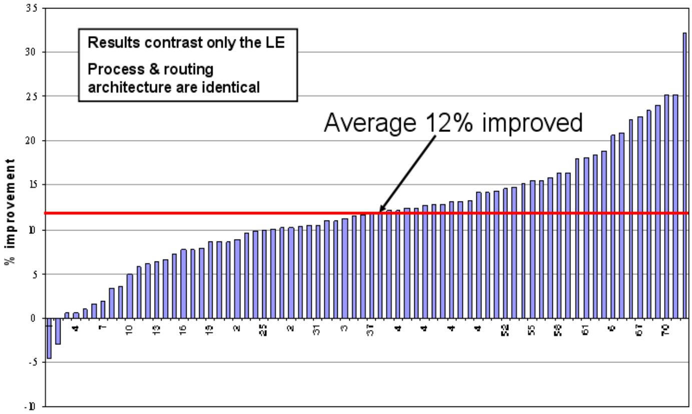


Fig. 11. Area improvement of 6,2 ALM vs. BLE4

Figure 12 shows the breakdown of overall performance gains in the Stratix II architecture over Stratix. In contrast with the earlier discussion, these are bottom-line results comparing production software and timing models in both cases and including the 90nm process gains for Stratix II.

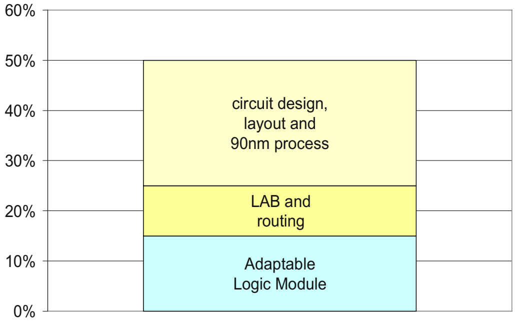


Fig. 12. Stratix II Silicon Performance Improvements vs. Stratix

5 Conclusions

This paper presents a new and novel adaptive logic module structure for FPGAs. The goals of the ALM is to allow technology mapping to 6-input functions in order to capture the depth benefits of wider functions without the unacceptable cost that would be incurred with a BLE6 based logic element.

We showed the sources of area cost of building logic elements with more than 4 inputs and how those costs break down into both the obvious costs (LUT-mask size) and the large but less apparent costs such as input and output muxing and appropriate number of FFs and outputs per block.

We presented a specific logic element based on a 6-LUT that is fracturable into 5 LUTs, but that using sharing and other optimizations can be implemented with area comparable to two BLE4 logic elements. Further extensions to the logic element improve propagation delay through the logic function, allow for partial functions of 7-inputs, and allow two 6-input functions that share 4 inputs and the same LUT mask to be implemented in the same logic element – a 2X area savings when used. Efficient balancing is achieved through improved software, and by choosing the right balance of extra inputs needed to achieve good packing results vs. the cost of providing them.

Overall comparisons between an FPGA architecture based on the 6,2 ALM and based on a 4-input LUT on the same process and with no routing architecture changes show an average performance gain of 15% and average decrease in chip area of 12%.

A version of the adaptable logic module described in this paper has been implemented as a key component of the Stratix II family of commercial FPGAs from Altera. The 15% performance improvement from the ALM along with further architectural changes and process migration results in a 50% average performance improvement between the 90nm Stratix II and it's predecessor Stratix on 130nm technology.

Acknowledgements. Thanks to Richard Cliff, Misha Burich, David Mendel and Paul Leventis for their reviews and comments improving the presentation of this paper.

References

[1] Ahmed and J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density," in Proc. ACM Symp. FPGAs, pp. 3-12, 2000.

[2] J. Anderson, F. Najm and T. Tuan, “Active Leakage Power Optimization for FPGAs”, in Proc. ACM Symp. FPGAs, pp. 33-41, 2004.

[3] V. Betz and J. Rose, “Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size”, in Proc. Custom Integrated Circuits Conference 1997, pp. 551-554.

[4] V. Betz, J. Rose and A. Marquardt. “Architecture and CAD for Deep-Subicron FPGAs”, Kluwer, 1999.

[5] S. Brown, R. Francis, J. Rose and Z. Vranesic, “Field-Programmable Gate Arrays”, Kluwer, 1992.

[6] J. Cong and Y. Ding, “FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs”, IEEE Trans. CAD Vol 13 No 1, pp. 1-12, 1994.

[7] J. Cong, J. Peck and Y. Ding, “RASP: A General Logic Synthesis System for SRAM-based FPGAs”, in Proc. ACM Symp. FPGAs, pp. 137-143, 1996.

[8] J. He and J. Rose, “Advantages of Heterogeneous Logic Block Architecture for FPGAs”, in Proc. IEEE Custom Integrated Circuits Conf. (CICC), pp. 7.4.1-7.4.5, 1993.

[9] Kaviani and S. Brown, “Hybrid FPGA Architecture”, in Proc. ACM Synp. FPGAs, pp. 3-9, 1996.

[10] J. Kouloheris and A.El Gamal, “FPGA Performance vs. Cell Granularity”, Proc. of Custom Integrated Circuits Conference, May 1991, pp. 6.2.1 - 6.2.4.

[11] G. Lemieux and D. Lewis, “Using Sparse Crossbars Within LUT Clusters”, in Proc. ACM Symp. FPGAs, pp. 59-68 2001.

[12] J. Rose, R.J. Francis, D. Lewis and P. Chow, “Architecture of Field-Programmable Gate Arrays: The Effect of Logic Functionality on Area Efficiency”, IEEE Journal of Solid-State Circuits, pp. 1217-1225, 1990.

[13] J. Rose, R.J. Francis, P. Chow and D. Lewis, “The Effect of Logic Block Complexity on Area of Programmable Gate Arrays”, Proc. IEEE Custom Integrated Circuits Conference (CICC), pp. 5.3.1-5.3.5 1989.

[14] S. Singh, “The Effect of Logic Block Architecture on FPGA Performance”, M.A.Sc. Thesis, University of Toronto, 1991.

[15] S. Singh, J. Rose, P. Chow and D. Lewis, “The Effect of Logic Block Architecture on FPGA Performance”, IEEE Journal of Solid-State Circuits, Vol 27 No. 3, pp. 282-287, 1992.

[16] S. Trimberger, K.Duong, and B. Conn, “Architecture Issues and Solutions for a High-Capacity FPGA”, Proc. ACM Synp. FPGAs, pp. 3-9, 1997.

[17] K. Veenstra, B. Pedersen, J. Schleicher and C. Sung, “Optimizations for a Highly Cost-Efficient Programmable Logic Architecture”, in Proc. ACM Symp. FPGAs, pp. 20-24, 1998.

A Dual-V_{DD} Low Power FPGA Architecture

A. Gayasen[1], K. Lee[1], N. Vijaykrishnan[1], M. Kandemir[1], M.J. Irwin[1], and
T. Tuan[2]

[1] Dept. of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
{gayasen,kiylee,vijay,kandemir,mji}@cse.psu.edu
[2] Xilinx Research Labs
2100 Logic Dr.
San Jose, CA 95124
Tim.Tuan@xilinx.com

Abstract. The continuing increase in FPGA size and complexity and the emergence of sub-100nm technology have made FPGA power consumption, both dynamic and static, an important design consideration. In this work, we propose a programmable dual-V_{DD} architecture in which the supply voltage of the logic blocks and routing blocks are programmed to reduce power consumption by assigning low-V_{DD} to non-critical paths in the design, while assigning high-V_{DD} to the timing critical paths in the design to meet timing constraints. We evaluate the effectiveness of different V_{DD} assignment algorithms and architectural implementations. Our experimental results show that reducing the supply voltage selectively to the non-critical paths provides significant power savings with minimal impact on performance. One of our V_{DD}-assignment techniques provides an average power saving of 61% across different MCNC benchmarks.

1 Introduction

In modern FPGAs, power consumption has become an important design consideration. Increasing performance and complexity have raised the dynamic power consumed per chip, while the use of deep sub-micron processes has resulted in higher static power in the forms of sub-threshold leakage and gate leakage. High power consumption requires expensive packaging and cooling solutions. In battery-powered applications, high power consumption may prohibit the use of FPGA altogether. Consequently, solutions for reducing FPGA power are needed.

Reducing the supply voltage (V_{DD}) is an effective technique for reducing both dynamic and static power. Dynamic power has a quadratic dependency on supply voltage, while both sub-threshold leakage (due to Drain Induced Barrier Lowering, DIBL) and gate leakage exhibit exponential dependencies on the supply voltage. However, reducing supply voltage also negatively affects circuit performance. A well-known technique to reap the benefits of voltage scaling without the performance penalty is the use of dual-V_{DD}. The timing critical blocks in the design operate on the normal V_{DD} (or V_{DDH}), while non-critical blocks

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 145–157, 2004.

operate on a second supply rail with a lower voltage (or V_{DDL}). While dual-V_{DD} ICs have been successfully used in low-power ASICs and custom ICs [17], no commercial FPGA today uses multiple V_{DD}'s for power reduction.[1]

The difficulty of designing a dual-V_{DD} FPGA is that the optimal V_{DD} assignment changes from one design to another. Consequently, if logic blocks are statically determined to be operating at low or high V_{DD}, the placement and routing algorithms need to be modified accordingly as in [11]. However, static assignment of V_{DD} to the blocks may prevent the ability to reduce power consumption or to meet timing constraints for some designs. In contrast, the use of V_{DD}-programmability for each block helps to tune the number of high and low V_{DD} blocks as desired by the application. In this approach, the challenge is in determining the V_{DD} assignments to each block. The need for level converters wherever a low-V_{DD} logic block drives a high-V_{DD} block and the associated delay and energy overheads are an important consideration when performing these V_{DD} assignments. Furthermore, positioning of the level converters influences the ability to assign lower V_{DD}'s to the routing blocks.

In this work, we propose a programmable dual-V_{DD} architecture in which the supply voltage of the logic and routing blocks are programmed to reduce power consumption by assigning low-V_{DD} to non-critical paths in the design, while assigning high-V_{DD} to the timing critical paths in the design to meet timing constraints. In our programmable dual-V_{DD} architecture (see figure 1), the V_{DD} of a circuit block is selected between V_{DDH} and V_{DDL} by using two high-V_T transistors (supply transistors) connecting the block to the supplies. The state (ON/OFF) of each supply transistor is controlled by a configuration bit, which is set by the V_{DD} assignment algorithm. The configuration bits are set either to connect the block to one of the power supplies or completely disconnect the block from both the power supply lines when the block is unused or idle. We evaluate the effectiveness of different V_{DD} assignment algorithms and implementation choices for an island style FPGA architecture designed in 65nm technology. Our results indicate that one of our V_{DD}-assignment techniques provides an average power saving of 61% across different MCNC benchmarks.

The remainder of this paper is organized as follows. In Section 2, we revise the related work, focusing in particular on power optimizations for FPGAs. In Section 3, we discuss our dual-V_{DD} FPGA architecture. Section 4 describes the experimental methodology we used, and discusses the V_{DD} assignment algorithms and the power estimation technique we used. Section 5 presents experimental results and section 6 concludes the paper.

2 Related Work

Most of the previous works on power modeling, estimation and reduction in FPGA have focused primarily on dynamic power. In [9], the dynamic power

[1] Xilinx Virtex-II FPGAs use different supply voltages for I/O and the core. Pass transistors used for interconnects are also supplied higher gate voltages to eliminate the V_T drop. But this is not targeted to reduce power.

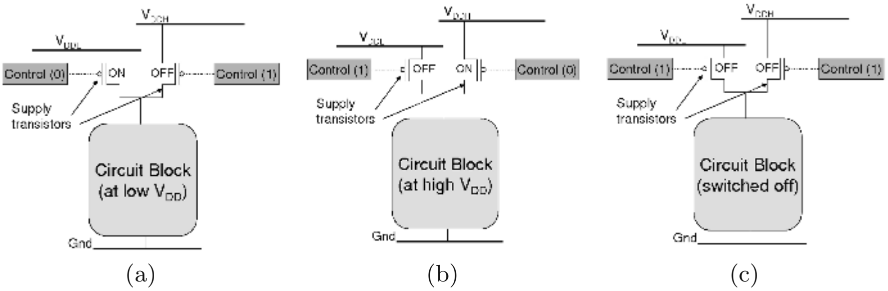


Fig. 1. Supply transistors used for programmable V_{DD}

of a Xilinx XC4003A FPGA was analyzed by taking measurements of test designs. [15] analyzes dynamic power consumption in Virtex-II FPGA family. [12, 10] evaluate different FPGA architectures for power efficiency. [16] presents a routability-driven bottom-up clustering technique for area and power reduction in clustered FPGAs.

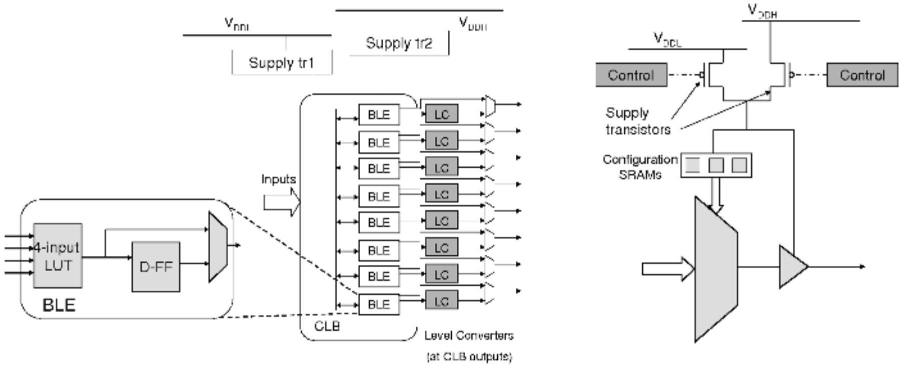
Leakage in FPGAs has captured interest only very recently. [18] makes a detailed analysis of leakage power in Xilinx CLBs. It concludes that significant reduction of FPGA leakage is needed to enable the use of FPGAs in mobile applications. [3] presents a fine-grained leakage control scheme using sleep transistors at gate level. [14] evaluates several low-leakage design techniques for FPGAs and shows that using multiple V_T switch blocks reduces leakage significantly. [1] selects the polarities of logic signals to reduce active leakage power in FPGAs. [5] presents a cut enumeration algorithm targeting low power technology mapping for FPGA architectures with dual supply voltages. [6] presents a region-constrained placement approach to reduce leakage in FPGAs.

Dual-V_{DD} techniques have been proposed previously for ASICs [19,17]. Recently, a low-power FPGA using pre-defined dual-V_{DD}/dual-V_T fabrics has been proposed in [11]. But, they have focused on reducing only dynamic power, while keeping the leakage constant. Further, they have used a fixed dual-V_{DD}/dual-V_T fabric, keeping all the routing resources at high-V_{DD}, which limits the power savings significantly.

3 Architecture

The proposed dual-V_{DD} architecture is built on cluster-based island-style FPGA architecture, with the configuration stored in SRAM cells. It facilitates configurable supply voltage for logic blocks and routing multiplexers. Figure 2 gives an overview of the architecture. The basic logic element (BLE) consists of a 4-input LUT and a flip-flop. Eight such BLEs cluster together to form a logic block (CLB). Figure 2(a) shows how the CLB is configured using high-V_T supply transistors to operate at two different voltages.

As mentioned in section 1, a dual-V_{DD} design needs level conversion when a low-V_{DD} block drives a block operating at high-V_{DD}. In our dual-V_{DD} architec-



(a) Dual-V_{DD} CLB (b) Dual-V_{DD} routing mux

Fig. 2. Dual-V_{DD} architecture

ture, level conversion takes place only at CLB pins. For this purpose, CLB pins have level converters (LCs) attached to them. A multiplexer allows to by-pass the level converter if level conversion is not needed at that pin. Placing the level converter only at CLB pins reduces the complexity of the routing fabric, and at the same time, limits the overheads due to level converters.

We experimented with two architectures differing in the placement of the level converters. While the first architecture places LCs at the output pins of CLBs; the second architecture places them at CLB input pins. Figure 2a shows the first case, where only the output pins of a CLB have LCs attached to them. In this case, a net with multiple fanouts operates at high V_{DD} if any one of the CLBs driven by this net is at high V_{DD} (since, the signal's voltage level does not change in the routing fabric). This limits the number of routing muxes that can be operated at low V_{DD}, and therefore, is less effective in reducing routing power compared to the case when LCs are attached to CLB input pins. But, the drawback of keeping LCs at input pins of CLBs (apart from area penalty) is that a larger number of LCs are needed, which increases the leakage in logic blocks. Our results support this reasoning, but show that overall leakage is lower for the second case.

Figure 2(b) shows a routing multiplexer (mux) in the dual-V_{DD} architecture. The multiplexer's output is connected to a level-restoring buffer to restore the V_T-drop through the NMOS-based multiplexer. Note that the same set of supply transistors control the voltage of configuration SRAM cells and the level-restoring buffer. Since the configuration SRAM is not timing critical, the supply transistors need to be sized just enough to supply the maximum current needed by the level-restoring buffer connected to it.

If a circuit block (CLB or routing multiplexer) is completely unused, then in order to save leakage, it is desirable to completely switch-off that block. This is achieved by keeping a separate configuration bit for every supply transistor[2]. Although this incurs more area overhead, it results in significant leakage savings, since resource utilization in an FPGA is typically low [18]. Due to the area overhead of level converters and supply transistors, the dual-V_{DD} FPGA takes approximately 21% more area than a single-V_{DD} FPGA when LCs are at CLB outputs. For the case when LCs are at CLB inputs, this number is estimated to be around 23%.

Majority of leakage in an FPGA occurs in the configuration SRAM cells. It has been previously shown in [6] that by increasing the threshold voltage of the configuration SRAM, its leakage can be reduced by 98%, while increasing configuration time by 20%. Since configuration time is not critical in most of our target designs, this tradeoff for power savings is reasonable. In order to see the effect of dual-V_{DD} on power consumption, we have neglected the configuration SRAM leakage both for single supply design, and for the dual supply design (since the reduction of configuration SRAM leakage is achieved by increasing its threshold voltage, and is equally applicable to both single and dual supply designs).

3.1 Level Conversion

Level converters have been studied widely ever since multi-V_{DD} circuits were proposed [19,13]. The area, delay and power overheads of level converters prohibit random V_{DD} assignment to logic elements of a circuit. For the present work, we have used the level converter circuit shown in Figure 3, and a 65nm BSIM4 SPICE model to simulate it. For an FPGA architecture where level converters are placed at CLB input pins, four level converters are required per BLE. For a V_{DDH} of 1.1V and V_{DDL} of 0.9V, the LC delay is almost 17% of the delay of an LUT, and as much as 41% of the clock-to-Q delay of the flip-flop. This significant delay in the LC prohibits the use of many LCs within a logical path of the circuit. In contrast to delay, power consumption in an LC was observed to be negligible ($< 1\%$) compared to a BLE. This allows us to place LCs at all input pins of a CLB and still get power savings.

4 Methodology

We used VPR and its power model [2,12] for this work. MCNC benchmarks were used for experimentation to evaluate the dual-V_{DD} architecture and V_{DD} assignment algorithms. The routing architecture that we supplied to VPR closely resembles a modern FPGA, with a routing channel width of 200, and buffered segments of lengths 1, 2, 6 and “long”. The LUT-size of 4, and cluster-size of 8 LUTs are chosen to be same as a Xilinx Virtex-II device.

Circuit simulations were performed in SPICE using 65nm BSIM4 device models. Delays of BLE and LC were obtained from these simulations. Power consumption, both static and dynamic, of the LC was also obtained by simulating

[2] In case of a routing mux, we may need to pull down the control signals when the mux is unused. The pull-down transistors can be sized very small.

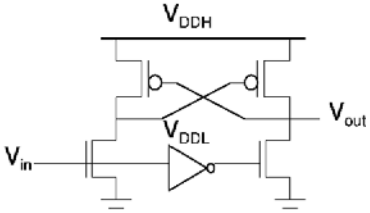


Fig. 3. Level converter circuit

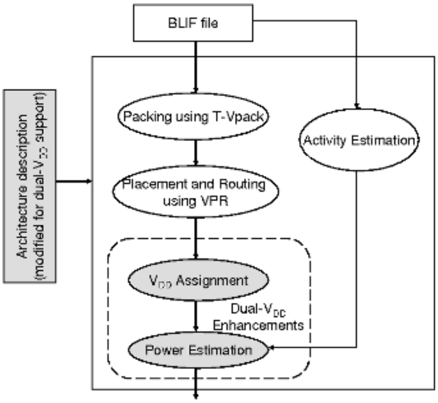


Fig. 4. Experimental Flow

in SPICE using BSIM4 models. Figure 4 shows the experimental flow. The flow deviates from a normal VPR flow after the place and route stage. We first assign voltage to all CLBs using algorithms that are discussed below, and then estimate power of the design placed and routed on the target dual-V_{DD} architecture. Assigning voltages after routing makes the timing analysis more accurate, since all the routing delays get incorporated in the timing graph.

4.1 V_{DD} Assignment

In order to be effective, a dual V_{DD} scheme requires that paths in the circuit vary in their delays. If all paths are of same delay then all circuit elements will require high V_{DD} to maintain the performance of the design.

Figure 5 shows the distribution of path delays averaged over MCNC benchmarks which we used for all our experimentation. It is evident from the figure that path delays in a circuit vary considerably. Therefore, a dual-V_{DD} scheme can be expected to reduce the power consumption significantly. Figure 5 also shows the path delays after using our dual-V_{DD} assignment algorithms.

Optimal assignment of V_{DD} to gates in a circuit is known to be an n-p complete problem. We use the heuristic shown in figure 6 for V_{DD} assignment. Initially we assign low V_{DD} to all CLBs in the FPGA, and find those paths whose delays become greater than the desired clock time period. e call such paths "critical". Those CLBs which do not belong to any of the critical paths can be kept at low voltage without affecting performance of the design. Some of the remaining CLBs and routing muxes need to operate at high-V_{DD} so that the design's performance target is met. The order in which these CLBs are analyzed is crucial for the performance of the heuristic. We define "criticality" of a CLB, as the number of critical paths that pass through this CLB[3]. The CLBs within

[3] This definition of criticality can potentially be improved by assigning priorities to paths depending on their delay or other parameters.

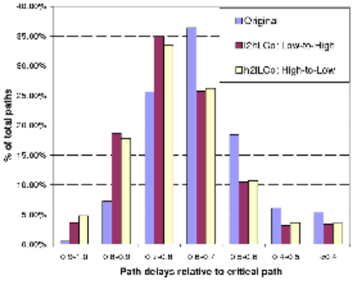


Fig. 5. Distribution of path delays

Table 1. Comparison of High-to-Low and Low-to-High algorithms (LC at CLB inputs, $V_{DDH} = 1.1$V, $V_{DDL} = 0.9$V

Design	# CLBs	# V_{DDL} CLBs	
		Low-to-High	High-to-Low
alu4	191	51	65
apex2	235	54	74
apex4	158	46	26
bigkey	214	24	81
des	200	87	127
dsip	172	12	31
elliptic	451	339	327
ex1010	575	177	185
ex5p	133	30	24
misex3	175	18	16
pdc	572	400	405
s38584.1	806	724	739
seq	219	61	58
spla	462	215	226
tseng	131	102	114

```
Assign $V_{DDL}$ to all CLBs and routing muxes
P = list of all paths in the design
T = longest delay path when all circuit blocks operate at $V_{DDH}$
$T_d = x$T, where $x \geq 1$ is a user-defined performance metric
critical_path = $\{P_i \in P \mid delay(P_i) > T_d\}$
for each CLB
        criticality(CLB) = number of paths passing through it
while (critical_path not empty) {
        $P_k$ = path $\in$ critical_path with maximum delay
        N = all blocks through which $P_k$ flows
        Sort N based on criticality (first entry has most paths)
        while ($delay(P_k) > T_d$) {
                $N_i$ = first(N)
                N = N - $N_i$
                Assign $V_{DDH}$ to $N_i$ and all the routing muxes driven by $N_i$
                update delay of all paths passing through $N_i$
        }
        critical_path = critical_path - $\{P_k\}$
}
```

Fig. 6. Algorithm for V_{DD} assignment: Low-to-High (assuming LCs at CLB input pins)

a path are analyzed in decreasing order of their criticalities. We started with CLBs on the most critical path, and proceeded to smaller paths in decreasing order of their delay. Figure 6 shows the algorithm for the case when LCs are at CLB inputs. In that case all routing muxes driven by a CLB have the same

```
Assign $V_{DDH}$ to all CLBs and routing muxes
P = list of all paths in the design
T = longest delay path when all circuit blocks operate at $V_{DDH}$
$T_d = x$T, where $x \geq 1$ is a user-defined performance metric
$vddl\_delay(P_i)$ = delay($P_i$) when all blocks in $P_i$ are at $V_{DDL}$
critical_path = $\{P_i \in P \mid vddl\_delay(P_i) > T_d\}$
for each CLB
        criticality(CLB) = number of paths passing through it
while (critical_path not empty) {
        $P_k$ = path $\in$ critical_path with maximum delay
        N = all blocks through which $P_k$ flows
        Sort N based on criticality (last entry has most paths)
        while (($delay(P_k) < T_d$) & (N not empty)) {
                $N_i$ = first(N)
                N = N - $N_i$
                Assign $V_{DDL}$ to $N_i$ and all the routing muxes driven by $N_i$
                calculate delays of all paths flowing through $N_i$
                if any of the delays > $T_d$
                        reset $N_i$ and all routing muxes driven by $N_i$ to $V_{DDH}$
                else
                        update delays of all paths flowing through $N_i$
        }
        critical_path = critical_path - $\{P_k\}$
}
```

Fig. 7. Algorithm for V_{DD} assignment: High-to-Low (assuming LCs at CLB input pins)

voltage as the CLB. For the other situation, when LCs are at CLB outputs, the voltage of routing muxes driving a CLB is the same as that of the CLB.

In order to enumerate all paths whose delays become larger than the required clock time period, we used the algorithm proposed in [8]. It maintains all paths in a heap data structure with their delays as the keys. Each path also maintains all the branch-points in the path in increasing order of their branch-slacks[4].

We experimented with a variant of the above algorithm (High-to-Low) too, in which all the CLBs are initially kept at high voltage and then some of them are changed to low V_{DD} (see figure 7). Before changing a CLB to low-V_{DD}, we need to make sure that this will not increase the delay of some other path in the circuit above the desired clock period. The number of low V_{DD} blocks using both versions, for V_{DDH} of 1.1V and V_{DDL} of 0.9V (for 65nm technology) is shown in table 1. For 10 out of 15 designs, the High-to-Low (h2l) version performs better than Low-to-High (l2h). This happens because in case of h2l, when the CLBs on a particular path are being analyzed whether they can be run on low-V_{DD}, the algorithm continues to look at all the other CLBs on the path even after it failed

[4] Branch slack is defined as the decrease in path delay if a particular branch point is used to generate a new path

to change the V_{DD} of some CLB. In contrast, in the l2h case, the algorithm keeps changing CLBs on a path to high V_{DD} (in decreasing order of criticality), till the delay of the path is less than the required clock period. This sometimes causes the path's delay to be reduced more than what was necessary.

4.2 Power Estimation

After all logic blocks have been assigned appropriate supply voltages, we estimate power consumption of the entire FPGA. We concentrate only on the power consumption in the core of the FPGA, and do not try to optimize or estimate IO power consumption. Furthermore, we did not estimate the power consumption in the global routing grid used for clock distribution.

In order to estimate dynamic power, VPR's power model calculates transition densities at all internal nodes of the FPGA, assuming that all inputs to the FPGA have the same static probability (default: 0.5). Capacitances are estimated from the capacitance values of a MOSFET, and that of wires and switches, all of which need to be provided in the architecture file taken by VPR as an input. We used Berkeley Predictive 65nm technology parameters for our experimentation.

We modified VPR's dynamic power model to include dual supply voltages, and the power consumption of level converters. Due to quadratic dependence of dynamic power on supply voltage, dynamic power of a circuit element reduces by $(\frac{V_{DDL}}{V_{DDH}})^2$ when its voltage is reduced from V_{DDH} to V_{DDL}. Dynamic power of a level converter (obtained from SPICE simulations) was added wherever a level converter was used (using the transition density at that node).

VPR has got a basic leakage model, which calculates sub-threshold leakage due to weak inversion. But in a 65nm technology, two more effects, namely, DIBL and gate leakage become significant, and need to be included in the leakage estimation. We also modified the leakage model to take into account multiple supply voltages, and sleep modes. Specifically, the following modifications were made to VPR's leakage estimation.

1. Gate leakage and sub-threshold leakage due to DIBL were included in the leakage estimation. In order to estimate leakage of a single MOSFET, we used results from SPICE simulations. 65nm BSIM4 device models were used. Simulations were performed for various supply voltages to get leakage numbers for different voltages. These numbers were incorporated into the power model of VPR to estimate gate leakage of the entire FPGA.
2. We estimated average leakage in a routing multiplexer by halving the worst case leakage, as discussed in [14]. To verify, we simulated multiplexers of various sizes and structures and found our leakage estimate to be very close to the SPICE results.
3. In the dual-V_{DD} FPGA, unused logic blocks and routing muxes are kept in a sleep state, by switching off both the supply transistors. Circuit simulations in SPICE showed that in sleep mode, leakage of a circuit block reduces to 10% of the original (high V_{DD}) leakage.

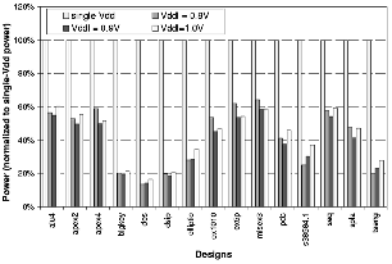


Fig. 8. Power consumption for different V_{DDL}'s. $V_{DDH} = 1.1V$.

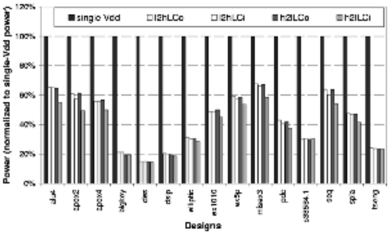


Fig. 9. Power consumption for different architectures and algorithms.

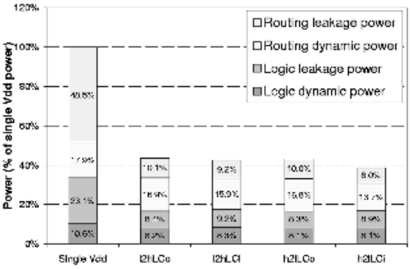


Fig. 10. Average power breakdown between logic and routing resources.

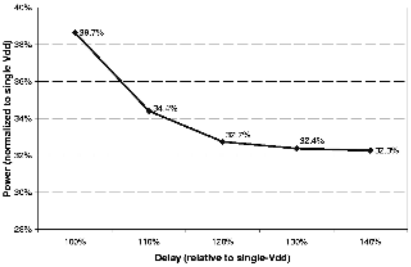


Fig. 11. Average power consumption for different critical path delay tolerances.

4. To estimate level converter leakage, we obtained the leakage number for one level converter from SPICE simulations, and multiplied this by the number of level converters in the FPGA.

5 Results and Analysis

Power reduction due to the dual-V_{DD} architecture strongly depends on the voltage values of V_{DDH} and V_{DDL}. In order to understand this dependence, and to come up with a good voltage choice, we fixed the high-V_{DD} at 1.1V and varied V_{DDL} from 0.8V to 1.0V. Figure 8 shows the power consumption for different V_{DDL} values (using High-to-Low Algorithm, LC at CLB's inputs) . Note that for 11 (out of 15) designs, V_{DDL} value of 0.9V results in maximum power savings. When V_{DDL} is increased to 1.0V, although the number of CLBs on low V_{DD} increases, the total power consumption increases. This happens because power consumption of the circuit elements at 1.0V is significantly higher than at 0.9V. On the other side, when we reduce V_{DDL} to 0.8V, power consumption again increases because the number of CLBs and routing muxes on low V_{DD} becomes too low. Therefore, for all other results in this section, we use a V_{DDL} of 0.9V. For this case, on an average, we get close to 61% power saving.

Figure 9 shows the power consumption of the designs for the two algorithms — High-to-Low (h2l) and Low-to-High (l2h), and level converter placements — at CLB outputs (LCo) or inputs (LCi). (h2lLCi denotes High-to-Low algorithm with LC at CLB Inputs.) Note that for most designs, the High-to-Low algorithm outperforms the Low-to-High algorithm. This is expected because we showed in section 4 that the High-to-Low algorithm resulted in larger number of low-V_{DD} CLBs. Further, the placement of LCs at CLB inputs saves more power (average: 61%) than their placement at outputs (average: 57%). This happens because LC leakage is not large enough to overshadow the gains we get in routing power by placing LCs at CLB inputs. But note that placing the LCs at CLB inputs increases the area of the FPGA.

Figure 10 shows the static and dynamic power consumption in both logic and routing resources for the different algorithms and LC placements. An important observation is that not all components of power are reduced by the same factor. The reduction in dynamic power is much less than that in leakage. For example, using High-to-Low algorithm and placing LC at CLB inputs saves 24% dynamic power and 76% leakage power. This can be attributed to two factors. First, in an FPGA since there exist a large number of unused circuit elements, it is possible to reduce the leakage in them by switching them off. And second, leakage varies exponentially with supply voltage, but dynamic power varies only quadratically with supply voltage. Note that leakage in routing resources reduces to less than 17% of the original, because in most designs it is possible to put a large number of routing muxes in sleep state, as they are sparsely used. Another trend to note is that the logic portion of leakage is larger when LCs are placed at CLB inputs (LCi) than when they are placed at CLB outputs (LCo). This implies that the larger overall power saving for the LCi case comes entirely from the routing resources.

Finally, figure 11 shows what happens when we modify the V_{DD} assignment algorithm to allow some degradation in the performance of the design. In the figure, a delay value of 110% denotes 10% performance penalty. Note that these delay values may increase after circuit implementation due to the use of supply transistors, and due to a possible increase of wire lengths (since total CLB area and consequently inter-CLB distances increase). Using h2lLCi, a 10% decrease in performance increases the average power saving by around 4%. But beyond 20%, power saving remains almost constant.

6 Conclusion and Future Work

We have presented a dual-V_{DD} FPGA architecture that provides significant power savings with minimal performance penalty. Variations of the V_{DD} assignment algorithm and level converter placement were explored. It was observed that High-to-Low Algorithm coupled with placement of level converters at the input pins of CLBs resulted in maximum power savings. An average power saving of 61% was observed for this case. The dynamic power was reduced by 24%, while the reduction in static power was close to 76%. But placing the level converters at CLB output pins reduces the area penalty by about 2% and still saves about 57% of total power.

In the present work, the router in VPR is essentially unaware of multiple supply voltages available for every logic block and routing switches. This could be improved by performing a dual-V_{DD} aware routing. We plan to work on this in future.

Acknowledgements. This work was supported in part by NSF CAREER Award 0093082, NSF Awards 0130143, 0202007, and a MARCO/DARPA GSRC Grant.

References

1. J. H. Anderson, F. Najm, and T. Tuan. "Active Leakage Power Optimization for FPGAs". In *Proceedings of ACM/SIGDA International Symposium on Field-programmable gate arrays*, 2004.
2. V. Betz and J. Rose. "VPR: A New Packing, Placement and Routing Tool for FPGA Research". In *International Workshop on Field-programmable Logic and Applications*, 1997.
3. B. Calhoun, F. Honore, and A. Chandrakasan. "Design Methodology for Fine-grained Leakage Control in MTCMOS". In *Proceedings of International Symposium on Low Power Electronics and Design*, 2003.
4. A. Chandrakasan, W. Bowhill, and F. Fox. *"Design of High-Performance Microprocessor Circuits"*. IEEE Press, 2001.
5. D. Chen, J. Cong, F. Li, and L. He. "Low-Power Technology Mapping for FPGA Architectures with Dual Supply Voltages". In *Proceedings of International Symposium on Field-programmable gate arrays*, 2004.
6. A. Gayasen, Y. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and T. Tuan "Reducing leakage energy in FPGAs using region-constrained placement". In *Proceedings of International Symposium on Field-programmable gate arrays*, 2004.
7. V. George, H. Zhang, and J. Rabaey. "The design of a low energy FPGA". In *Proceedings of International Symposium on Low Power Electronics and Design*, 1999.
8. Y-C. Ju and R. A. Saleh. "Incremental Techniques for the Identification of Statically Sensitizable Critical Paths". In *Design Automation Conference*, 1991.
9. E. Kusse and J. Rabaey. "Low-Energy Embedded FPGA Structures". In *Proceedings of International Symposium on Low Power Electronics and Design*, 1998.
10. F. Li, D. Chen, L. He, and J. Cong. "Architecture Evaluation for Power-Efficient FPGAs". In *Proceedings of ACM/SIGDA International Symposium on Field-programmable gate arrays*, 2003.
11. F. Li, Y. Lin, L. He, and J. Cong. "Low-power FPGA using Pre-Defined Dual-Vdd/Dual-Vt Fabrics". In *Proceedings of ACM/SIGDA International Symposium on Field-programmable gate arrays*, 2004.
12. K. Poon, A. Yan, and S. Wilton. "A flexible Power Model for FPGAs". In *Proceedings of International Conference on Field Programmable Logic and Applications*, 2002.
13. R. Puri, L. Stok, J. Cohn, D. Kung, D. Pan, D. Sylvester, A. Srivastava, and S. Kulkarni. Pushing ASIC performance in a power envelope. Design Automation Conference, 2003.

14. A. Rahman and V. Polavarapuv. “Evaluation of Low-Leakage Design Techniques for Field Programmable Gate Arrays”. In *Proceedings of ACM/SIGDA International Symposium on Field-programmable gate arrays*, 2004.
15. L. Shang, A. S. Kaviani, and K. Bathala. “Dynamic power consumption in Virtex[tm]-II FPGA family”. In *Proceedings of ACM/SIGDA International Symposium on Field-programmable gate arrays*, 2002.
16. A. Singh and M. Marek-Sadowska. “Efficient Circuit Clustering for Area and Power Reduction in FPGAs”. In *Proceedings of ACM/SIGDA International Symposium on Field-programmable gate arrays*, 2002.
17. M. Takahashi et.al. “A 60-mW MPEG4 Video Codec Using Clustered Voltage Scaling with Variable Supply-Voltage Scheme”. In *IEEE Journal of Solid-State Circuits*, Vol. 33, No. 11, Nov 1998.
18. T. Tuan and B. Lai. “Leakage Power Analysis of a 90nm FPGA”. In *Custom Integrated Circuits Conference*, 2003.
19. K. Usami and M. Horowitz. “Clustered voltage scaling technique for low-power design”. In *Proceedings of International Symposium on Low Power Electronics and Design*, 1995.

Simultaneous Timing Driven Clustering and Placement for FPGAs

Gang Chen and Jason Cong

Computer Science Deparment, University of California,
Los Angeles, CA 90095, USA
{chg, cong}@cs.ucla.edu

Abstract. Traditional placement algorithms for FPGAs are normally carried out on a fixed clustering solution of a circuit. The impact of clustering on wirelength and delay of the placement solutions is not well quantified. In this paper, we present an algorithm named *SCPlace* that performs simultaneous clustering and placement to minimize both the total wirelength and longest path delay. We also incorporate a recently proposed path counting-based net weighting scheme [16]. Our algorithm *SCPlace* consistently outperforms the state-of-the-art FPGA placement flow (*T-VPack* + *VPR*) with an average reduction of up to 36% in total wirelength and 31% in longest path delay.

1 Introduction

A typical LUT-based FPGA architecture [1] contains a two-level physical hierarchy: Basic Logic Elements (BLE) and Cluster-based Logic Blocks (CLB). As described in Fig. 1, each BLE contains one *K*-input LUT and one flip-flop (FF), and the LUT and FF share the same output. As described in Fig. 2, each CLB contains *N* BLEs, *I* inputs and *N* outputs. Each of the *I* inputs can drive all the BLEs, and each BLE drives an output. Here *K*, *N*, and *I* are parameters described by the architecture file. The interconnect delay between BLEs within the same CLB is usually much smaller than the delay between BLEs in different CLBs.

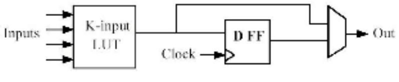


Fig. 1. *VPR*'s Basic Logic Element (BLE)

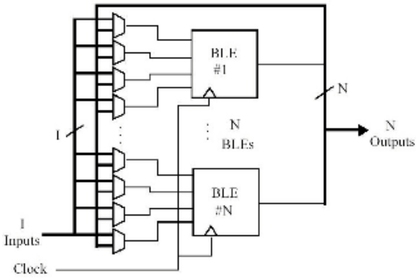
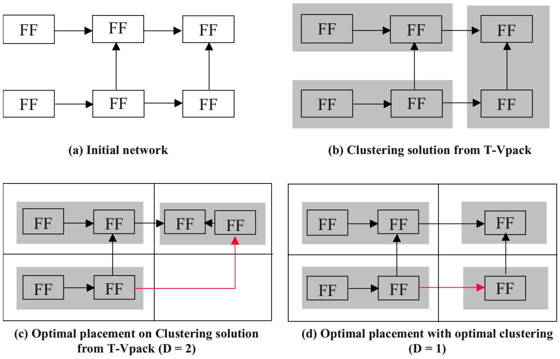


Fig. 2. *VPR*'s Cluster-Based Logic Block (CLB)

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 158–167, 2004.

In a typical FPGA design flow, a circuit is first synthesized and mapped into a netlist of LUTs and FFs. Then it goes through the following three steps: clustering, placement and routing. The clustering step arranges LUTs and FFs into CLBs according to the timing and the connectivity of the mapped netlist; the placement places the clustered netlist onto the array of on-chip CLBs; the routing routes all the wires in the netlist with the available routing resources on the device.

The drawback of this design flow is that the clustering and placement stages are artificially separated. During the clustering stage, we have great freedom to change a circuit's structure, but a fast and accurate estimation of the final placement wirelength, timing and routability information is not available. During the placement stage, we can optimize wirelength, timing and routability simultaneously, but the solution space is greatly confined because we are committed to a fixed circuit structure. Since the mistakes made during the clustering phase cannot be corrected during the placement process, it will ultimately generate a sub-optimal place and route result.



(a) Initial network

(b) Clustering solution from T-Vpack

(c) Optimal placement on Clustering solution from T-Vpack (D = 2)

(d) Optimal placement with optimal clustering (D = 1)

Fig. 3. Impact of Clustering on Placement

Fig. 3 illustrates the impact of clustering on placement. The initial network (a) consists of six FFs. We assume each CLB contains two BLEs, and the device is a 2 x 2 grid. The delay model used here is the Manhattan distance. The "optimal" clustering solution in (b), which consists of three CLBs and one logic level, can be obtained from *T-VPack* [17] (which minimizes the number of clusters and the number of levels). However, the optimal placement solution (c) on this optimal clustering has a longest path delay of two. Instead, when we perform clustering together with placement, we can obtain a placement solution (d) with a longest path delay of one.

In this paper, we propose a novel algorithm to perform clustering optimization during the placement for wirelength and timing minimization. We also incorporate a recently proposed path counting-based net weighting scheme in our approach. This new algorithm outperforms the current state-of-the-art FPGA placement flow *T-VPack + VPR* with an average reduction of up to 36% in total wirelength and 31% in longest path delay. Another significant contribution is that our combined approach has a runtime complexity similar to the existing *VPR* placement algorithm.

2 Review of Existing FPGA Clustering and Placement Algorithms

Packing LUTs and FFs into CLBs is a critical step in the cluster-based FPGA design flow, since it has a great impact on both timing and routability. *VPack* [17] packs each logic block to its capacity to minimize the number of clusters and encourages input sharing to minimize the number of connections between clusters. The timing-driven version, *T-Vpack* [17], minimizes the number of connections on the critical path since on average the internal connections are much faster than the external connections. *Rpack* [4] introduces an effective routability metric and presents a routability driven clustering algorithm for cluster-based FPGAs. *PRIME* [10] integrates retiming with performance-driven clustering/partitioning. For a given area bound for each cluster, if duplication is allowed, *PRIME* can generate a quasi-optimal solution with a delay of no more than a small constant over the minimal delay.

Placement is a classic problem and becomes increasingly difficult and important as the design size rapidly increases. There are three classes of widely used placement methods: min-cut based placer [11][5][23], analytical placer [12][15][20] and simulated annealing-based placer [14][21][1]. Min-cut based placers recursively partition the circuit until the number of cells in each partition is small enough and then assign cells to appropriate rows. The min-cut based methods are usually very fast, but since the cutsize is not an exact function of either wirelength, timing or routability, the quality is not as good as other placers. The analytical method includes the force directed and quadratic programming method. The force directed method introduces attracting, repelling and other additional forces and then solves a linear equation system using the forces. The quadratic programming (QP) method solves the placement problem by solving a sequence of quadratic programming problems derived from the circuit connectivity information. The force directed and quadratic-programming methods have a short runtime and produce good results, but they are not flexible enough to handle complex constraints. The simulated annealing algorithm simulates the annealing process that is used to produce high-quality metal structures by gradually cooling down the temperature. The initial placement is gradually optimized by performing a number of moves at each temperature. Each move is accepted with a certain probability $p = e^{-delta\_cost/T}$, where *delta_cost* is the change in cost function and T is the current temperature. Simulated annealing-based placers are very flexible for handling different kinds of constraints, and they usually generate a good solution in a reasonable amount of time. In recent years there have been several novel placement algorithms that incorporate multiple placement techniques. For example, *Mongrel* [13] adopts a *middle-down* methodology in which a *global placement* solution is obtained by placing logic cells into coarse bins. During the global placement phase, a *Relaxation Based Local Search* methodology is applied to generate global complex modifications to the current placement. A novel *ripple move* [13] based legalization procedure is also presented. After the global placement is completed, a detailed placement is obtained by applying the *optimal interleaving* [13] technique. *Dragon2000* [22] uses a top down hierarchical approach, and integrates the partitioning-based cutsize minimization techniques and the simulated annealing-based wirelength minimization techniques. mPL [6] and mPG [7] are based on the multilevel framework to improve both runtime and quality of the placement

3 Simultaneous Timing Driven Clustering and Placement Algorithm

3.1 Overview

Our algorithm uses a simulated annealing-based optimization engine [21][1][18]. We first perform an initial clustering on the mapped netlist, and then generate a random placement of the clustered netlist. During the annealing process, we optimize the clustering structure and circuit placement at the same time. To improve the suboptimal clustering structure during placement, we introduce a *fragment level move*. After each move, we update the cost function and decide whether to keep the move or not. We iteratively perform a certain number of moves at each temperature and then reduce the temperature until the acceptance rate is too low. In order to optimize both wirelength and circuit delay, we minimize a weighted function of bounding-box wirelength cost and timing cost (weighted edge delays). For the net weighting, we implement a recently proposed path counting-based net weighting scheme.

3.2 Clustering Optimization During Placement

Our main contribution is to perform clustering optimization during placement. There are two types of moves in our approach. The first type of move is the *block level move*, in which an entire CLB is moved to a new location and swapped with another CLB if necessary. The second type of move is the *fragment level move*, in which only a BLE is moved to a new CLB and swapped with another BLE if necessary. Due to the powerful *fragment level move*, we are able to significantly improve the suboptimal clustering structure to achieve a high quality placement. This is especially important when the chip utilization is high, and the clustering stage has to perform unrelated packing to squeeze the design into the device. Due to the lack of physical information, it is almost impossible for a clustering algorithm to make the right packing decisions among unrelated logics. With the simultaneous clustering optimization and placement optimization, we can correct mistakes made during the previous stage and significantly improve both routability and timing.

When we perform a *fragment level move*, we need to check whether the new CLB is in a valid configuration. When we check the feasibility of each CLB, we need to check the number of BLEs and the number of inputs. For real industry architectures, we also need to check the number of clocks, the number of feedbacks, the number of control signals, etc. Hence, we dynamically update a hash-map for each involved CLB whenever a *fragment level move* is performed. The complexity of the update is $O(K)$, where K is the input size of the LUT.

3.3 Path Counting-Based Net Weighting

The net-based timing-driven placers (e.g. [18]) convert timing information into net weight and optimize a weighted function of all nets. The basic idea of net weighting is to assign higher weights to timing critical nets and lower weights to non-critical nets. The net weighting scheme is both efficient and flexible enough to handle complex constraints, but most existing methods do not take into account the path information.

Here we incorporate a novel net weighting scheme [16] proposed by Dr. T. Kong, which accurately counts all paths (critical and non-critical) for certain types of discount functions such as $D(x, y) = a^{-x/y}$. This scheme considers path sharing, and thus assigns a higher weight to the edges shared by two or more critical paths. For more details about path counting, please refer to [16].

4 Runtime/Quality Trade-Off

For a given architecture, each CLB contains N BLEs, I inputs and N outputs. In the input clustered netlist, the number of CLBs is n, and the number of BLEs is m. $n \leq m \leq N{*}n$, and $O(m) = O(N{*}n) = N{*}O(n)$. If every swap performed at each temperature is at the BLE level, the number of swaps needed will be $O((N{*}n)^{4/3})$, which is quite costly.

However, we perform both block level move and fragment level move in our approach. At each temperature, the number of block level moves performed is $n^{4/3}$, and the number of fragment level moves performed is $(\alpha{*}m)^{1.33} \approx (\alpha{*}N{*}n)^{1.33}$. We can change the value of α between 0 and 1, and achieve the runtime/quality trade-off.

5 Complexity Analysis

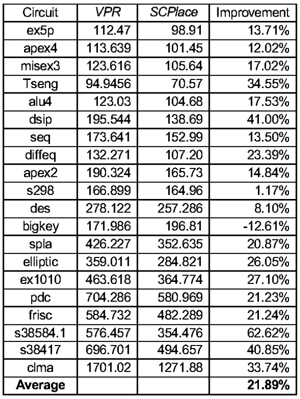
We first analyze the computation complexity of *VPR*'s placement engine T-VPlace [18]. The timing analysis is performed once per temperature change, which is an $O(n)$ operation. At each temperature the inner loop of the placer is executed $O(n^{4/3})$ times (i.e., $O(n^{4/3})$ swaps are performed). In the inner loop is the incremental-bounding-box-update operation that is worst case $O(k_{\max})$, where $k_{\max}$ is the fanout of the largest net in the circuit. The average case complexity for this bounding box update is $O(1)$ [2][3]. Also in the inner loop is the computation of the *Timing_Cost* for each connection affected by a swap. This is also $O(k_{max})$. In the average case this is $O(k_{avg})$ where k_{avg} is the average fanout of all nets in the circuit. Since k_{avg} is typically quite small, the average complexity of this *Timing_Cost* computation is $O(1)$ as well. The overall result is that the *VPR* algorithm is worst case $O[k_{max}\cdot(n)^{4/3}]$, but on average it is $O(n^{4/3})$. The average case complexity is really the only relevant value here. The complexity of the algorithm is the average over millions of swaps, so a user will always see the average case complexity.

In our algorithm *SCPlace*, at each temperature the complexity of the block level moves is $O(n^{4/3})$, and the complexity of the fragment level move is $O((\alpha*N*n)^{4/3})$. In reality, the value of N is not very big, and we can always choose α to make $O((\alpha*N*n)^{4/3}) = O(n^{4/3})$. Hence, the overall complexity is $O(n^{4/3}+n^{4/3}) = O(n^{4/3})$. As a result, our algorithm's complexity can be similar to *VPR*, and hence very scalable.

6 Experimental Results

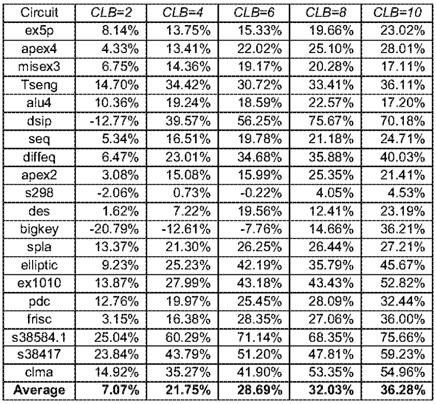
We implemented our algorithm *SCPlace* under the *VPR* framework. For the purpose of comparison, we downloaded the *VPR* 4.3 source code, architecture file and the complete set of 20 MCNC benchmark circuits used by *VPR* from [24]. We modified the architecture file to specify the number of BLEs contained in a single CLB. For all of the 20 MCNC circuits, we compare with the commonly used academic FPGA design flow [17]. We first run the *script.algebraic* in SIS [19], followed by *Flowmap* [9]. Then we run *T-VPack* [17] to generate an initial clustering solution. This initial clustering is then given to both *VPR* and *SCPlace* to perform placement. The default architecture we use assumes that each CLB contains 4 LUTs, and each LUT has 4 inputs. In section 6.1 and 6.2, we perform 100% fragment moves and no block moves. In section 6.3 we perform both block and fragment moves and explore the trade-off between quality and runtime. Only the runtime of the second half of benchmark set is reported since the circuits in the first half are too small.

Table 1. Wirelength Comparison with *T-VPack + VPR*



Circuit	*VPR*	*SCPlace*	Improvement
ex5p	112.47	98.91	13.71%
apex4	113.639	101.45	12.02%
misex3	123.616	105.64	17.02%
Tseng	94.9456	70.57	34.55%
alu4	123.03	104.68	17.53%
dsip	195.544	138.69	41.00%
seq	173.641	152.99	13.50%
diffeq	132.271	107.20	23.39%
apex2	190.324	165.73	14.84%
s298	166.899	164.96	1.17%
des	278.122	257.286	8.10%
bigkey	171.986	196.81	-12.61%
spla	426.227	352.635	20.87%
elliptic	359.011	284.821	26.05%
ex1010	463.618	364.774	27.10%
pdc	704.286	580.969	21.23%
frisc	584.732	482.289	21.24%
s38584.1	576.457	354.476	62.62%
s38417	696.701	494.657	40.85%
clma	1701.02	1271.88	33.74%
Average			**21.89%**

Table 2. Impact of Architecture on Wirelength



Circuit	*CLB=2*	*CLB=4*	*CLB=6*	*CLB=8*	*CLB=10*
ex5p	8.14%	13.75%	15.33%	19.66%	23.02%
apex4	4.33%	13.41%	22.02%	25.10%	28.01%
misex3	6.75%	14.36%	19.17%	20.28%	17.11%
Tseng	14.70%	34.42%	30.72%	33.41%	36.11%
alu4	10.36%	19.24%	18.59%	22.57%	17.20%
dsip	-12.77%	39.57%	56.25%	75.67%	70.18%
seq	5.34%	16.51%	19.78%	21.18%	24.71%
diffeq	6.47%	23.01%	34.68%	35.88%	40.03%
apex2	3.08%	15.08%	15.99%	25.35%	21.41%
s298	-2.06%	0.73%	-0.22%	4.05%	4.53%
des	1.62%	7.22%	19.56%	12.41%	23.19%
bigkey	-20.79%	-12.61%	-7.76%	14.66%	36.21%
spla	13.37%	21.30%	26.25%	26.44%	27.21%
elliptic	9.23%	25.23%	42.19%	35.79%	45.67%
ex1010	13.87%	27.99%	43.18%	43.43%	52.82%
pdc	12.76%	19.97%	25.45%	28.09%	32.44%
frisc	3.15%	16.38%	28.35%	27.06%	36.00%
s38584.1	25.04%	60.29%	71.14%	68.35%	75.66%
s38417	23.84%	43.79%	51.20%	47.81%	59.23%
clma	14.92%	35.27%	41.90%	53.35%	54.96%
Average	**7.07%**	**21.75%**	**28.69%**	**32.03%**	**36.28%**

6.1 Wire-Length Comparison

In Table 1, we compare our algorithm *SCPlace* with *VPR* using the total weighted bounding box wire lengths as the only optimization objective. The weights for nets of different sizes can be found in [8]. When we combine clustering with placement, we can outperform *VPR* by 22% on average.

In Table 2, we illustrate the impact of architecture on the wirelength improvement obtained from *SCPlace*. When we change the size of the CLB (*N)* from 2 to 10, the wirelength gap between *SCPlace* and *T-Vpack+VPR* increases monotonically from 7% to 36%. The result shows that as the size of CLB increases, it is more and more difficult to generate a good clustering solution with small wirelength without physical information. Since *SCPlace* explores different clustering solutions during the placement stage, it generates clustering and placement solutions with much shorter wirelength.

Table 3. Timing Comparison with *T-VPack + VPR*

Circuit	*VPR*	*Path count*	%	*Fragment*	%	*Frag +path count*	%
ex5p	50.45	44.95	12.24%	44.65	12.99%	40.75	23.80%
apex4	47.44	44.71	6.12%	44.02	7.77%	41.44	14.49%
misex3	51.04	44.15	15.61%	43.91	16.24%	38.53	32.47%
tseng	38.85	36.43	6.65%	35.89	8.24%	35.11	10.65%
alu4	53.16	45.46	16.95%	47.51	11.89%	42.50	25.07%
dsip	38.32	40.96	-6.45%	38.32	0.00%	40.12	-4.49%
seq	51.26	46.56	10.11%	43.97	16.59%	42.90	19.51%
diffeq	47.73	38.76	23.15%	39.58	20.60%	41.15	16.01%
apex2	56.36	50.97	10.58%	52.37	7.62%	47.23	19.34%
s298	87.36	90.76	-3.74%	89.31	-2.18%	80.98	7.88%
des	83.88	67.15	24.91%	74.39	12.75%	65.44	28.18%
bigkey	41.37	40.95	1.03%	43.35	-4.57%	41.35	0.03%
spla	72.47	63.12	14.81%	64.09	13.07%	58.27	24.35%
elliptic	71.07	55.72	27.54%	59.17	20.11%	48.49	46.58%
ex1010	97.88	79.10	23.75%	86.85	12.70%	74.82	30.81%
pdc	113.15	76.95	47.04%	78.44	44.24%	67.60	67.38%
frisc	81.39	92.53	-12.0%	79.82	1.97%	75.73	7.47%
s38584.1	64.37	46.66	37.96%	56.93	13.07%	47.78	34.71%
s38417	76.63	70.15	9.24%	56.89	34.71%	49.89	53.61%
clma	137.20	116.8	17.52%	113.8	20.61%	102.0	34.52%
Average			14.15%		13.42%		24.62%

Table 4. Impact of Architecture on Timing

Circuit	*CLB=2*	*CLB=4*	*CLB=6*	*CLB=8*	*CLB=10*
ex5p	8.69%	23.80%	23.04%	17.09%	13.71%
apex4	5.82%	14.49%	16.79%	24.06%	23.25%
misex3	11.19%	32.47%	34.07%	30.73%	14.08%
Tseng	1.71%	10.65%	0.16%	22.32%	20.69%
alu4	22.95%	25.07%	28.02%	15.56%	20.23%
dsip	5.36%	-4.49%	13.35%	-6.06%	7.40%
seq	18.38%	19.51%	35.20%	27.04%	27.26%
diffeq	4.93%	16.01%	12.58%	30.15%	6.21%
apex2	11.51%	19.34%	25.09%	16.86%	18.39%
s298	-4.20%	7.88%	1.10%	-6.83%	4.40%
des	5.82%	28.18%	7.48%	38.16%	17.10%
bigkey	11.87%	0.03%	-7.81%	27.89%	0.61%
spla	40.52%	24.35%	39.41%	28.42%	40.85%
elliptic	31.94%	46.58%	19.21%	23.02%	33.41%
ex1010	23.26%	30.81%	28.96%	41.19%	27.86%
pdc	31.40%	67.38%	44.15%	59.63%	45.95%
frisc	6.95%	7.47%	5.21%	-4.52%	3.81%
s38584.1	23.67%	34.71%	-1.80%	-0.68%	20.86%
s38417	53.19%	53.61%	45.03%	70.61%	41.38%
clma	26.05%	34.52%	60.14%	58.87%	79.80%
Average	**17.05%**	**24.62%**	**21.47%**	**25.68%**	**23.36%**

6.2 Timing Comparison

In Table 3, we compare *SCPlace* with both *VPR* and *TTT* [16] in timing optimization. If we use path counting-based net weighting scheme only in *SCPlace*, we can outperform *VPR* by 14% (column 4); if we perform clustering optimization only in *SCPlace*, we can outperform *VPR* by 13% (column 6); if we integrate the path counting-based net weighting scheme with the clustering optimization, *SCPlace* significantly outperforms the original *VPR* result by 25%.

In Table 4, we illustrate the impact of architecture on the delay improvement obtained from *SCPlace*. For architecture with the CLB size of 2, the timing gap between *SCPlace* and *T-Vpack+VPR* is 17%. When the size of the CLB (*N)* increases from 4 to 10, the timing gap between *SCPlace* and *T-Vpack+VPR* remains in a narrow range between 22 to 25%. The result shows that even when the CLB size is relatively small (2 or 4), it is difficult to generate a good clustering solution with small delay without physical information. Since *SCPlace* explores different clustering solutions during the placement stage, it generates clustering and placement solutions with much better delay.

Table 5. Effect of α on timing (CLB = 4)

	α=0.25		*α=0.50*		*α=1.0*	
Circuit	Timing	runtime	Timing	runtime	Timing	runtime
des	24.47%	26.15%	28.29%	38.13%	32.64%	70.28%
bigkey	-10.20%	30.42%	16.00%	42.59%	7.18%	72.58%
spla	27.09%	41.17%	34.99%	49.39%	28.57%	76.59%
elliptic	51.06%	42.89%	48.61%	51.12%	49.63%	73.86%
ex1010	29.69%	36.08%	31.24%	41.53%	31.73%	64.08%
pdc	58.75%	32.75%	69.24%	39.72%	86.41%	58.41%
frisc	0.66%	33.61%	-0.72%	40.23%	7.33%	60.61%
s38584.1	43.58%	26.62%	47.86%	32.53%	34.81%	47.41%
s38417	27.05%	32.01%	53.17%	37.47%	60.67%	59.77%
clma	27.80%	25.82%	38.38%	31.75%	41.08%	48.29%
Average	**21.59%**	**32.75%**	**31.18%**	**40.45%**	**30.83%**	**63.19%**

Table 6. Effect of α on timing (CLB = 10)

	α=0.25		*α=0.50*		*α=1.0*	
Circuit	Timing	runtime	Timing	runtime	Timing	runtime
des	16.59%	24.53%	19.22%	36.35%	20.88%	71.59%
bigkey	-4.34%	36.24%	-9.78%	54.49%	2.95%	98.44%
spla	33.13%	61.99%	43.21%	74.56%	40.32%	121.37%
elliptic	50.30%	45.57%	54.03%	47.84%	52.97%	72.16%
ex1010	38.80%	54.47%	26.00%	61.75%	32.12%	98.68%
pdc	49.99%	50.79%	52.58%	60.38%	54.95%	95.26%
frisc	-2.29%	46.78%	10.42%	57.29%	14.86%	88.07%
s38584.1	30.30%	31.73%	39.66%	33.08%	39.01%	52.39%
s38417	22.18%	43.91%	41.60%	50.76%	40.63%	81.65%
clma	54.89%	28.85%	71.51%	36.69%	83.96%	58.08%
Average	**20.18%**	**42.49%**	**27.05%**	**51.32%**	**31.25%**	**83.77%**

Table 7. Routed Delay and Track Count Comparison

	VPR		*SCPlace*		%	
Circuit	Routed delay	#tracks	Routed delay	#tracks	Routed delay	#tracks
ex5p	52.38	646	45.47	627	15.20%	3.03%
apex4	55.93	627	46.32	665	20.75%	-5.71%
misex3	56.56	588	40.92	588	38.22%	0.00%
tseng	41.08	483	36.35	437	13.01%	10.53%
alu4	55.16	594	47.47	506	16.20%	17.39%
dsip	38.80	935	35.73	660	8.59%	41.67%
seq	58.13	744	49.12	768	18.34%	-3.13%
diffeq	50.41	506	39.57	506	27.39%	0.00%
apex2	58.00	775	48.03	725	20.76%	6.90%
s298	103.69	648	89.43	621	15.95%	4.35%
des	88.32	960	69.26	832	27.52%	15.38%
bigkey	42.60	495	48.40	550	-11.98%	-10.00%
spla	78.65	1452	67.68	1287	16.21%	12.82%
elliptic	75.16	1156	62.14	1054	20.95%	9.68%
ex1010	102.88	1188	81.58	1008	26.11%	17.86%
pdc	125.46	2028	93.18	1755	34.64%	15.56%
frisc	87.64	1560	127.02	1600	-31.00%	-2.50%
s38584.1	66.41	1276	47.02	924	41.24%	38.10%
s38417	81.54	1410	54.15	1128	50.58%	25.00%
clma	144.02	2760	124.14	2040	16.01%	35.29%
Average					**19.23%**	**11.61%**

6.3 Runtime Speedup

For a given architecture, each CLB contains N BLEs, I inputs and N outputs. In the input clustered netlist, the number of CLBs is n, and the number of BLEs is m, and $m \approx N*n$. From Table 1 to Table 4, we perform $m^{1.33} \approx (N*n)^{1.33}$ fragment moves and 0 block moves. In this section, we fix the number of block moves to be $n^{1.33}$, and set the number of fragment moves to be $(\alpha*m)^{1.33} \approx (\alpha*N*n)^{1.33}$, where α is between 0 and 1.

In Table 5, we show the impact of α on the amount of timing improvement achievable. It is no surprise that when α increases, i.e., the number of fragment moves increase, the timing improvement increases from 22% to 31%. And this is better than the 25% we achieve in Table 4 when we perform fragment moves only. The results illustrate that performing both block and fragment moves is better than only performing one type of moves. Our runtime is generally shorter than *VPR* due to the fact that the number of block moves we perform is only 10% of *VPR*'s. If we reduce the number of block moves *VPR* performs to be the same as *SCPlace*, it yields about 5% worse result (both timing and wirelength) and consumes 15% of standard *VPR* 's runtime. When α = 0.25*, SCPlace* uses 33% of standard *VPR* 's runtime. *SCPlace*'s runtime increases up to 63% as α increases to 1. Table 6 shows the same trend when the size of the CLB is 10. The bottom line is that you could easily tradeoff runtime with quality by changing the value of α.

6.4 Routed Results

In Table 7, we show the comparison of routed delay and track count between *SCPlace* and *T-Vpack+VPR*. The given architecture has a CLB size of 4, and the *SCPlace* run is from Table 5 when $\alpha = 0.50$. The routed delay improvement is 19% on average and the reduction in routed tracks is 12% on average. This is consistent with the estimated delay/wirelength reduction after placement.

7 Conclusions

We introduce a novel simultaneous clustering and placement algorithm and incorporate a novel path counting-based net weighting scheme. The new algorithm produces impressive results for both bounding box wire length optimization and timing optimization. When compared with the state-of-the-art separate FPGA design flow *T-VPack + VPR*, our algorithm improves up to 36% in wirelength and 31% in longest path delay. Since our algorithm has a similar computational complexity, our approach is also very scalable.

Acknowledgement. This research is partially supported by NSF Grant CCF-0096383.

References

[1] V. Betz and J. Rose, “VPR: A New Packing, Placement and Routing Tool for FPGA Research,” *International Workshop on Field Programmable Logic and Application*, pp. 213-222, 1997

[2] V. Betz, “Architecture and CAD for Speed and Area Optimization of FPGAs,” *Ph. D. Dissertation*, *University of Toronto*, 1998

[3] V. Betz, J. Rose and A. Marquardt, “Architecture and CAD for Deep-Submicron FPGAs,” *Kluwer Academic Publishers*, February 1999

[4] E. Bozorgzadeh, S. Ogrenci and M. Sarrafzadeh, "Routability-Driven Packing for Cluster-Based FPGAs," *ASPDAC*, Yokohama, Japan, 2001

[5] Andrew E. Caldwell, Andrew B. Kahng and Igor L. Markov, “Can Recursive Bisection Alone Produce Routable Placements?” *ACM/IEEE Design Automation Conference*, pp. 477–482, 2000

[6] T. Chan, J. Cong, T. Kong and J. Shinnerl, "Multilevel Optimization for Large-scale Circuit Placement," *Proc. IEEE International Conference on Computer Aided Design*, San Jose, California, pp. 171-176, November 2000

[7] C.-C. Chang, J. Cong, D. Pan, and X. Yuan, "Multilevel Global Placement with Congestion Control," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 22, no. 4, pp. 395-409, July 2002

[8] C. Cheng, “RISA: Accurate and Efficient Placement Routability Modeling,” *IEEE/ACM International Conference on Computer-Aided Design*, pp. 690-695, 1994

[9] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs", *IEEE Trans. on Computer-Aided Design*, vol. 13, no. 1, pp. 1-12, January 1994

[10] J. Cong, H. Li and C. Wu "Simultaneous Circuit Partitioning/Clustering with Retiming for Performance Optimization," *Proc. 36th ACM/IEEE Design Automation Conf.*, New Orleans, Louisiana, pp. 460-465, June 1999

[11] A. Dunlop and B. Kernighan, “A Procedure for Placement of Standard-Cell VLSI Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4:92-98, January 1985

[12] H. Eisenmann and F.M. Johannes, “Generic Global Placement and Floorplanning,” *ACM/IEEE Design Automation Conference*, pp. 269-274, 1998

[13] S-W Hur and J. Lillis, “Mongrel: Hybrid Techniques for Standard Cell Placement,” *IEEE/ACM International Conference on Computer-Aided Design*, pp 165-170, 2000

[14] S.S. Kirkpatrick, C. Gelatt and M. Vecchi, “Optimization by Simulated Annealing,” *Science*, pp. 671-680, May 13, 1983

[15] J.M. Kleinhans, G. Sigl, F.M. Johannes and K.J. Antreich, “GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10:356-365, 1991

[16] T. Kong, “A Novel Net Weighting Algorithm for Timing-driven Placement,” *IEEE/ACM International Conference on Computer-Aided Design*, pp. 172-176, 2002

[17] A. Marquardt, V. Betz and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, pp. 37-46, 1999

[18] A. Marquardt, V. Betz and J. Rose, ``Timing-Driven Placement for FPGAs," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, pp. 203 – 213, February 2000

[19] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, “SIS: A System for Sequential Circuit Synthesis,” *Electronics Research Laboratory*, Memorandum No. UCB/ERL M92/41, 1992

[20] G. Sigl, K. Doll and F.M. Johannes, “Analytical Placement: A Linear or a Quadratic Objective Function?” *ACM/IEEE Design Automation Conference*, pp. 427-432, 1991

[21] W. Swartz and C. Sechen, “Timing Driven Placement for Large Standard Cell Circuits,” *ACM/IEEE Design Automation Conference*, pp. 211-215, 1995

[22] M. Wang, X. Yang and M. Sarrafzadeh, “Dragon2000: Standard-Cell Placement Tool For Large Industry Circuits,” *IEEE/ACM International Conference on Computer-Aided Design*, pp. 260-263, 2000

[23] K. Zhong and S. Dutt, “Effective Partitioning-Driven Placement with Simultaneous Level Processing and Global Net Views,” *Proc. IEEE International Conference on Computer Aided Design*, San Jose, California, pp. 254-259, November 2000

[24] http://www.eecg.toronton.edu/~vaughn/challenge/challenge.html, “The FPGA Place-and-Route Challenge”

Run-Time-Conscious Automatic Timing-Driven FPGA Layout Synthesis

Jason Anderson[1], Sudip Nag[2*], Kamal Chaudhary[2], Sandor Kalman[2],
Chari Madabhushi[2], and Paul Cheng[3]

[1] Xilinx Inc., Toronto, ON, Canada
[2] Xilinx Inc., San Jose, CA, USA
[3] Xilinx Inc., Longmont, CO, USA
{janders,kamal,sandor,chari,pcheng}@xilinx.com

Abstract. Layout tools for FPGAs can typically be run in two different modes: non-timing-driven and timing-driven. Non-timing-driven mode produces a solution quickly, without consideration of design performance. Timing-driven mode requires that a designer specify performance constraints and then produces a performance-optimized layout solution. The task of generating constraints is burdensome since design performance is difficult to gauge at the pre-layout stage and the relationship between the constraints supplied and tool execution time is unpredictable. In this paper, we propose a new mode for layout tools, called "automatic timing-driven" mode that produces a performance-optimized layout, without requiring any constraint specification. A key feature of this mode is a novel and practical method for automatic constraint generation that creates constraints that result in *predictable* and *controlled* layout execution time. The automatic constraint generation approach has been integrated into commercial FPGA layout tools and tuned to provide layouts having 28% better performance than non-timing-driven mode, on average. Results show that the ratio of the automatic to non-timing-driven layout execution time is consistent and predictable across a suite of designs.

1 Introduction

State-of-the-art field-programmable gate arrays (FPGAs) can implement systems with millions of gates that operate at speeds in the hundreds of megahertz. An important part of the FPGA CAD flow is the layout step, comprised of placing and routing a design on a target FPGA device. Place and route systems for FPGAs typically support two modes of operation: non-timing-driven and timing-driven. The former mode ignores delays, producing a valid placed and routed design as quickly as possible. Since delays are ignored, the resultant solution can have poor performance characteristics. In contrast, timing-driven mode typically requires that the user (designer) provide performance constraints. Layout tools then perform delay optimization in order to meet the specified constraints.

[*] Sudip Nag was with Xilinx when this work was conducted and is now with Flexlogics.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 168–178, 2004.

In this paper, we propose a new mode of operation for FPGA layout tools in which a performance-optimized layout solution is produced without the need for a user to provide performance constraints. A relevant question that arises relates to the definition of "performance". The aspect of performance we choose for automatic optimization is a design's clock frequency, which represents the most common optimization goal in digital circuit design. We refer to the new layout mode as *automatic timing-driven layout*.

The proposed automatic mode offers a number of benefits. First, as we will demonstrate, the layouts produced by the automatic mode have better performance than those produced by non-timing-driven mode. Often, the automatic layout performance will be sufficient to meet design requirements and no constraint specification and additional layout passes will be needed. Second, and perhaps most important, the automatic mode offers *predictable* and *controlled* layout execution time. In particular, the automatic approach can be tuned to produce performance-driven layout solutions in execution times that are consistently longer by a *pre-specified* percentage than those associated with non-timing-driven layout. Thus, its aim is to offer the best possible performance within a given execution time. This differs considerably from other automatic performance-driven FPGA layout systems, such as [1], which attempt to optimize design performance to the maximum extent possible. Rather, our approach provides a user with a reasonably good performance-optimized layout, as well as offers the run-time predictability that is a crucial part of a quality "push-button" tool experience – a primary concern for commercial FPGA tool vendors.

To optimize performance in today's high-speed FPGAs, users employ a relatively "ad hoc" process. A trial-and-error approach is typically used for constraint generation whereby a user specifies performance constraints, executes layout tools and then analyzes the results. If the constraints are met, new and more aggressive constraints are specified and the process is repeated. On the other hand, if constraints are not met, long and unpredictable tool run-times are incurred, the constraints must be relaxed and layout tools re-executed. Each iteration of this process can take hours or days for a large design, leading to a lengthy design cycle and increased cost. An important application of the proposed automatic approach is as an initial layout pass that requires no user intervention and takes limited execution time. The resultant solution can then be used as a starting point for further optimization, leading to a reduction in the number of passes through the design flow.

One of the key contributions of this paper is a novel method for performance constraint generation that chooses design constraints in a way that is conscious of layout execution time. The approach works by doing a careful analysis of the distribution of delay slacks in circuits and produces realistic and feasible performance constraints. The aggressiveness of the generated constraints and the related layout execution time are managed elegantly through a single parameter. The proposed automatic approach has been integrated into a commercial FPGA layout system and used for performance optimization of industrial FPGA designs. The rest of this paper is organized as follows. In Section 2, we present preliminary background material and discuss related work. The automatic layout approach is described in Section 3. Experimental results are given in Section 4. Conclusions are offered in Section 5.

2 Background

For the purpose of static timing analysis, the combinational part of a digital circuit can be represented using a timing graph, $G(V, E)$, which is a directed acyclic graph (DAG) with nodes, V, corresponding to circuit blocks and edges (connections), E, between nodes corresponding to electrical connections between blocks. Delay values are associated with each node and edge in a circuit's timing graph. Let $input(v)$ and $output(v)$ represent the predecessors and successors of a node v ($v \in V$), respectively. A node v with $input(v) = \emptyset$ corresponds to a primary input or register output. Similarly, a node v with $output(v) = \emptyset$ corresponds to a primary output or register input. The maximum performance of a layout solution (clock frequency) is limited by the delay of the longest-delay path in the timing graph, as given by:

$$T_{period} = \max_{\pi \epsilon \Pi} \{ \sum_{b \ \epsilon \ nodes(\pi)} Delay(b) + \sum_{c \ \epsilon \ edges(\pi)} Delay(c) \} \tag{1}$$

where $nodes(\pi)$ and $edges(\pi)$ represents the circuit blocks and connections on a path π, respectively, and Π represents the set of *all* paths in the timing graph. An important property of a combinational path π is its *slack*, which is defined to be the difference between the path's required delay (RT_π) and its actual delay (AT_π):

$$slack(\pi) = RT_\pi - AT_\pi \tag{2}$$

The required delays for paths are generally fixed by user constraints specified prior to layout synthesis. Paths with negative slack are referred to as *critical*; such paths have excessive delay that must be reduced if performance constraints are to be met. Layout tools usually operate at the level of individual driver/load connections (edges) rather than entire paths. The slack of a connection, c, is defined to be the *minimum* slack of any path through c:

$$slack(c) = \min_{\pi \epsilon \Pi_c} \{slack(\pi)\} \tag{3}$$

where Π_c represents the set of all paths through connection c. An early work by Hitchcock et. al. showed how path delays, path slacks and connection slacks can be computed in $O(|V|)$ time [2]. Other related work has focused on distributing the slack of a path amongst its constituent connections to yield an upper bound on the allowable delay of each connection, for use by layout tools [3,4,5].

Substantial research has been dedicated to performance-driven layout synthesis for FPGAs (e.g., [1,6,7,8]). Generally, the approach taken has been to optimize performance by giving higher priority to connections with negative slack, versus non-critical connections. The priority notion can have different meanings, depending on the context in which it is applied. In the placement phase for example, nets with negative slack connections are given priority for wirelength and/or delay minimization, which can be estimated using metrics such as half-perimeter bounding box length. In the routing phase, steps can be taken to ensure that negative slack connections are given preference for allocation of low-delay FPGA routing resources [7].

3 Automatic Timing-Driven Layout Synthesis

In this section, we present the proposed automatic timing-driven layout approach. We first describe how performance constraints are automatically generated and subsequently, we outline how the constraint generation process is integrated into layout tools.

3.1 Constraint Generation

A straight forward way to handle automatic constraint generation is to analyze the circuit prior to placement and routing, estimate the achievable clock frequency and then perform timing-driven placement and routing with the estimated target. The problem with this approach is that the ability to predict performance at the pre-layout stage is quite limited. In FPGAs, path delay is often dominated by interconnect delay rather than logic block delay. Interconnect delays are difficult to predict and are known accurately only *after* layout is complete. In pre-layout constraint estimation/generation, if the estimated design frequency is too low, then the resultant design performance will be far from the best achievable. Conversely, if the estimated performance is too aggressive, long and unpredictable layout run-time will be incurred and the target frequency may not be feasible. To address these issues, in our approach, we dynamically adjust the performance target throughout the flow, based on the current layout status.

Our constraint generation approach is based on the crucial observation that the run-time of layout tools is highly dependent on the number of critical connections rather than the absolute performance target. We will demonstrate this assertion in our experimental results; however, it makes sense intuitively since greater numbers of critical connections imply greater competition for the FPGA resources with the best delay characteristics, leading to more complex tradeoffs and increased run-time. Furthermore, critical connections must be routed in a “delay-driven” manner. Delay-driven routing is compute-intensive since it involves detailed RC delay calculations, for example using [9] or [10], to find minimum delay paths through the FPGA routing fabric from a critical connection’s source to load pin. Thus, a good constraint generation approach must be cognizant of how many connections are made critical by the constraints imposed.

The goal of tightly controlling the number of critical connections eliminates the possibility of employing a “naive” constraint generation approach, which uses an intermediate layout solution’s maximum path delay to derive a performance constraint. For example, consider the timing graphs for two circuits shown in Fig. 1(a)[1]. The top of the figure shows a timing graph for a circuit with four

[1] Block delays are assumed to be zero in these examples.

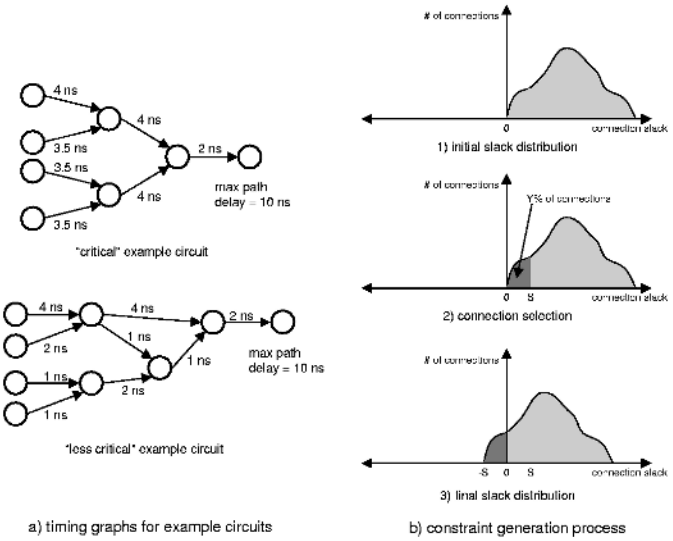


Fig. 1. (a) Timing graphs; (b) Constraint generation.

paths. The maximum path delay is 10 *ns*. The bottom of the figure shows a timing graph for a circuit with six paths. The maximum path delay is also 10 *ns*. If, for example, we generated path delay constraints by taking 90% of the maximum path delay, then the maximum allowable path delay for both circuits would be constrained to 9 *ns*. For the circuit in the top of Fig. 1(a), this would result in *all* its paths and connections becoming critical, since the delays of all paths in this circuit lie between 9 and 10 *ns*. On the other hand, a 9 *ns* constraint applied to the circuit in the bottom of Fig. 1(a) would result in only a single path with 3 connections being critical. Thus, the naive approach lacks control over the number of connections that become critical and is unsuitable for automatic constraint generation.

The novel aspect of our constraint generation approach is that it selects a performance target in a way that carefully manages the number of critical connections. Fig. 1(b) gives an abstract view of the constraint generation process. The input to the process is a delay for each connection in the design being optimized. At the placement stage, connection delay estimates are used; at the routing stage, accurate delays are available. Let T_{layout} be the maximum delay of any path in the current layout. We begin by computing the slacks of all connections in the circuit based on a performance constraint of T_{layout}. Part 1 of Fig. 1(b) shows the distribution of connection slacks at this stage (top of figure). The horizontal axis represents connection slack; the vertical axis represents the number of connections having a given slack. Observe that since slacks are computed based on a constraint that is equal to the current maximum path delay, there are no connections with negative slack.

Our goal is to choose a constraint that results in a specific fraction of connections becoming critical. In this example, assume we aim to make $Y\%$ of the design's connections critical. Larger values for Y imply increasingly hard-to-meet delay constraints and longer run-times. We analyze the slack distribution to identify the slack, S, such that $Y\%$ of connections have a slack less than or equal to S. This set of connections is represented by the dark, shaded region in part 2 of Fig. 1(b). We set the new path delay constraint, $T_{constraint}$, for the circuit as follows:

$$T_{constraint} = T_{layout} - S \tag{4}$$

The new slack distribution, based on a constraint of $T_{constraint}$, is shown in in part 3 of Fig. 1(b). Precisely $Y\%$ of connections are critical in the layout solution for constraint $T_{constraint}$. This form of constraint determination is used throughout the layout flow, with varying values for parameter Y. Note that in this discussion, we have restricted ourselves to designs having a single clock. However, extending the approach to designs with multiple clock domains is straight forward as slack distributions and period constraints can be generated independently for each clock domain in the same manner.

Observe that because our constraint generation approach is connection-based, it does not require enumerating the paths in a circuit (an operation with exponential time complexity). Rather, the proposed approach simply involves determining connection slacks using [2] and then "binning" the slacks to generate a histogram, similar to those in Fig. 1(b). Histogram generation and connection slack computation are both $O(|V|)$ operations; therefore, the complexity of the constraint generation approach is $O(|V|)$.

3.2 Integration into Layout Tools

We implemented the constraint generation approach and integrated it into a commercial FPGA layout system. The specific algorithms used in the system are proprietary; however, the constraint generation procedure described above is not limited for use with any particular layout algorithm. The placement step of the layout system proceeds in phases. Early phases optimize the design at a high-level of abstraction, permitting large changes in the placement solution. Later phases are mainly concerned with finer-grained placement refinement. Each placement phase prioritizes connections based on their delay slacks and the most critical connections are afforded preference for wirelength minimization. Prior to each placement phase, the circuit's interconnect delays are estimated and annotated onto the timing graph. At this point, the automatic constraint generation process is invoked. We empirically determined that setting parameter Y to 3% (3% of connections are made critical) achieves our objective of producing reasonably good quality performance-optimized layout solutions in a predictable and relatively low execution time. The placement phase then commences with consideration of the newly generated constraint (and its associated connection slacks). We have observed that this approach results in successively aggressive, but realistic constraints as placement proceeds through its phases.

The routing tool operates in two phases. In the initial phase, a design's connections are routed without timing considerations. The goal is to produce a routing solution with minimal resource usage. After initial routing, automatic constraint generation is executed and 2% of connections are made critical. To meet the constraint, critical connections are re-routed in a delay-driven manner and their delays are reduced. If, after delay-driven routing, the constraint is met, the constraint generation processes is invoked again with parameter $Y = 2\%$. This causes 2% more connections to be added to the total pool of connections that must be routed in delay-driven mode. The iterative process of successive constraint generation and delay-driven routing continues until one of two conditions is true: (1) a performance constraint that is difficult or impossible to meet is identified, or (2) a fixed iteration limit is exceeded (3 iterations). We have observed that router run-time is highly sensitive to the number of connections that must be routed in delay-driven mode. Run-time considerations are managed through condition (2), which places strict limits on the fraction of a design's connections that are permitted to become critical. Following constraint generation and delay-driven routing, the final routing phase proceeds, which refines the initial routing solution and removes any remaining infeasibilities.

One of the elegant features of our approach is that no timing constraint or performance data is passed between the various stages of the layout tool, for example, between the placer and the router. Instead, the various phases dynamically determine the performance of the current layout and a performance target is chosen accordingly. This greatly simplified the integration of the proposed approach into existing timing-driven layout tools.

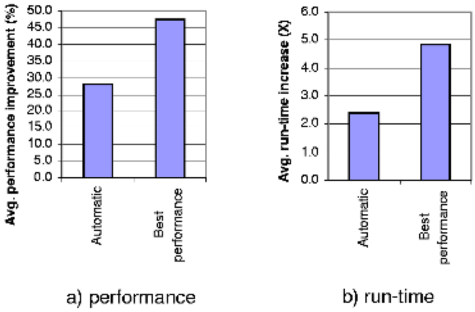
A second attractive feature of the approach is that the "aggressiveness" of the constraint generation process is controlled by a single parameter (Y). As discussed above, specific values for Y have been selected for use in our layout framework. This particular tuning reflects performance results and execution times that we believe to be acceptable to users of the automatic timing-driven flow. Of course, higher (lower) values for Y will lead to better (worse) performance at the expense of longer (shorter) tool execution time. In the next section, we validate our choices for this parameter experimentally through an analysis of tool run-time and layout quality.

4 Experimental Study

To evaluate our approach, we compare it with the layout solutions that represent the extremes in the run-time/performance trade-off space. Specifically, we compare the automatic approach with two different scenarios: *non-timing-driven* layout and *best performance* layout.

In the non-timing-driven scenario, the placement and routing tools are run without a performance objective. A layout solution is generated in as little time as possible, without regard for design performance. In the best performance scenario, layout tools are run with a difficult-to-meet performance objective (clock frequency constraint), which must be selected individually for each design. We determined the performance objective for each design in an iterative manner by starting with an easy-to-meet objective and then increasing it gradually until eventually, a performance objective that could not be met was discovered. The highest, meet-able objective was then selected as the performance objective for the best performance layout scenario.

In our experiments, we use 35 industrial benchmark circuits collected from Xilinx® customers and target a popular commercial FPGA (Xilinx Virtex^{TM}-II) [11]. The primary combinational logic element in the target FPGA is a 4-input look-up-table, which is a small memory capable of implementing any logic function requiring less than or equal to 4 inputs. FPGA logic blocks are referred to as slices; each slice contains two 4-input look-up-tables, two registers as well as arithmetic and other circuitry. The sizes of the circuits in our study range from 109 slices to 14334 slices. The FPGA's interconnection network is comprised of variable length wire segments that connect to one another through programmable buffered switches.



a) performance

b) run-time

Fig. 2. (a) Performance vs. non-timing-driven; (b) run-time vs. non-timing-driven.

4.1 Experimental Results

We begin by summarizing the performance of the three layout solutions. Fig. 2(a) shows the average percentage improvement in clock frequency for the automatic approach and the best performance solution versus the non-timing-driven layout solution. The average performance improvement offered by the automatic solution is about 28%. The best performance layout solutions have clock frequencies that are 48% faster than the non-timing-driven solutions, on average. The results underscore the huge benefits of timing-driven layout: the performance improvements over non-timing-driven amount to the equivalent of several speed grades. Note that the disparity in performance between the automatic and best performance layouts simply reflects our tuning preferences, described in Section 3.2. Below, we show that other tunings are also possible.

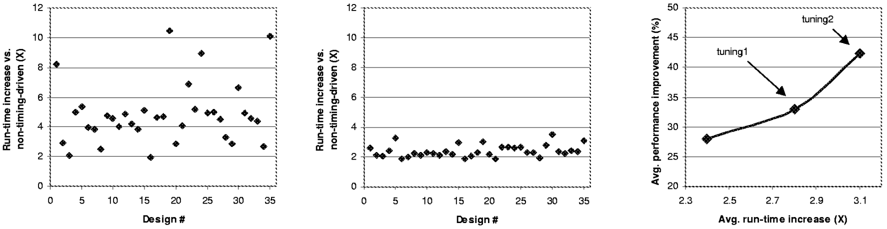
Fig. 2(b) summarizes the run-time results. The average increase in the layout tool's run-time is shown for the automatic solution and the best performance solution, in comparison with the non-timing-driven solution. Observe that the run-time hit associated with both forms of performance-driven optimization is considerable. On average, the run-time of automatic timing-driven layout synthesis is about 2.4 times (X) longer than the run-time for non-timing-driven layout. Producing the layout solution with the best performance takes 4.9 times longer, on average, than non-timing-driven layout.

Table 1 gives detailed performance results for a subset of the circuits used in the experiments. Columns 2 through 4 of the table present performance data. The clock frequency for each circuit is shown in megahertz; the percentage improvement versus the non-timing-driven layout solution is shown in parentheses. Observe that the performance gap between the automatic and best performance solution is fairly design dependent. For example, for the circuit industry2, the automatic solution performance is within 5% of the best performance result. Conversely, for the circuit industry5, the automatic solution performance is 31% better than non-timing-driven performance, and the best performance solution is superior by 68% to the non-timing-driven solution. The variability in the performance gap between the automatic and the best performance solutions across the design suite is explained by considering the approach taken to automatic constraint generation. Performance constraints are generated based on making a specific fraction of a design's connections critical rather than on the basis of a design's maximum potential performance.

Table 1. Performance results for individual circuits.

Circuit	Non-timing-driven (MHz)	Automatic (MHz) (%)	Best perf. (MHz) (%)
industry1	79.9	110.7 (38.5)	127.7 (59.8)
industry2	111.9	149.8 (33.9)	156.9 (40.2)
industry3	167.1	174.2 (4.2)	190.4 (13.9)
industry4	92.7	111.3 (20.1)	116.3 (25.5)
industry5	109.2	142.7 (30.7)	183.9 (68.4)
industry6	88.8	150.3 (69.3)	162.8 (83.3)
industry7	84.3	110.1 (30.6)	146.4 (73.7)
industry8	133.4	179.3 (34.4)	201.1 (50.7)
industry9	158.5	182.2 (15.0)	198.8 (25.4)
industry10	56.7	89.9 (58.6)	97.9 (72.7)
industry11	73.9	100.4 (35.9)	113 (52.9)
industry12	26.4	32.3 (22.3)	35.5 (34.5)
industry13	60.1	70.9 (18.0)	83.1 (38.3)
industry14	123.2	153.8 (24.8)	179 (45.3)
industry15	112.5	133.3 (18.5)	156.8 (39.4)
	Avg. % impr. (these circuits)	30.3%	48.3%
	Avg. % impr. (all circuits)	28.0%	47.50%

Figs. 3(a) and (b) show the run-time results for the individual circuits and demonstrate a key benefit of our approach: execution time predictability. Fig. 3(a) gives results for the best performance layout runs; Fig. 3(b) gives results for the automatic timing-driven runs. Each point in these figures represents the run-time increase (vs. non-timing-driven layout run-time) for one of the 35 benchmark circuits. Fig. 3(a) shows that the time needed to generate the layout solution with the best performance is highly variable and strongly design dependent. The standard deviation in run-time for this case is 2.03X the non-timing-driven run-time. In addition to the variability apparent in Fig. 3(a), the number of layout runs needed to determine the best performance (as described above) was also variable, and generally involved between 3 and 10 iterations of layout execution and constraint tightening. In contrast, Fig. 3(b) clearly illustrates the power of the proposed constraint generator to deliver performance-driven layouts in a *predictable* amount of execution time. For the automatic timing-driven runs, the standard deviation in run-time is 0.4X the non-timing-driven run-time. Predictable run-time strongly influences user perception, making the proposed approach well-suited for use in a push-button FPGA layout tool.



a) best performance execution times

b) automatic timing-driven execution times

c) run-time/performance for alternative tunings

Fig. 3. (a),(b) Run-time results; (c) Results for alternative tunings.

For the results presented so far, the automatic timing-driven solution was tuned based on our desire to provide layout solutions with reasonably good performance within a consistent and relatively low execution time. Here however, we demonstrate that by increasing the aggressiveness (parameter Y) of the constraint generation, we can tune the automatic approach to produce solutions that represent other points in run-time/performance trade-off space. We investigated more aggressive constraint generation and the results are shown in Fig. 3(c) for two alternative tunings, labeled *tuning*1 and *tuning*2. For *tuning*1, parameter Y was increased to 10% for constraint generation in placement. For *tuning*2, parameter Y was increased to 20% for placement and 3% for routing. The bottom-left data point in Fig. 3(c) reflects the results presented above. The data in Fig. 3(c) demonstrates the effectiveness with which parameter Y controls layout performance and run-time. Observe that the performance characteristics for *tuning*2 layout solutions are quite close to those of the best performance layout solutions studied above, but require considerably less run-time. From this, we conclude that the *extra* run-time needed to move from a "close to" best performance layout to a "true" best performance layout is substantial.

5 Conclusions

Performance-driven layout synthesis is a mandatory component of modern high-speed FPGA design. A key task in the design process is that of determining appropriate performance objectives to supply to layout tools as constraints. In this paper, we presented a new approach to FPGA layout synthesis, called automatic timing-driven layout. The proposed layout approach produces performance optimized layout solutions, without requiring constraints to be specified. Constraints are generated automatically as layout progresses, in a manner that results in a specific fraction of a circuit's connections becoming delay critical.

This method permits performance-optimized layout solutions to be generated, while providing predictable layout execution time. The approach has been integrated into a commercial FPGA layout tool, where it has been tuned to produce solutions having a 28% performance advantage (on average) over non-timing-driven layout solutions, in execution times that are predictably 2.4 times that required for generating a non-timing-driven layout.

Acknowledgements. The authors thank Walt Manaker, Tom Wurtz and Nick Mezei for their helpful comments and their contributions to the implementation.

References

1. Betz, V., Rose, J.: VPR: A new packing, placement and routing tool for FPGA research. In: Proc. of FPL. (1997) 213–222
2. Hitchcock, R.B., Smith, G.L., Cheng, D.D.: Timing analysis of computer hardware. IBM Jour. of Research and Development **26** (1982) 100–105
3. Nair, R., Berman, C.L., Hauge, P.S., Yoffa, E.J.: Generation of performance constraints for layout. IEEE Transactions on CAD **8** (1989) 860–874
4. Youssef, H., Shragowitz, E.: Timing constraints for correct performance. In: Proc. of IEEE/ACM ICCAD. (1990) 24–27
5. Frankle, J.: Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. In: Proc. of ACM/IEEE DAC. (1992) 536–542
6. Marquardt, A., Betz, V., Rose, J.: Timing-driven placement for FPGAs. In: Proc. of ACM/SIGDA FPGA. (2000) 203–213
7. McMurchie, L., Ebeling, C.: Pathfinder: A negotiation-based performance-driven router for FPGAs. In: Proc. of ACM/SIGDA FPGA. (1995) 111–117
8. Nag, S., Rutenbar, R.: Performance-driven simultaneous place and router for row-based FPGAs. In: Proc. of IEEE/ACM ICCAD. (1995) 332–338
9. Elmore, W.: The transient response of damped linear networks with particular regard to wideband amplifiers. Jour. of Applied Physics **19** (1948) 55–62
10. Rubinstein, J., Penfield, P., Horowitz, M.: Signal delay in RC tree networks. IEEE Transactions on CAD **2** (1983) 202–211
11. Xilinx: Virtex-II FPGA Data Book. (2004)

Compact Buffered Routing Architecture

A. Lodi[1], R. Giansante[1], C. Chiesa[1],
L. Ciccarelli[1], F. Campi[1], and M. Toma[2]

[1] ARCES University of Bologna,
Viale Pepoli 3/2, 40123 Bologna
{andrea.lodi, rgiansante, lciccarelli}@deis.unibo.it
[2] STMicroelectronics, NVM-DP, CR&D, Agrate Brianza, Italy
Mario.Toma@st.com

Abstract. In this paper we propose a new routing architecture, based on a new switch called **T**-switch, which we implement in two different versions. Our approach is based on a modified disjoint topology in order to reduce the number of buffers required and on the introduction of a decoding stage between configuration memories and the switch to reduce the number of SRAM cells. This solution is particularly suitable for multi-context arrays, where configuration memory cells need to be replicated as many times as the number of contexts.
The buffered switch proposed has been implemented in two different gate array architectures, in order to evaluate its effectiveness. The results show that the **T**-switch routing architecture reduces the device area occupation up to 29% in a 4-context array. We also show that the critical path delay is reduced, while routability is substantially unaffected.

1 Introduction

The area of an island-style FPGA is dominated by programmable interconnections where switch blocks are the most complex and challenging component. Hence special care must be taken in the design of switch blocks, since they determine routability and most of the delays. In this paper we present a new interconnect architecture and two buffered switch designs that greatly reduce area occupation of previous schemes. The circuits proposed use only one buffer and introduce a decoding stage in the switch block, in order to reduce the number of configuration memories. The architecture presented is also suitable for multi-context FPGAs [1,2]. In these run-time reconfigurable (RTR) devices, different contexts coexist, but only one is active. Switching from one context to another changes FPGA functionality in very short time. However, this advantage is achieved at the cost of replicating each SRAM cell used to store configuration bits as many times as the number of contexts. Therefore area can increase considerably, together with the length of wires which affect delays and power consumption more and more as technology scales. The limited number of memories required by the routing switch proposed can significantly reduce the area penalty of this class of FPGAs. Evaluation of the proposed architecture has

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 179–188, 2004.

been conducted both on a datapath-oriented multi-context array implemented on silicon and on a generic FPGA, in order to verify the generality of the approach.

2 Buffered Switch Design

Technology scaling is dramatically augmenting the portion of delay due to interconnections in FPGA devices. For this reason the trend is to mix buffers and pass-transistors in routing architectures [3]. Unfortunately each buffered switch needs two configuration bits to determine both its on/off state and the signal direction. Although a pass-transistor switch requires only a bit to set its state, it is more inefficient in terms of delay especially for the implementation of high-fanout nets.

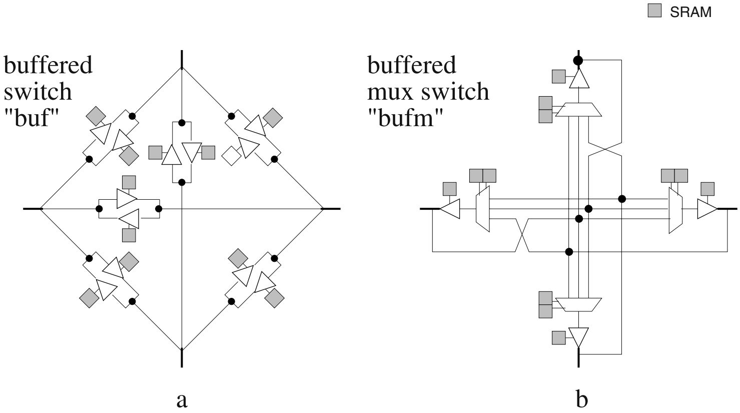
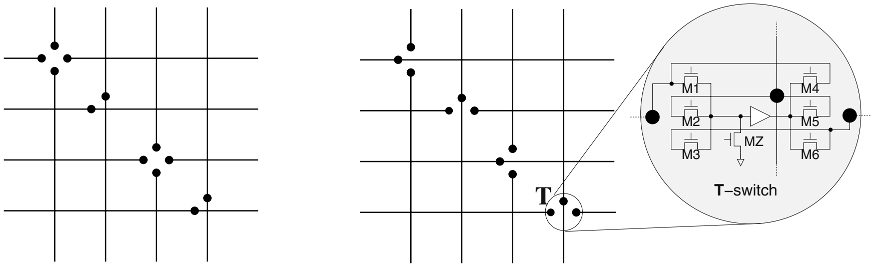


Fig. 1. Typical buffered switch designs

In [4] several routing switches are presented and evaluated from the area and delay point of view. The typical buffered switch (*buf*) is shown in figure 1-a, while in 1-b (*bufm*) a multiplexing stage allows to share buffers among the four wire endpoints converging into the switch. In this way the area occupation due to large sized buffers can be greatly reduced.

2.1 Decoder Based T-switch

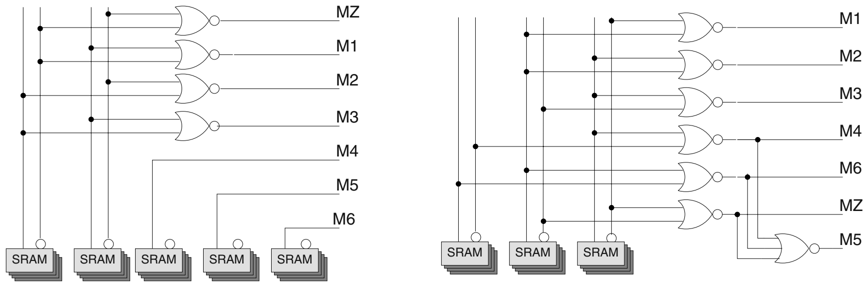
In order to further reduce the number of buffers a new **T**-switch routing architecture is presented. The basic architecture is a modified disjoint topology where horizontal and vertical tracks are connected between the end point of a wire and the mid point of the other one (Fig. 2-b). Differently from the typical switch (Fig. 2-a), it does not allow the connection between two end-points of orthogonal tracks. Since one ending wire entering the switch can be propagated in two



a) connection between ending tracks

b) connection between a midpoint and two endpoints

Fig. 2. **T**-switch topology and schematic.



a) $\mathbf{T}^{\pm}$-switch decoder

b) $\mathbf{T}^{S}$-switch decoder

Fig. 3. Decoding logic structure in **T**-switches.

opposite directions, **T**-nets can be easily implemented. With this architecture, the switch block flexibility F_s is 2, thus the number of possible connection patterns is reduced. Therefore a switch schematic featuring only one buffer (Figure 2) can be adopted, still preserving all the flexibility provided by the **T**-switch architecture.

In multi-context FPGAs the number of configuration bits is extremely critical, since the SRAM cells grow with the number of contexts. Therefore the approach adopted is to introduce a decoding stage between memories and switches which limits configuration redundancy [5]. Since this is an additional block we introduce in our switch, it has been carefully designed in order to keep its area overhead small. Two decoding schemes are presented which have different area occupation and provide different routing flexibility. Since the delay of the decoder stage is not critical all the transistors are minimum sized. The first scheme shown in Figure 3-a ($\mathbf{T}^{+}$-switch) encodes only switch inputs, while each output is individually controlled by a SRAM cell. This decoding scheme preserves all the flexibility provided by the **T**-switch, featuring multi-fanout nets.

Table 1. Area occupation of different switches.

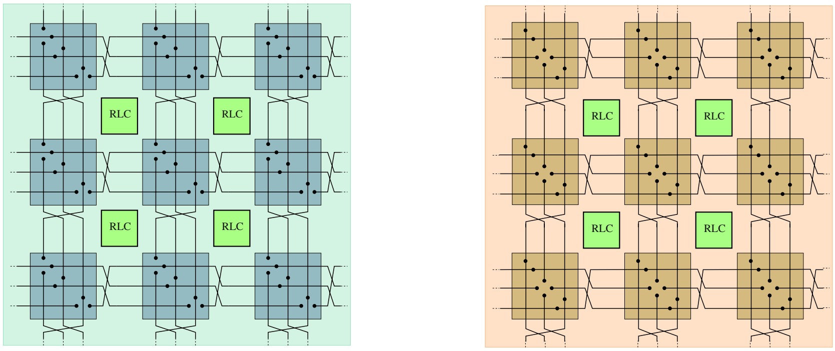
		Number of contexts (k)			
Switch Type	Area Profile	1	2	4	8
pass	6S + 6P	78	138	210	354
buf	12S + 12B + 12P	504	624	768	1056
bufm	12S + 4B + 4P + 20p	228	348	492	780
$\mathbf{T}^+$-switch	5S + 1B + 3P + 4p + D$^+$	98	148	208	328
$\mathbf{T}^s$-switch	3S + 1B + 3P + 4p + D	102	132	168	240
Key:	S(SRAM) = 5T for 1 context S(SRAM) = 6kT + 3 otherwise B(buffer)=29T, P(out pass) = 8T p(in pass)=1T D$^+$(decoder) = 16T, D(decoder) = 30T				

In order to further reduce the number of SRAM cells, the decoding scheme in Figure 3-b is presented ($\mathbf{T}^s$-switch), where only 3 configuration bits are required. In this case switch outputs are also encoded, therefore only one active connection can be implemented. However since the switch provides **T** connections, to some extent multi-fanout nets can still be implemented.

Table 1 shows the area required by different types of switches for different number of contexts. Transistor area has been calculated similarly to what done in [4], though assuming 2-stage buffers having large size. In this way the results achieved in improving area occupation are more conservative. For single-context FPGAs both the $\mathbf{T}^s$-switch and the $\mathbf{T}^+$-switch are the smallest buffered switch, less than half the area of the *bufm* switch. Area reduction is even more remarkable when the number of contexts grows. For a 4-context array, the $\mathbf{T}^s$-switch occupies about 1/3 and 1/4 the area of respectively the *bufm* and the *buf* switch.

Concerning delays, the **T**-switch and the *bufm* have almost the same design, except from the presence of three output pass-transistors instead of one that is typical of tristate buffers. In the case of the $\mathbf{T}^s$-switch only one of the three pass can be active, eliminating any fanout degradation. Since the capacitive contribution of two off pass-transistors is negligible with respect to wire parasitic capacitance, $\mathbf{T}^s$-switch and *bufm* present the same delay performance. With respect to the *buf* switch, which does not have the input multiplexing stage, the delay increase is only 7% as shown in [4]. However it must be noted that these comparisons do not take into account the variation in wire parasitics due to different area occupation of the switches.

On the contrary the $\mathbf{T}^+$-switch allows the buffer to drive two different nets, which can affect delays.



a) T−switch pattern

b) buf(m) pattern

Fig. 4. Switch block pattern for length 3 wires.

3 Architectures Considered

In order to evaluate the performance of the proposed routing architecture, a complete FPGA architecture need to be designed. Since we showed that the switch area considerably changes when adopting a **T**-switch approach, we need to consider its impact on the tile area and consequently on wire parasitics in order to correctly estimate delays.

Two different FPGA architectures have been considered to test the generality of the **T**-switch proposed. The first one is the PiCoGA [8], a datapath-oriented multi-context array which has been implemented on silicon. Based on the information derived from the design of the PiCoGA a second more generic FPGA architecture called GA has been analyzed, which eliminates the datapath-oriented feature of PiCoGA and some peculiarities of its routing architecture.

3.1 Routing Architecture

The baseline routing architecture uses length 3 wires, which defines regular patterns for groups of 3 tracks in order to have identical tiles to be repeated. In the case of **T**-switches the routing scheme involving 3 tracks is shown in Figure 4-a. With this scheme one connection between the midpoints of two wires arises, which is implemented using a pass-transistor. Therefore a net can turn in every switch block through either a **T**-switch or a pass-transistor. The use of pass-switches reduces area occupation, but unfortunately pass-series cause most of routing delays. With the pattern proposed after a turn where a pass-transistor is used, a buffered switch is necessarily encountered. This makes the calculation of propagation delays more accurate even if non-buffered switches are used. In the case of *buf* and *bufm* switches, a typical length 3 wire pattern (Figure 4-b) is adopted, implementing the two midpoint connections with pass-transistors.

Concerning area calculation of the whole switch block, it can be noticed that in the first case the scheme requires one pass-transistor every two **T**-switches,

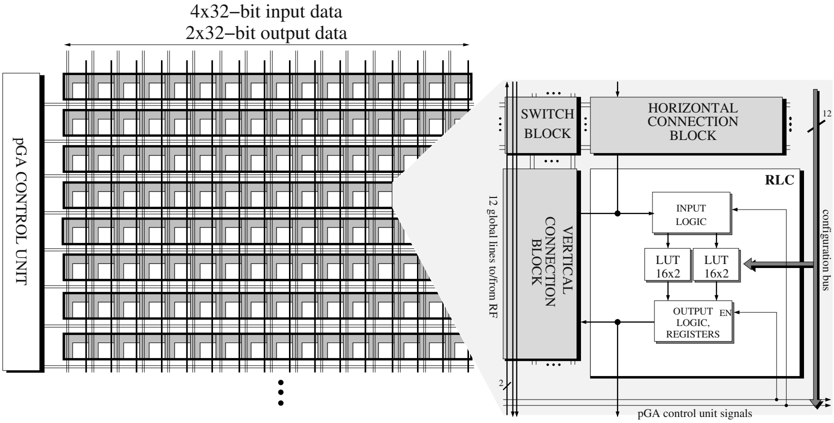


Fig. 5. PiCoGA architecture.

while in the second one there are two pass-transistors for each *buf(m)* switch. Therefore the **T**-switch approach presents twice the number of buffered switch points than *buf(m)*.

3.2 Connect Blocks

The design of connect blocks is important as much as that of switch blocks since they contribute more or less equally to determine most of the area occupation of FPGAs. In the case of multi-context FPGAs a solution based on Decoder-Based Multi-context (DBM) connect blocks can be exploited [6]. The DBM approach is based on the introduction of a decoding stage between configuration memories and pass-transistors connecting the routing channel with the logic block. In the case of a line which has to be connected to one of n other wires, $m = k\lceil log_2 n \rceil$ memory cells are needed, where k is the number of contexts.

The DBM structure achieves a high reduction in the transistor count, if compared with the standard solution where a dedicated SRAM cell individually drives each pass-transistor. As shown in [6] area reduction is remarkable, especially in case of a high number of contexts or of a wide bus: 60 % and 70 % respectively for 4 and 8 contexts when 32 connections are implemented. Note that the application of this approach to an output of a logic block, obviously reduces routability, as it does not allow connections with fanout > 1 in the connect block. Concerning delays, the DBM structure has the same delay of the standard solution, since it avoids the pass-transistor series delay of output multiplexing approaches.

4 PiCoGA Implementation

The PiCoGA (Pipelined Configurable Gate Array) is a configurable datapath designed to implement high throughput pipelines. From a structural point of view the PiCoGA is an array of rows, each representing a possible stage of a customized pipeline processing up to 32-bit wide operands. Each row is composed of 16 Reconfigurable Logic Cells (RLC) which are directly connected to PiCoGA pins with four 32-bit input and two 32-bit output busses. Switch blocks connect vertical and horizontal wires. Each logic cell is composed of a cluster of 2 4-input LUTs having 2-bit granularity. The PiCoGA is designed as a multi-context device featuring 4 configuration cache layers. Context switch occurs in one clock cycle, providing 4 immediately available circuits.

Since the PiCoGA is a datapath-oriented array, the routing network has been designed with 2-bit granularity to reduce area occupation. However, input connection blocks have 1-bit granularity in order to preserve routability and efficiency of resources even in the case of odd shifting, single bit control signals and coupling of bits coming from different RLCs. Channels are composed of 15 couples of length 3 tracks, which has been found to be a good compromise between propagation delay and routability. Direct connection from an RLC to the one below has also been provided. Furthermore, two couples of global horizontal lines have been designed to support fast propagation of control signals such as multiplexer selection bits which often need to be routed to all RLCs in a row.

PiCoGA configuration flow is based on the Griffy compiler [9] which translates a generic DFG, described using a subset of ANSI C, into a netlist of RLCs. Configuration of logic blocks is obtained by using a library based synthesis. Griffy-C also places the netlist of RLCs on the PiCoGA rows, implementing a pipelined execution of the DFG.

In order to correctly configure the DBM routing architecture, we designed a specific router for PiCoGA, called XiRouter, which is based on the routing algorithm from VPR [7]. One main modification concerns the implementation of $\mathbf{T}^s$-switches, which are critical when implementing multi-fanout nets. Since a decoding stage is introduced in $\mathbf{T}^s$-switches, a wire can only drive one of the other two converging ones. This is implemented in XiRouter by increasing the cost of the connections in a $\mathbf{T}^s$-switch if a net already passes through the switch.

4.1 Experimental Results

The design of the PiCoGA has been carried out exploiting STMicroelectronics 0.13 μm CMOS technology. This version of the PiCoGA adopts both $\mathbf{T}^s$-switches and DBM connect blocks. A standard cell based design flow starting from a *vhdl* description was used for random logic inside the RLC, while LUTs, memory cells and the routing network were custom designed. With this approach almost half of the area is occupied by standard cells, SRAM cells to configure the RLC and overhead due to standardization of hard macro dimensions during back-end flow. Since this portion of the tile could be drastically reduced, the tile area decrease achieved when adopting a $\mathbf{T}^s$-switch approach is certainly a conservative number with respect to a full custom design.

The *buf*, *bufm* and $\mathbf{T}^s$ switches are compared, in terms of area and delays. The tile area of the different approaches are calculated starting from that of the 0.13 μm implemented version of PiCoGA and augmenting it. Since DBM

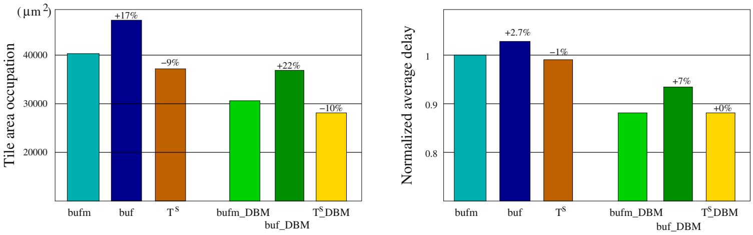


Fig. 6. PiCoGA area and delay performance.

connections reduce routability when applied to RLC outputs, their impact need to be carefully evaluated especially when used in conjunction with $\mathbf{T}^s$-switches which also have limited fanout capabilities. Therefore we considered different models with DBM or traditional connect blocks.

The tile area and delays of the different routing architectures are illustrated in Fig. 6 where the terminology adopted is described below:

- buf(m): which is the baseline architecture with buf(m) switches;
- T^s: which adopts $\mathbf{T}^s$-switches;
- buf(m)_DBM: which adopts buf(m) switches and DBM connect blocks;
- T^s_DBM: which adopts both $\mathbf{T}^s$-switches and DBM connect blocks;

In the histograms *bufm* and *bufm_DBM* are used for reference when calculating the percent variations. Results show that $\mathbf{T}^s$-switch blocks reduce tile area of 10% and 27% with respect to the *bufm* and *buf* switches in the case DBM connect blocks are adopted. In order to evaluate the delays of the described architectures, we implemented a set of DSP algorithms. The delay associated to the critical path is lower in the models adopting $\mathbf{T}^s$-switches and DBM connect blocks. This is essentially due to the smaller device area, such that signals have to drive shorter wires which present lower parasitic effects.

The four models have shown very little difference in channel occupation, thus demonstrating that routability with DSP-oriented algorithms is still granted even in the $\mathbf{T}^s$_DBM architecture. This is also due to the fact that each rlc can connect separately to the routing channel on the left and bottom side, to the horizontal global lines below and directly to the rlc below. In this way multi-fanout nets can be supported by the different connect blocks, though with limited flexibility.

5 Generic FPGA Implementation

In order to evaluate the efficiency of the **T**-switch routing architecture on a generic device different from a datapath-oriented architecture such as PiCoGA, we introduce a more generic FPGA, called GA. CLBs are composed of four 4-inputs LUTs, having eight inputs and four outputs. A local crossbar connecting the eight inputs of the CLB to the sixteen of the LUTs is fully populated and any feedback connection among outputs and inputs of the LUTs are allowed.

The GA routing architecture is characterized by:

Table 2. Different GA routing architecture performance.

	Routing architecture	Tile area ($K\mu^2$)	Delay variation	Channel width variation
1-context GA	bufm	15		
	buf	20.5 (+36%)	+3%	0 (0%)
	$\mathbf{T}^+$	14.1 (-6%)	-3%	0 (0%)
	$\mathbf{T}^s$	14.2 (-5%)	-5%	+0.5 (+4%)
4-context GA	bufm	38		
	buf	43.6 (+15%)	+2%	0 (0%)
	$\mathbf{T}^+$	35.8 (-6%)	-4%	0 (0%)
	$\mathbf{T}^s$	34.2 (-10%)	0%	+0.5 (+3%)
4-context GA_DBM	bufm_DBM	27.2		
	buf_DBM	32.8 (+21%)	+2%	0 (0%)
	$\mathbf{T}^+$_DBM	25 (-8%)	-2%	-0.1 (0%)
	$\mathbf{T}^s$_DBM	23.4 (-14%)	0%	+2.3 (+11%)

- Two orthogonal identical thirty tracks wide channels of length 3 wires.
- Internal switch block population of 100%.
- Fully populated input and output connect blocks.

Sis [10] was used to technology map each circuit to 4-LUTs and flip-flops and T-VPack to cluster them into CLBs. Differently from PiCoGA, placement of logic blocks on the GA is provided by the placement algorithm of VPR. Programmable interconnections are implemented by a modified version of XiRouter which also take into account degradation under fanout of the $\mathbf{T}^+$-switch.

5.1 Experimental Results

We tested the proposed interconnect architecture by comparing the routing performance of six different versions of GA on a set of 35 MCNC circuits. As in the case of PiCoGA, models are characterized by the adoption of different switch blocks, with DBM or traditional connect blocks. Table 2 shows that area performance of **T**-switches is better than in the case of PiCoGA, since we consider to have custom design also for CLB logic, thus reducing its impact on the total area. In this case $\mathbf{T}^s$-switches reduce tile area up to 14% (29% with respect to *buf*), even if routability is affected when applied together with DBM connect blocks. Concerning delays, the average critical path delay is substantially unchanged.

$\mathbf{T}^+$-switches still achieve a considerable reduction of the tile area (6-8%), while having the same routability of traditional *buf(m)* switches. Furthermore critical path delay is reduced up to 4%, mainly due to shorter wires of the $\mathbf{T}^+$ architecture.

Experiments on a standard single-context GA version have also been conducted. In this case DBM connect blocks are no more convenient in terms of area occupation, so they have not been considered. Results substantially don't change with respect to the multi-context array, showing that the **T**-switch routing architecture still achieves the best performance.

6 Conclusions

A new routing architecture and a new buffered switch, called **T**-switch, has been proposed. Schematic details of the switch blocks have been presented. Performance analysis compared **T**-switch with typical routing switches in two kinds of gate arrays, having different number of contexts. Results have shown that the proposed solution achieves a considerable reduction in the device area, and the best performance in critical path delay without affecting routability.

References

1. A. DeHon, DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.31–39, Napa Valley, California, April 1994.
2. S. Trimberger, D. Carberry, A. Jhonson and J. Wong. A Time Multiplexed FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.34–40, Napa Valley, California, April 1997.
3. M. Sheng and J. Rose. Mixing buffers and pass transistors in FPGA routing architectures. In *ACM/SIGDA International Symposium on FPGAs*, pp. 75–84, February 2001.
4. G. Lemieux and D. Lewis Circuit Design of Routing Switches. In *ACM/SIGDA International Symposium on FPGAs*, February 2002.
5. A. DeHon, Entropy, Counting, and Programmable Interconnect. In *ACM/SIGDA International Symposium on FPGAs*, February 1996.
6. A. Lodi, L. Ciccarelli, A. Cappelli, F. Campi, M, Toma. Decoder-based interconnect structure for multi-context FPGAs. In *Electronic Letters*, pp. 362–364, 38, 2003.
7. V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In *International Workshop on Field Programmable Logic and Applications*, Sept. 1997.
8. A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo and R. Guerrieri. A Pipelined Configurable Gate Array for Embedded Processors. In *Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 21–30, February 2003.
9. C. Mucci, C. Chiesa, A. Lodi, M. Toma and F. Campi A C-based Algorithm Development Flow for a Reconfigurable Processor Architecture. In *International Symposium on System-on-Chip*, Tampere, Finland, 2003.
10. E.M. Sentovich, K.J. Singh; L. Lavagno, C. Moon, R. Murgai A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. *UCB/ERL M92/41*, 1992.

On Optimal Irregular Switch Box Designs

Hongbing Fan[1], Yu-Liang Wu[2*], Chak-Chung Cheung[3], and Jiping Liu[1]

[1] The University of Lethbridge, Lethbridge, AB. Canada T1K 3M4
{fan, liu}@cs.uleth.ca
[2] The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
ylw@cse.cuhk.edu.hk
[3] Department of Computing, Imperial College London, United Kingdom
rcheung@doc.ic.ac.uk

Abstract. In this paper, we develop a unified theory in analyzing optimal switch box design problems, particularly for the unsolved irregular cases, where different pin counts are allowed on different sides. The results drawn from our system of linear Diophantine equations based formulation turn out to be general. We prove that the divide-and-conquer (reduction) design methodology can also be applied to the irregular cases. Namely, an optimal arbitrarily large irregular or regular switch box can be obtained by combining small prime switch boxes, which largely reduces the design complexity. We revise the known VPR router for our experiments and show that the design optimality of switch boxes does pay off.

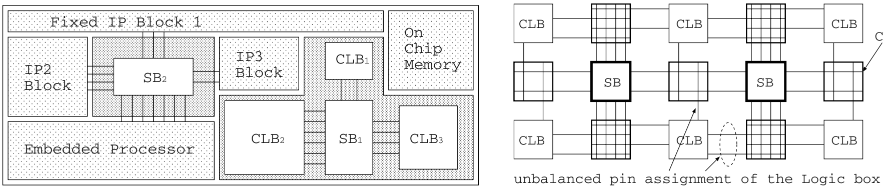
Keywords. Configurable computing, on-chip network, FPGA, switch box

1 Introduction

A switch box (SB) consists of terminals (pins) and programmable switches, with each switch connecting two pins on different sides. A switch box is *regular* if all sides have the same number of pins; otherwise it is *irregular*. As the optimality of a switch box design imposes a crucial impact on silicon cost and performance of FPGAs, extensive investigations on this problem have been carried out in recent years, see [3,4,5,7,12,13] for examples. Chang et al.[5] started the study on the so-called optimal Universal Switch Block (USB) structure, which is defined as a switch box being able to accommodate any 2-pin net routing requirement with the least number of switches. In [9,7,8], the so-called Hyper-Universal Switch Box (HUSB) was investigated to cover the general cases of multi-pin routings. Although it has been shown that this optimal switch box design problem can be solved by divide-and-conquer (reduction) approaches [7,8], only regular switch box cases were analyzed before.

[*] Research partially supported by a Hong Kong Government RGC Earmarked Grant, Ref. No. CUHK4236/01E, and Direct Grant CUHK2050244

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 189–199, 2004.



(a) SoC with customized Logic IP Cores (b) Unbalanced pin assignment

Fig. 1. Examples of irregular switch boxes.

Despite the surprising result suggesting that square switch boxes might be the best in terms of area and delay [10], Fig. 1 gives some scenarios where irregular switch boxes are efficient and desirable. Besides the classic crossbar structures with unequal input and output pins [11], some recent technological advances have somehow stirred the study interest on developing more general irregular switch boxes, which, for example, can allow more customization flexibility for embedded FPGA cores of SoC designs. In [1], directional bias and non-uniform FPGA architectures were experimentally addressed. The directional bias refers to the different number of tracks between horizontal and vertical channels, while the non-uniformness refers to channel width variation between different channels of the same direction. In [10], rectangular switch blocks formed by a union of several aligned regular switch boxes [14] were studied. Irregular switch boxes can also be used in hierarchical FPGA architectures and circuit-switching based reconfigurable on-chip networks with non-uniform I/O port densities of different sides. In all these examples, irregular switch boxes provide extra flexibility in designing on-chip networks with non-uniform channel densities.

Similar to the regular switch boxes design problems, the problem is to design optimal irregular switch boxes satisfying two specifications: 1) shape specification, which includes the number of sides (dimension) and the number of terminals on each side (channel density), and 2) routability specification, which is characterized by the set of routable routing cases.

We use $(r_1, \ldots, r_k)$-SB to denote a k-sided switch box with channel density r_i on side i for $i = 1, \ldots, k$. We are interested in designing a generic class of switch boxes determined by a *channel density ratio vector* $\mathbf{d}$ and a *residual vector* $\mathbf{c}$, i.e., a group of $(w\mathbf{d} + \mathbf{c})$-SBs with all integer scales $w \geq 1$. In particular, when $\mathbf{d} = (1, \ldots, 1)$ and $\mathbf{c} = (0, \ldots, 0)$, a $(w\mathbf{d} + \mathbf{c})$-SB is a regular switch box of k sides with w terminals on each side. We will show that a solution scheme for the generic switch box design problem can be used to design a specific irregular switch box.

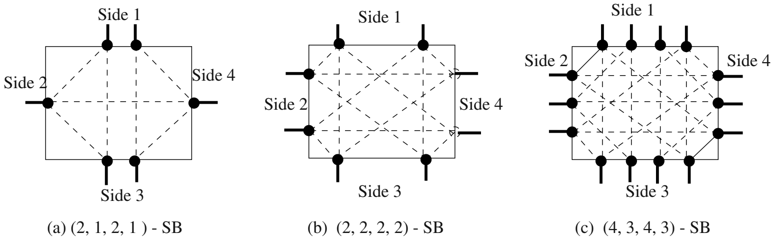
In this paper, we first formulate routing requirements as nonnegative integer solutions of System of Linear Diophantine Equations (SLDEs), then apply the theory of SLDE to find decompositions of routing requirements. Accordingly a reduction design scheme for irregular switch box design is obtained, which generalizes the design scheme for regular switch boxes. In other words, an arbitrarily large irregular switch box can also be obtained by combining some small prime switch boxes. The VPR [2] router is used to compare the routability of different irregular switch-boxes on a fixed channel density ratio. The large MCNC benchmark circuits are used in the experimental test.

This paper is organized as follows. Terminology and switch box design problem are given in Section 2. Section 3 formulates a new modelling of routing requirements and develops decomposition theory of routing requirements by applying the theory of system of linear Diophantine equations. In Section 4, the generalized reduction design scheme for irregular switch boxes of arbitrary shapes is introduced. Two design examples for illustration and experimental results are presented in Sections 5 and 6, respectively. Conclusions are drawn in Section 7.

2 The Switch Box Design Problem

We model a switch box as a graph as in [7]. For an $(r_1, \ldots, r_k)$-SB, we denote the j-th terminal on side i by $v_{i,j}$ for $i = 1, \ldots, k, j = 1, ..., r_i$. If there is a switch joining terminals $v_{i,j}$ and $v_{i',j'}$, then we denote the switch by an edge $v_{i,j}v_{i',j'}$. Thus, an $(r_1, \ldots, r_k)$-SB corresponds to a k-partite simple graph with vertex partition $(V_1, \ldots , V_k)$, where $V_i = \{v_{i,j} | j = 1, \ldots , r_i\}, i = 1, \ldots , k$.

The *disjoint union* of two k-sided switch boxes G_1 and G_2 is a k-sided switch box with the i-th side being the union of the i-th sides of both G_1 and G_2 together with all switches of G_1 and G_2, denoted by $G_1 + G_2$. The disjoint union of h copies of G_1 is denoted by hG_1. As depicted in Fig.2, the $(4, 3, 4, 3)$-SB (c) is a disjoint union of $(2, 1, 2, 1)$-SB (a) and $(2, 2, 2, 2)$-SB (b).



(a) (2, 1, 2, 1) - SB (b) (2, 2, 2, 2) - SB (c) (4, 3, 4, 3) - SB

Fig. 2. An example of the disjoint union of two switch boxes.

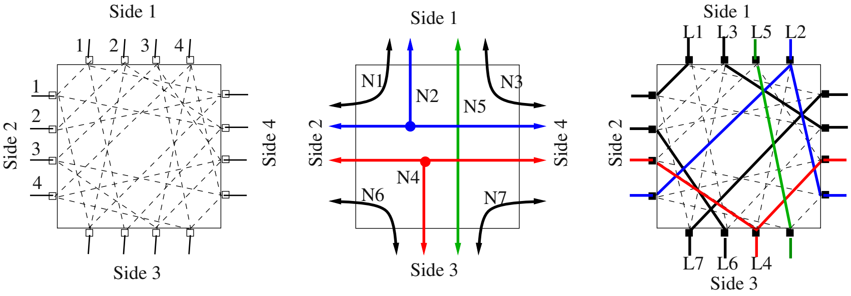
A (signal) net for a k-sided switch box is a connection request on some terminals of the switch box. In our switch box design problems, a net only specifies the sides where its terminals are located; a router will take care of exact terminal assignments besides switch connection assignments [5,7,8,13]. A net is said to be an m-pin net if it specifies m different sides; an m-pin net which specifies sides $i_1, \ldots, i_m$ will be expressed as $\{i_1, \ldots, i_m\}$, which is a subset of $\{1, \ldots , k\}$. For example, a 3-pin net connecting three terminals in sides 1, 2 and 3 is represented by $\{1, 2, 3\}$. Sometimes only certain types of nets are considered in the switch box design; this set of types consists of some subsets of $\{1, \ldots , k\}$, it is called a *net pattern set (over* $\{1, \ldots , k\}$*)*, denoted by $\mathcal{P}$. A net N in $\mathcal{P}$ is referred as a *$\mathcal{P}$-net*. A net of size 1 (singleton) does not need a switch in routing, but it is very convenient when consider its mathematical properties. Therefore, we always assume that any $\mathcal{P}$-net contains all singletons.

For examples, the net pattern set $\mathcal{P} = \mathcal{P}_2 = \{N|N \subset \{1, \dots, k\}, |N| \leq 2\}$ is used in the study of universal switch boxes[5], while $\mathcal{P} = \mathcal{P}_k = \{N|N \subset \{1, \dots, k\}\}$ is used for hyperuniversal switch box designs[7].

A *routing requirement* (RR) for a switch box is a group of nets need to be connected simultaneously through the switch box. Formally, a $\mathcal{P}$-net $(r_1, \dots, r_k)$-RR is a collection of $\mathcal{P}$-nets $[N_1, \dots, N_r]$ such that $N_j \in \mathcal{P}$ for $j = 1, \dots, r$, and the number of N_j's that specify side i is equal to r_i, i.e., $|\{j|i \in N_j\}| = r_i$ for $i = 1, \dots, k$.

A *feasible routing* of a routing requirement in a switch box is an ON/OFF assignment of the switches such that all the nets of the routing requirement are connected (realized) simultaneously. A realization of a net is modelled as a tree with one vertex in each side specified by the net. Formally, it is defined as follows. Let G be a $(r_1, \dots, r_k)$-SB with sides $V_i = \{v_{i,j}|j = 1, \dots, r_i\}, i = 1, \dots, k$. An $(r_1, \dots, r_k)$-RR $R = [N_1, \dots, N_m]$ is said to be *routable* in G if G contains m vertex disjoint subtrees $L_1, \dots, L_m$ such that for each $i = 1, \dots, m$, L_i has exactly one vertex in the sides specified by N_i, i.e., $|V(L_i) \cap V_j| = 1$ for each $j \in N_i$. We call $\{L_1, \dots, L_m\}$ a *feasible routing* of R in G, and L_i a feasible routing of N_i in G. We note that if N_i is a singleton, then its feasible routing only consists of a terminal with no switch used. Therefore adding (or removing) singletons to a routing requirement does not change its routability.

Fig.3(a) shows a $(4, 4, 4, 4)$-SB, where each side has four terminals which are assigned unique track IDs (1 to 4). Fig.3(b) shows a $(4, 4, 4, 4)$-RR, which has seven nets: $N_1 = \{1, 2\}, N_2 = \{1, 2, 4\}, N_3 = \{1, 4\}, N_4 = \{2, 3, 4\}, N_5 = \{1, 3\}, N_6 = \{2, 3\}, N_7 = \{3, 4\}$. Net N_2 is a 3-pin net, which requires two switches to connect its three terminals in sides 1, 2, and 4. Fig.3(c) shows a feasible routing for the routing requirement.



(a) a (4, 4, 4, 4) – SB (b) a (4, 4, 4, 4) – RR (c) a feasible routing

Fig. 3. An example of switch box, routing requirement and feasible routing.

An $(r_1, \dots, r_k)$-SB G is said to be $\mathcal{P}$-*universal* if every $\mathcal{P}$-net $(r_1, \dots, r_k)$-RR is routable in G, and an *optimal* $\mathcal{P}$-universal switch box is one with the least number of switches. The notion of $\mathcal{P}$-universal unifies both universal and hyperuniversal discussed in [5,7]. The $\mathcal{P}_2$-universal is just the so called universal, while the $\mathcal{P}_k$-universal is the hyperuniversal.

Generic switch box design problem: *Given k-dimensional nonnegative integer vectors* $\mathbf{d}$ *and* $\mathbf{c}$ *and a net pattern set* $\mathcal{P}$*, design an optimal* $\mathcal{P}$*-universal* $(w\mathbf{d}+\mathbf{c})$*-SB for every* $w \geq 1$.

Our ultimate goal is to derive a general method to solve the generic switch box design problem. A solution scheme for a generic switch box design problem can be used to design a specific $(r_1,\ldots,r_k)$-SB. For a given vector $(r_1,\ldots,r_k)$, we can select proper $\mathbf{d},\mathbf{c}$ and w_0 such that $(w_0\mathbf{d}+\mathbf{c}) = (r_1,\ldots,r_k)$, then a $(w_0\mathbf{d}+\mathbf{c})$-SB is an $(r_1,\ldots,r_k)$-SB.

3 Decomposition Theorems

Our design technique for generic switch boxes is based on the decomposition properties of routing requirements. We prove the general decomposition theorems by employing the routing requirement vectors and the theory of system of linear Diophantine equations.

The routing requirement vectors were first used to represent (w,w,w,w)-RRs in [5]. We modify the definition to fit in our routing requirements modelling as follows. For a 2-pin net (w,w,w,w)-RR R, let $n_{i,j}$ denote the number of net $\{i,j\}$ in R, and let n_i denote the number of singleton $\{i\}$ in R, we call vector $(n_1,n_2,n_3,n_4,n_{1,2},n_{1,3},n_{1,4},n_{2,3},n_{2,4},n_{3,4})$ a 2-pin net *routing requirement vector* of R. Obviously a nonnegative integer vector is a routing requirement vector if and only if it satisfies the following equation.

$$\begin{cases} n_{1,2}+n_{1,3}+n_{1,4}+n_1 = w \\ n_{1,2}+n_{2,3}+n_{2,4}+n_2 = w \\ n_{1,3}+n_{2,3}+n_{3,4}+n_3 = w \\ n_{1,4}+n_{2,4}+n_{3,4}+n_4 = w \end{cases} \tag{1}$$

In general, for a given net pattern set $\mathcal{P} = \{S_1,\ldots,S_t\}$, a $\mathcal{P}$-net $(r_1,\ldots,r_k)$-RR $R = [N_1,\ldots,N_m]$ can be expressed by a vector $X = (x_1,\ldots,x_t)$ where x_i is the number of N_is in R, i.e., $x_i = |\{j|N_j = S_i\}|$, denoted by $\mathcal{P}$-net $(r_1,\ldots,r_k)$-RRV. A vector $X = (x_1,\ldots,x_t)$ is a $\mathcal{P}$-net $(r_1,\ldots,r_k)$-RRV if and only if it is a nonnegative integer solution of

$$AX^T = (r_1,\ldots,r_k)^T, \tag{2}$$

where $A = (a_{i,j})_{k\times t}$ is the incidence (characterization) matrix of $\mathcal{P}$. I.e., $a_{i,j} = 1$ if $i \in S_j$; otherwise $a_{i,j} = 0$. Therefore, we can compute all routing requirements by finding all nonnegative integer solutions of equation (2).

In mathematics, a linear system $AX^T = \mathbf{b}^T$ is called a *system of linear Diophantine equations (SLDE)* if the entries of A and $\mathbf{b}$ are integers, and only nonnegative integer solutions are considered. If $\mathbf{b}^T = \mathbf{0}$, the system is *homogeneous.* The SLDE has been studied extensively. Let $X = (x_1,\ldots,x_t)$ and $X' = (x'_1,\ldots,x'_t)$ be two nonnegative integer solutions of an SLDE. Define $X \preceq X'$ if $x_i \leq x'_i$ for all $i = 1,\ldots,t$. A solution of an SLDE X is said to be a *minimal solution* if there is no other solution X'' satisfying $X'' \preceq X$. It is known that the set of all minimal solutions is finite, and that any nonnegative integer solution of a homogeneous SLDE is a nonnegative integer linear combination of its minimal solutions (called the *Hilbert basis*). We use $\mathcal{B}[S]$ to denote the set of all minimal solutions of an SLDE S. There are several known algorithms for computing the set of minimal solutions of an SLDE. Interested readers can consult Contejean and Devie [6].

Given nonnegative integer vectors $\mathbf{d} = (d_1, \ldots, d_k)$ and $\mathbf{c} = (c_1, \ldots, c_k)$, a $\mathcal{P}$-net ($w\mathbf{d}+\mathbf{c}$)-RRV corresponds to a (X, w), which is a nonnegative integer solution of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T$. There is a vector $(X', w') \in \mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T]$ such that $(X', w') \preceq (X, w)$. $(X, w) - (X', w')$ is a solution of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T$, thus, $(X, w) - (X', w')$ is a nonnegative-integer linear combination of minimal solutions of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T$. Therefore,$(X, w) = (X', w') + \sum_{i=1}^{m} a_i(X_i, w_i)$, where (X_i, w_i)s are minimal solutions of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T$. In summary, we have the following theorem.

Theorem 3.1 (The first decomposition theorem). *Let* $\mathbf{d}$ *and* $\mathbf{c}$ *be two k-dimensional nonnegative integer vectors and $\mathcal{P}$ be a net pattern set. Then any $\mathcal{P}$-net* $(w_0\mathbf{d} + \mathbf{c})$*-RRV can be expressed as a vector in* $\mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T]$ *plus a nonnegative integer linear combination of vectors in* $\mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T]$*, where A is the incidence matrix of $\mathcal{P}$.*

Theorem 3.2 (The second decomposition theorem). *Let* $\mathbf{d}$ *and* $\mathbf{c}$ *be two k-dimensional nonnegative integer vectors and $\mathcal{P}$ be a net pattern set. Then there exists an integer $p > 0$ and a finite set of nonnegative integers D satisfying the following properties: for any $w \geq 1$, there is a $q_w \in D$ such that every* $(w\mathbf{d} + \mathbf{c})$*-RRV can be represented as a sum of one* $(q_w\mathbf{d} + \mathbf{c})$*-RRV and* $\frac{w-q_w}{p}$ $(p\mathbf{d})$*-RRVs. Consequently, if U_0 is a $\mathcal{P}$-universal* $(p\mathbf{d})$*-SB and U_w is a $\mathcal{P}$-universal* $(q_w\mathbf{d}+\mathbf{c})$*-SB, then* $U_w + \frac{w-q_w}{p}U_0$ *is a $\mathcal{P}$-universal* $(w\mathbf{d} + \mathbf{c})$*-SB.*

Proof. Due to page limitations, the proof is not included in this paper and is available upon request. □

4 Generalized Reduction Design Scheme

The decomposition theorems described in the last section establish the following reduction design scheme for generic switch boxes with simple structure and reduced number of switches.

Reduction Design Scheme for Generic Switch Boxes
Given two k-dimensional nonnegative integer vectors $\mathbf{d}$ and $\mathbf{c}$ and a net pattern set $\mathcal{P}$ with an incidence matrix A:

I. Compute $\mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T]$ and $\mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T]$ using Hilbert basis algorithm, where A is the incidence matrix of the net pattern set $\mathcal{P}$. Suppose $\mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T] = \{(X_1, w_1), \ldots, (X_m, w_m)\}$ and $\mathcal{B}[(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T]= \{(X'_1, w'_1), \ldots, (X'_l, w'_l)\}$.

II. Use $S = \{w_1, \ldots, w_m\}, S' = \{w'_1, \ldots, w'_l\}$ to compute an integer p and a set D satisfying the conditions of Theorem 3.2. We have that p is bounded by the least common multiple of $w_1, \ldots, w_m$, but p could be much smaller, and $D \subset \{0, 1, \ldots, mp - m + \max\{w'_1, \ldots, w'_l\}\}$.

III. Design a $\mathcal{P}$-universal $(p\mathbf{d})$-SB U_0 and set up a feasible routing table recording feasible routings for every $p\mathbf{d}$-RRs in U_0. For each $r \in D$, design a $\mathcal{P}$-universal $(r\mathbf{d} + \mathbf{c})$-SB, U_r, and set up the corresponding feasible routing table. We call U_0 and U_r $(r \in D)$ *prime switch boxes.*

IV. For any $w \geq 1$, construct a $\mathcal{P}$-universal $(w\mathbf{d} + \mathbf{c})$-SB as follows: if $w \in D$, then use the prime $(w\mathbf{d} + \mathbf{c})$-SB U_w, otherwise choose the minimum q such that $w - qp \in D$. The disjoint union of one U_{w-qp} and q copies of U_0, i.e., $U_{w-qp} + qU_0$, is a $\mathcal{P}$-universal $(w\mathbf{d} + \mathbf{c})$-SB. We call it a compound switch box.

Remark: *We note that if we only want to construct a $\mathcal{P}$-universal $(w\mathbf{d} + \mathbf{c})$-SB for a specific w, we only need to construct a $\mathcal{P}$-universal $(p\mathbf{d})$-SB U, and a $\mathcal{P}$-universal $(q_w\mathbf{d} + \mathbf{c})$-SB U_{w-qp}. Then $U_{w-qp} + \frac{w-q_w}{p}U_0$ is a $(w\mathbf{d} + \mathbf{c})$-SB.*

The reduction design scheme reduces the generic switch box design problem to its prime switch box design problems. Although there is still no efficient known method for designing optimal prime switch boxes, the degree of difficulty has been largely reduced due to the much smaller sizes of prime switch boxes. Nonetheless, as a complete switch box has a switch joining every pair of terminals from different sides, it is $\mathcal{P}$-universal for any $\mathcal{P}$. Therefore, if we simply let U_0 be the complete $(p\mathbf{d})$-SB $K_{p\mathbf{d}}$ and U_r be the complete $(r\mathbf{d} + \mathbf{c})$-SB $K_{(r\mathbf{d}+\mathbf{c})}$, then $K_{(q_w\mathbf{d}+\mathbf{c})} + \frac{w-q_w}{p}K_{p\mathbf{d}}$ is a $\mathcal{P}$-universal $(w\mathbf{d} + \mathbf{c})$-SB, and it has $O(w)$ number of switches. We also have that the decomposition of a routing requirement can be done in a polynomial time, and finding a feasible routing in a prime switch box can be done in a constant time by looking up a routing table created for the prime switch box. Therefore, there is a polynomial time algorithm for finding a feasible routing in the compound switch box.

Theorem 4.1. *For any given vectors* $\mathbf{d}$, $\mathbf{c}$ *and net pattern* $\mathcal{P}$, *there is a* $\mathcal{P}$-*universal* $(w\mathbf{d} + \mathbf{c})$-*SB with* $O(w)$ *switches for every* $w \geq 1$, *and an algorithm which finds a feasible routing for any* $(w\mathbf{d} + \mathbf{c})$-*RR in the switch box in time polynomial of* w.

5 Two Examples of Irregular Switch Box Designs

In this section, we show how to design a specific optimal $(4, 5, 6)$-HUSB and a $(5, 6, 7)$-HUSB using the reduction design scheme. The strategy consists of choosing $\mathbf{d} = (1, 1, 1)$ and $\mathbf{c} = (0, 1, 2)$ first, then designing the generic $(w, w + 1, w + 2)$-HUSBs. The target switch boxes are the cases when $w = 4, 5$.

I. The net pattern set for 3-sided hyper-universal switch boxes is $\{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. The incidence matrix of the net pattern set is

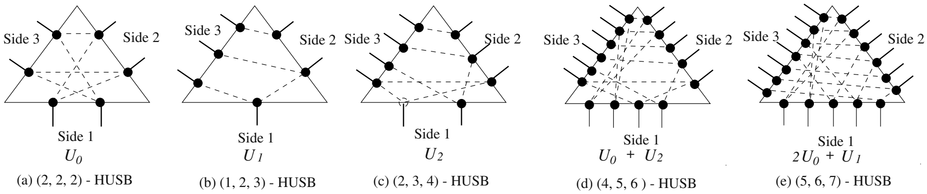
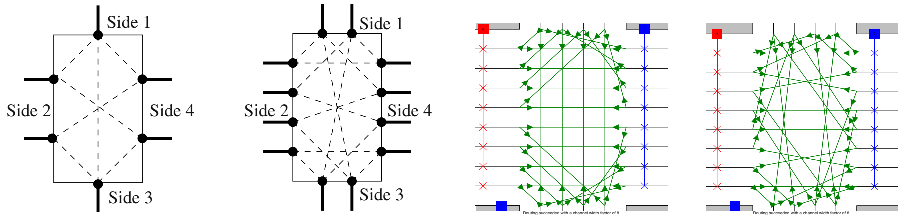


Fig. 4. Optimal $(3, 4, 5)$-HUSB and $(5, 6, 7)$-HUSB.


Side 1
Side 2
Side 4
Side 3
Side 1
Side 2
Side 4
Side 3
Routing succeeded with a channel width factor of 8.
Routing succeeded with a channel width factor of 8.

(a) $(1, 2, 1, 2)$-HUSB (b) $(2, 4, 2, 4)$-HUSB (c) Disjoint-like (d) Optimal design

Fig. 5. Rectangular switch boxes.

$$A = \begin{pmatrix} 1\,0\,0\,1\,1\,0\,1 \\ 0\,1\,0\,1\,0\,1\,1 \\ 0\,0\,1\,0\,1\,1\,1 \end{pmatrix}.$$

By computing the set of minimal solutions of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T$, we obtain

$$(1, 1, 1, 0, 0, 0, 0, 1), (0, 0, 0, 0, 0, 0, 1, 1), (1, 0, 0, 0, 0, 1, 0, 1),$$
$$(0, 1, 0, 0, 1, 0, 0, 1), (0, 0, 0, 1, 1, 1, 0, 2), (0, 0, 1, 1, 0, 0, 0, 1)$$

By computing the set of minimal solutions of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T$ we obtain

$$(0, 1, 2, 0, 0, 0, 0, 0), (0, 0, 1, 0, 0, 1, 0, 0), (0, 0, 0, 0, 1, 2, 0, 1).$$

II. Compute p and D of Theorem 3.2. We have $p = 2$ and $D = \{1, 2\}$. That is, any solution (X, w) of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T$ can be expressed as $(X, w) = (X', w') + \sum_{i=1}^{(w-w')/2}(X_i, 2)$, where (X', w') is a minimal solution of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{c}^T$ and $(X_i, 2)$s are solutions of $(A, -\mathbf{d}^T)(X, w)^T = \mathbf{0}^T$, and $w' = 1$ or 2 according to the parity of w.
III. Design an optimal $(2\mathbf{d})$-HUSB U_0, $(1\mathbf{d} + c)$-HUSB U_1 and $(2\mathbf{d} + c)$-HUSB U_2, see Fig.4(a),(b),(c).
IV. An optimal $(w\mathbf{d} + \mathbf{c})$-SB can be obtained by combining $(w - w')/2$ copies of U_0 and one U_1 or U_2 depending on the parity of w. In particular, $U_2 + U_0$ is an optimal $(4, 5, 6)$-HUSB, and $U_1 + 2U_0$ is an optimal $(5, 6, 7)$-HUSB. See Fig.4(d) and (e). The second example is the design of generic rectangular universal switch boxes with channel density ratio vector $\mathbf{d} = (1, 2, 1, 2)$ and residual vector $\mathbf{c} = (0, 0, 0, 0)$. Following the design scheme, we obtain $p = 2$ and $D = \{1, 2\}$. Since $\mathbf{c} = \mathbf{0}$, we only need to design two prime switch boxes $(2, 4, 2, 4)$-USB U_0 and

$(1, 2, 1, 2)$-USB U_1. Fig. 5(a) and (b) show the optimal design of the prime switch boxes, which can be used to construct optimal $(w, 2w, w, 2w)$-USBs for all $w \geq 3$.

6 Experimental Results

In the experiment, we focus on the simple issue: what could be the routability difference on entire-chip routings between FPGAs adopting optimal irregular switch boxes, or other random but basically reasonable irregular switch boxes?

Table 1. Comparison of VPR experimental results on channel density w between disjoint like $(w, 2w, w, 2w)$-SBs and our optimal $(w, 2w, w, 2w)$-USBs.

	Disjoint-like	Optimal Design		Disjoint-like	Optimal Design
alu4	7	7	ex5p	11	10
apex2	8	8	frisc	10	9
apex4	10	9	misex3	9	8
bigkey	5	5	s298	6	6
clma	9	9	s38417	6	5
des	6	5	s38584.1	6	6
diffeq	6	6	seq	9	8
dsip	5	5	spla	10	10
elliptic	10	9	tseng	5	5
ex1010	8	7	e64	6	6
Total				152	143 (-6.3%)

Direct experimental comparisons with other previous works are basically not available, since the result given in [1] was global routing only, and the switch density used in [10] is quite different from ours.

Here we give the experimental test for our $(w, 2w, w, 2w)$-USB designs. We revise the well considered, effective, and fair FPGA router VPR [2] and run large MCNC benchmark circuits for our routing experiments. The logic block structure for our VPR runs is set to consist of one 4-input LUT and one flip-flop. The input or output pin of the logic block is able to connect to any track in the adjacent channels, i.e. $F_c = w$ (or 2w for wide sides). A reasonable switch design with the same switch count, which is an extension of the known disjoint-like (Fig. 5(c)) switch structure, is adopted for comparison.

Fig. 5(d) illustrates our proposed optimal S-box structure and its corresponding routing result. As shown in Table 1, the switch box design optimality does matter. FPGAs adopting the optimal switch box design can save 6% switch resources according to this experiment.

7 Conclusions

We presented a Divide and Conquer method for designing a wide range of irregular switch boxes. That is, an arbitrarily large optimal irregular switch box can be constructed by a simple disjoint union of some smaller prime switch boxes. To achieve this, we expressed a routing requirement as an integer vector satisfying

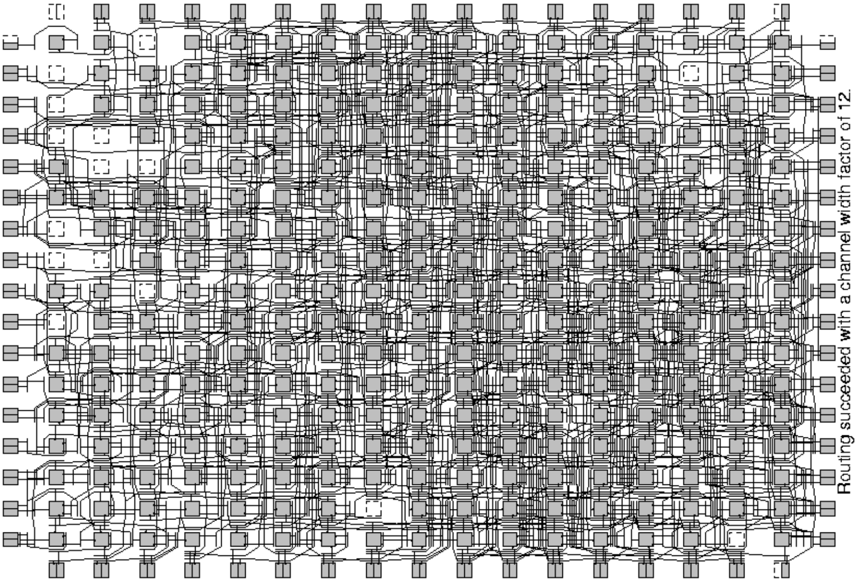


Fig. 6. Routing result of **e64** by using Optimal S-Box, w=6 on $(w, 2w, w, 2w)$-USB

a System of Linear Diophantine Equations (SLDE). By applying the theory of SLDE, we solved the generating problem of routing requirements and proved a general decomposition theorem, which established our reduction design scheme: first design a few prime switch boxes, then use them to build others. As a direct consequence, a switch box designed in this way has a linear number of switches and a linear time detailed routing algorithm.

References

1. V. Betz and J. Rose. Directional bias and non-uniformity in FPGA global routing architectures. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 652–659, Washington, Nov. 10–14 1996. IEEE Computer Society Press.
2. V. Betz and J. Rose. "A New Packing, Placement and Routing Tool for FPGA Research". *Seventh International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.
3. V. Betz, J. Rose, and A. Marquardt. *Architecure and CAD for Deep-Submicron FPGAs.* Kluwer-Academic Publisher, Boston MA, 1999.
4. S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field Programmable Gate Arrays.* Kluwer-Academic Publisher, Boston MA, 1992.
5. Y.-W. Chang, D. F. Wong, and C. K. Wong. Universal switch modules for FPGA design. *ACM Transactions on Design Automation of Electronic Systems.*, 1(1):80–101, Jan. 1996.

6. E. Contejean and H. Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Inform. and Comput.*, 113(1):143–172, 1994.
7. H. Fan, J. Liu, and Y. L. Wu. General models and a reduction design technique for FPGA switch box designs. *IEEE Transactions on Computers*, 52(1):21–30, Jan. 2003.
8. H. Fan, J. Liu, Y. L. Wu, and C. C. Cheung. On optimum switch box designs for 2-D FPGAs. In *Proceedings of the 2001 Design Automation Conference (DAC-01)*, pages 203–208, New York, June 18–22 2001. ACM Press.
9. H. Fan, J. Liu, Y. L. Wu, and C. K. Wong. Reduction design for generic universal switch blocks. *ACM Transactions on Design Automation of Electronic Systems*, 7(4):526–546, Oct. 2002.
10. P. Hallschmid and S. Wilton. Detailed routing architectures for embedded programmable logic IP cores. In *in the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 69–74, Monterey, CA, Feb. 2001.
11. S. Nakamura and G. M. Masson. Lower bounds on crosspoints in concentrators. *IEEE Transactions on Computers*, 31:1173–1179, 1982.
12. J. Rose and S. Brown. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE Journal of Solid State Circuits*, 26(3):277–282, Mar. 1991.
13. M. Shyu, G. M. Wu, Y. D. Chang, and Y. W. Chang. “Generic Universal Switch Blocks”. *IEEE Trans. on Computers*, pages 348–359, April 2000.
14. S. J. Wilton. *Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory.* PhD thesis, University of Toronto, 1997.

Dual Fixed-Point: An Efficient Alternative to Floating-Point Computation

Chun Te Ewe, Peter Y.K. Cheung, and George A. Constantinides

Department of Electrical & Electronic Engineering,Imperial College,
Exhibition Road, London SW7 2BT, England.
{ct.ewe, p.cheung, g.constantinides}@imperial.ac.uk

Abstract. This paper presents a new data representation known as *Dual FiXed-point* (DFX), which employs a single bit exponent to select two different fixed-point scalings. DFX provides a compromise between conventional fixed-point and floating-point representations. It has the implementation complexity similar to that of a fixed-point system together with the improved dynamic range offered by a floating-point system. The benefit of using DFX over both fixed-point and floating-point is demonstrated with an IIR filter implementation on a Xilinx Virtex II FPGA.

1 Introduction

Most arithmetic operations implemented on FPGAs use fixed-point arithmetic representations. For applications where a large dynamic range is required, fixed-point representation may result in implementations with very wide bit-width . In contrast, floating-point representation has a much larger dynamic range than fixed-point for a given bit-width, but arithmetic circuits for floating-point numbers are considerably larger and slower than their fixed-point counterparts. In this paper, a new representation known as *Dual FiXed-point* (DFX) is introduced. It combines the simplicity of a fixed-point system with the wider dynamic range offered by a floating point system. Using a single bit exponent which selects two different fixed-point representations, it allows dynamic scaling of signals throughout the system.

The original contributions of this paper are: 1) to propose the new *Dual FiXed-point* (DFX) system; 2) to present the design of basic arithmetic operators using DFX; 3) to demonstrate the use of DFX through the implementation of an IIR filter on a FPGA; 4) to compare DFX with conventional fixed-point and floating-point implementations in terms of area, accuracy and speed.

The paper is organised as follows. Section 2 compares fixed-point and floating-point number systems. The new DFX number system is described in Section 3. Section 4 shows the design of the basic arithmetic functions using DFX and compares their size and speed to those using fixed-point and floating-point. The implementation and performance of an IIR filter in all three number formats are given in Section 5. Section 6 concludes the paper and suggest future work.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 200–208, 2004.

2 Floating-Point Versus Fixed-Point

In fixed-point arithmetic, all numbers are represented by integers, fractions or a combination of both. This is done by partitioning a binary word of n digits into two sets: q digits in the integral part and p digits in the fractional part, satisfying $p + q = n$. In two's complement fixed-point notation, the value of an n-tuple with radix point between q most significant digits and p least significant digits is

$$X = -x_{q-1} \cdot 2^{q-1} + \sum_{i=-p}^{q-2} x_i 2^i \tag{1}$$

The position of the radix point determines the range of the fixed-point number. Throughout this paper the word-length and the radix point of a fixed-point number is denoted as n_p.

Floating-point representation allows designers to attain a large dynamic range without having to scale the signals. Generally, a floating-point number F is represented by the pair (M,E) having the value

$$F = M \cdot \beta^E \tag{2}$$

where M is the significand (or mantissa), E is the exponent and β is the base of the exponent. Typically for digital systems $\beta = 2$.

For all practical systems it is possible to choose a word-length long enough to reduce the finite precision effects to a negligible level, it is often desirable to use as few bits as possible while achieving user-defined output error conditions in order to optimize area, power or speed. Recent work in mulitple word-length optimisation for fixed-point and floating-point systems can be found in Constantinides [1] and Gaffar [2] respectively. Often, fixed-point designs out-perform floating-point designs in overall system-wide cost including area, power and speed [3] as long as its inputs are properly scaled with the appropriate bit-width [4]. However when signals have a large dynamic range, floating-point designs prevail due to its large dynamic range as compared to fixed-point.

Some work has been done attempting to combine the best of the two number formats. Horrocks and Bull [5] used a pseudo floating-point structure for FIR filters while [6] uses a floating-point representation for design parameters. Both methods show good output performance with low complexity but since they inherently run on fixed-point, they do not possess the large dynamic range capability of floating-point. Block floating-point approach [7], commonly used in FFT analysis, provides most of the advantages associated with floating-point realizations with an implementation complexity approaching that of fixed-point. However, block floating-point only scales a block of data; it lacks the dynamic scaling property offered by the proposal in this paper.

3 Dual FiXed-Point

The proposed n-bit Dual FiXed-point (DFX) format consists of an exponent bit E, and $n-1$ bits of a signed significand X, as shown in Figure 1. The exponent selects between two scalings for the significand X, giving two possible ranges for the number. The lower

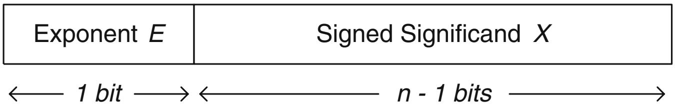


Fig. 1. DFX Format

number range is referred to as $Num0$ while the higher number range is referred to as $Num1$.

In order to achieve two different scalings, we define two radix points p_0 and p_1 such that the radix point of $Num0$ and $Num1$ are p_0 and p_1 bits from the least significant bit respectively, and $p_0 > p_1$.

The value of this DFX number is given by

$$D = \begin{cases} X \cdot 2^{-p_0} & \text{if } E = 0 \\ X \cdot 2^{-p_1} & \text{if } E = 1 \end{cases} \tag{3}$$

A *boundary value*, B, is needed to decide the best scaling to use and hence the value of E. E is determined as follows,

$$E = \begin{cases} 0 & \text{if } -B \leq D < B \\ 1 & \text{if } D < -B \text{ or } D \geq B \end{cases} \tag{4}$$

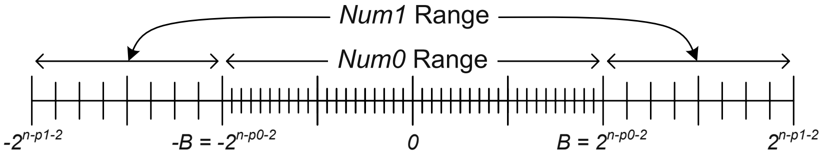


Fig. 2. $Num0$ and $Num1$ range in a DFX Number

In order to simplify the design of the arithmetic units, the boundary value is defined as the next incremental value after the maximum positive number of $Num0$, i.e. $B = 2^{n-p_0-2}$ (-2 because of the exponent and sign bits). The range and precision of Num0 and Num1 are illustrated in Figure 2. To completely define a DFX number, we need n, the size of the DFX number, p_0 and p_1, the radix points. The notation used in this paper is DFX $n\_p_0\_p_1$.

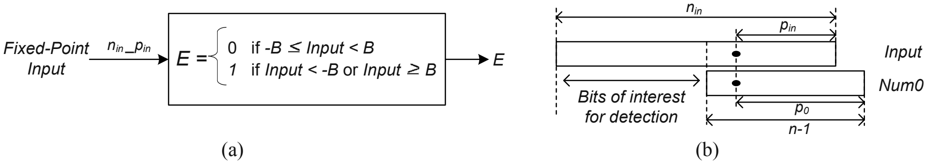
Dynamic range is defined by the ratio between the largest and the smallest absolute number in the data format. The smallest absolute value of a DFX number is 2^{-p0} while the largest absolute value is 2^{n-p_1-2}, hence the dynamic range of a DFP number is given as

$$\text{Dynamic range} = 20 \log_{10}(2^{n+p_0-p_1-2}) \text{ dB} \tag{5}$$

Having two possible scaling for a number gives DFX better range capability than fixed-point as shown in Table 1.

Table 1. Dynamic Range comparisons

Number System	*Dual FiXed-point*	*Dual FiXed-point*	*Fixed Point*	*Floating Point*
Format	32_30_0	32_16_4	32-bit	32-bit IEEE
Dynamic Ranges	$2^{60} \approx$ 361dB	$2^{46} \approx$ 276dB	$2^{31} \approx$ 187dB	$2^{254} \approx$ 1529dB



(a)

(b)

Fig. 3. (a)DFX Range Detector Module and (b) Input Bits the range detector is interested in

4 Dual FiXed-Point Circuits

Arithmetic modules in DFX have been designed in VHDL and mapped onto a Xilinx Virtex II (XC2V80-6fg256) in order to evaluate their area and speed.

4.1 DFX Range Detector

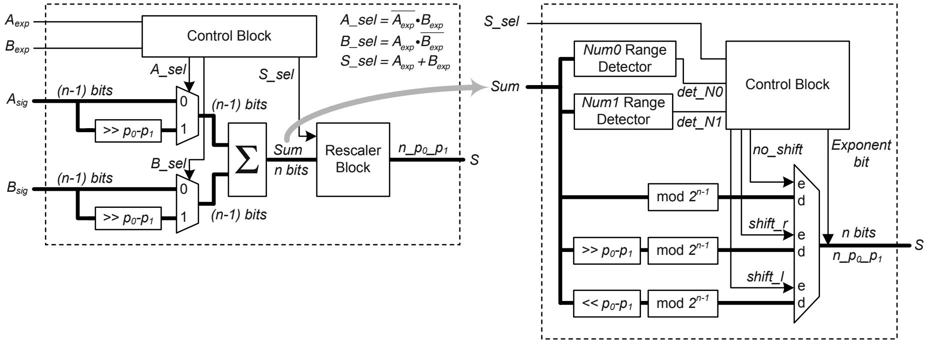
The function of the DFX Range Detector, shown in Figure 3(a), is to generate the exponent bit, E, which selects the range used in the DFX number. The input to this module is a fixed-point number with the format n_{in}_p_{in} (n_{in} being the input word-length and p_{in} being the position of its radix point). The boundary chosen allows this operation to be simplified down to a logic operation given by (6). If the input is in the $Num0$ range, all the bits above the boundary will be 0's (when it is a positive input) or 1's (when it is a negative input) since the input is a two's complement number. The bits of interest for detection are shown in Figure 3(b).

$$\begin{aligned} E = {} & \overline{d_{n_{in}-1} \cdot d_{n_{in}-2} \cdot \ldots \cdot d_{n_{in}-(n-p_0-2)-p_{in}}} \\ & + \overline{\overline{d_{n_{in}-1}} \cdot \overline{d_{n_{in}-2}} \cdot \ldots \cdot \overline{d_{n_{in}-(n-p_0-2)-p_{in}}}} \end{aligned} \qquad (6)$$

4.2 DFX Adder

The DFX Adder module adds together two DFX numbers (see Figure 4(a)). Similar to a floating-point adder, DFX inputs may have to be scaled in order to align the radix points before adding. But unlike floating-point, the number of bits to shift is known *a priori*. Therefore only multiplexers instead of barrel shifters are necessary to perform the necessary scaling. As a result the DFX Adder is expected to be both smaller and faster than the equivalent floating-point adder. Note that ">>" and "<<" are the shift operators which requires only wire routing and mod 2^{n-1} simply extracts the least significant (n-1) bits.

The Adder Control Block determines the shifting of the inputs via the A_sel and B_sel signals. If the input exponents are different, i.e. one input is $Num0$ and the other



(a) (b)

Fig. 4. (a) DFX Adder Module and (b) Rescaler Block

is $Num1$, the $Num0$ number will be shifted up to the $Num1$ range. If both exponents are the same, there will be no shifting.

The adder is a full precision adder and its result sum is fed into the Rescaler Block whose scaling is provided by the signal S_sel. The adder's resulting scale is always $Num1$ unless both the inputs are $Num0$. The output signals of the Adder Control Block are given below in Figure 4.

The Rescaler Block (Figure 4(b) first detects the range of sum with two range detectors that are aligned to the two possible scales. The $Num0$ Range Detector assumes the its input is a $Num0$ number producing the signal det_N0 while the other assumes a $Num1$ input producing the signal det_N1.

No shifting is needed if the adder's result remains in the same range. If the result changes from a $Num0$ to a $Num1$, Sum has to be shifted $p_0 - p_1$ bits to the right and sign extended. The result will however be shifted $p_0 - p_1$ bits to the left and zero padded if the result changes from a $Num1$ to a $Num0$ number. The combinational logic of the internal signals and the exponent bit are given below. Finally, the multiplexer truncates the output to give $(n-1)$ bits for the significand.

$$\begin{aligned} \mathit{no\_change} &= (\overline{S\_sel} \cdot \overline{det\_N0}) + (S\_sel \cdot det\_N1) \\ \mathit{shift\_r} &= \overline{S\_sel} \cdot det\_N0 \\ \mathit{shift\_l} &= S\_sel \cdot \overline{det\_N1} \\ \mathit{Exponent\ bit} &= (\overline{S\_sel} \cdot det\_N0) + (S\_sel \cdot det\_N1) \end{aligned} \tag{7}$$

Table 2 shows the size and speed comparison of a 32-bit adder implemented in all three number formats. It can be seen that while DFX is about 4 times larger and slower than an equivalent fixed-point adder, it is also almost 4 times smaller and faster than the floating-point circuit.

4.3 DFX Multipliers

Two versions of the multiplier have been designed. The DFX-H Multiplier takes one DFX input and one fixed-point input, while the DFX-F Multiplier performs a full multi-

Table 2. Size and latency delay comparison table of 32-bit adders

Adder Type	Size (Slices)	Latency (ns)
Fixed-Point	17	2.5
DFX	64	10.28
Floating-Point (IEEE)	255	34.48

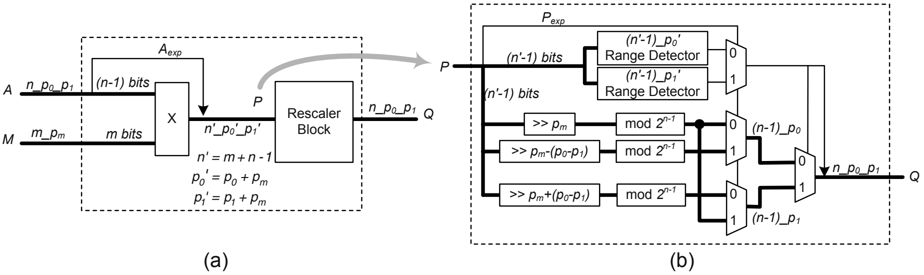


Fig. 5. (a) DFX-H Multiplier module and (b) Rescaler Block

plication between two DFX inputs. Due to space constraints, only the DFX-H multiplier is described here.

Figure 5(a) shows the DFX-H Multiplier forming the product between a DFX input A and a fixed-point input M. This is particularly useful in applications such as filtering where one of the operands is a constant. Unlike the DFX Adder, a DFX-H Multiplier does not require aligning the radix points at the inputs to the binary multiplier. However, the product P needs to be properly scaled and converted into DFX format.

Consider the multiplication of a DFX $n\_p_0\_p_1$ number with a FX m_p_m number, as shown in Figure 5(a), giving a product P which is in DFX $n'\_p'_0\_p'_1$ format, where $n' = m + n - 1, p'_0 = p_0 + p_m$ and $p'_1 = p_1 + p_m$. The product P needs to be converted back to a DFX $n\_p_0\_p_1$ formatted number.

Figure 5(b) show the circuit for the DFX-F Rescaler Block. The range detectors are aligned to the radix points of p'_0 and p'_1 respectively. Further optimization could be done assuming the multiplier M is a constant value.

Table 4 shows the size and speed comparison of 32-bit multipliers implemented in all three number formats. The optimized DFX-H Multiplier is about 1.2 times larger and slower than an equivalent fixed-point multiplier. However it is also about 1.2 times smaller and faster than a floating-point multiplier. The DFX-F Multiplier is comparable with its floating-point counterpart and it is about 1.5 times larger and slower than fixed-point.

4.4 DFX Encoder and Decoder

In order to utilize this number system, a method is needed to convert a number from a known type to DFX. Currently modules exists to encode and decode to and from two's complement fixed-point. The size and latency of the 32-bit DFX modules are given

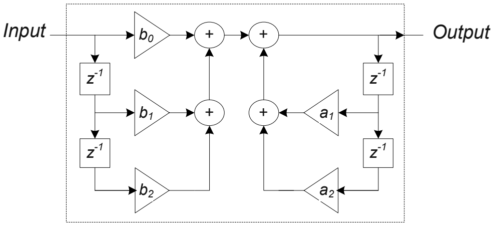


Fig. 6. IIR Filter signal flow diagram

Table 3. DFX Encoder and Decoder size and latency delay table

Module	Size (Slices)	Latency (ns)
Encoder	17.5	7.8
Decoder	10	5.8

Table 4. Multiplier size and latency delay comparison table

Multiplier Type	Size (Slices)	Latency (ns)
Fixed-point	43	13.946
DFX-H	58	17.308
DFX-F	76	19.149
Floating-point	73	20.683

in Table 3. The values for the decoder are approximate because the decoder is usually absorbed into adjacent blocks by logic optimization.

5 Example and Results

The effectiveness of using DFX as an alternative computation method to floating-point is demonstrated by using a Direct Form I implementation of a 2nd order notch IIR filter with the notch at 0.15 of the Nyquist frequency as shown in Figure 6. The filter has five coefficients, three of it in the forward path and two in the feedback path. 32-bit versions of the filter were implemented with DFX (designs D1 and D2), fixed-point (X1, X2 and X3) and floating-point (P1 and P2) formats for comparison and the result is given in Table 7. The DFX Multiplier FX is used in the design since the coefficients are constants.

The target chip for the IIR design is a Xilinx Virtex II XC2V500-6fg456. Comparing the formats with the same bit-width, i.e. 32-bit, DFX designs fall between fixed-point design X1, the smallest and fastest, and floating-point design P1, the largest and slowest. Designs X3 and P2 are about the same size with DFX designs with design X3 being the fastest of the four.

In order to exercise the dynamic range capability of DFX, a set of input data with the frequency distribution as shown in Figure 8(a) and an appropriate spectrum was created. The output SNR, average relative error and maximum relative error of different filter types are shown in Table 5. The error is with reference to double precision floating-point

Filter Type	Design	Format	Size (Slices)	Latency (ns)
DFX	D1	32_18_6	584	52.29
	D2	32_9_6	580	51.28
Fixed Point	X1	32_7	255	24.26
	X2	33_8	272	24.18
	X3	43_18	572	31.51
Floating Point	P1	32bit M23 E8	1459	127.39
	P2	17bit M10 E6	586	88.183

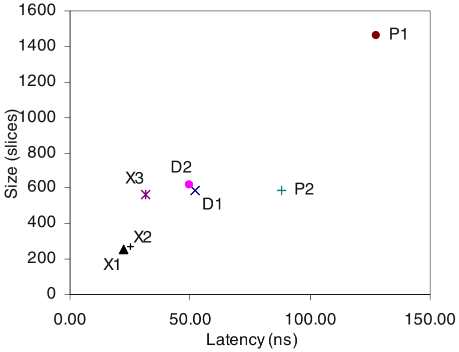
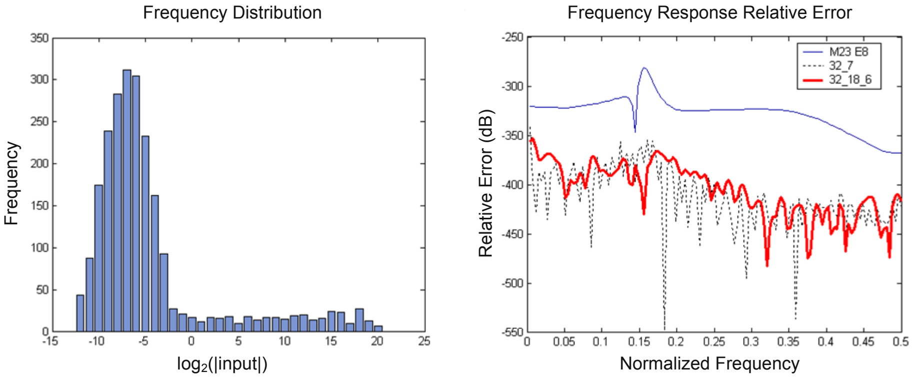


Fig. 7. IIR filters size and latency comparison



(a) (b)

Fig. 8. (a) The frequency distribution of the input and (b) the frequency response relative error for the filter in Fig. 6

Table 5. The error results for the IIR Filter in Fig. 6

Filter Type	Design	Format	SNR (dB)	Av Relative Error (dB)	Max Relative Error (dB)
DFX	D1	32_18_6	333.88	-82.89	-41.02
	D2	32_9_6	347.25	-29.63	13.07
Fixed Point	X1	32_7	330.53	-18.20	24.14
	X2	33_8	344.09	-23.95	17.77
	X3	43_18	482.21	-84.75	-44.21
Floating Point	P1	32bit M23 E8	299.13	-85.90	-29.88
	P2	17bit M10 E6	115.42	-7.24	48.25

results taken to be the expected results. Relative error is calculated as a ratio of the difference error over the reference result.

According to Table 5 floating-point design P1 performs pretty well in terms of relative error (the lower the value the better) but poorly in terms of output SNR. DFX designs outperforms floating-point designs because the DFX design a more precise number number format (i.e. the signifand of design D1 has 31-bits, compared to 24-bits in design P1).

Fixed-point designs show improvement in terms of output SNR and relative error as its word-length increases. In terms of output SNR, DFX design D2 may beat fixed-point design X1, but by increasing the bit-width by one, design X2 is able to out perform design D2. However, design D1 shows good average relative error performance which can only be match by designs X3 and P1. Being a floating-point design, design P1 is notably larger than D1 and fixed-point design X3 is similar in size to D1.

Figure 8(b) shows the relative error of the frequency response measured against the maximum output range of designs with similar word-length. It shows that the floating-point performance is poor overall especially at the notch frequency. The DFX implementation performs similarly to fixed-point but, notably, DFX performs better than fixed-point at the notch frequency.

6 Conclusion and Future Work

This paper demonstrates that by only providing two possible scalings, as in DFX, reduces the design complexity to give smaller and faster designs as compared to floating-point. By choosing the right scaling, DFX can have similar performance to fixed-point while capable of handling a wider dynamic range.

Future work will include the exploration of multiple word-length designs using DFX and the optimization of DFX design for area, accuracy and speed.

References

1. Constantinides, G.A., Cheung, P.Y.K., Luk, W.: Wordlength optimization for linear digital signal processing. IEEE Transactions on CAD of Integrated Circuits and Systems **22** (2003) 1432–1442
2. Gaffar, A.A., Luk, W., Cheung, P.Y.K., Shirazi, N.: Customising floating-point designs. In: IEEE Symposium on Field-Programmable Custom Computing Machines. (2002)
3. Inacio, C., Ombres, D.: The DSP decision: fixed point or floating? IEEE Spectrum **33** (1996) 72–74
4. Oppenheim, A.V., Weistein, C.J.: Effects of finite register length in digital filtering and the fast fourier transform. Proceedings of the IEEE **60** (1972) 957–976
5. Horrocks, D.H., Bull, D.R.: A floating-point FIR filter with reduced exponent dynamic range. In: IEEE International Symposium on Circuits and Systems. (1992)
6. Wust, H., Kasper, K., H.Reininger: Hybrid number representation for the FPGA-realization of a versatile neuro-processor. In: Euromicro Conference. (1998)
7. Oppenheim, A.V.: Realisation of digital filters using block-floating-point arithmetic. IEEE Transaction on Audio and Electroacoustics **18** (1970) 130–136

Comparative Study of SRT-Dividers in FPGA

Gustavo Sutter [1], Gery Bioul [2], and Jean-Pierre Deschamps [3]

[1] School of Engineering, Universidad Autonoma de Madrid, Spain;
gustavo.sutter@uam.es
[2] UNC-Tandil, UFASTA-Mar del Plata, Argentina; gbioul@exa.unicen.edu.ar
[3] Dept. Eng. Electrònica, Elèctrica i Automàtica, Universitat Rovira i Virgili ,Tarragona, Spain

Abstract. This paper describes different implementations of dividers on FPGA. Many division algorithms have been adapted for FPGA technology; nevertheless the peculiar characteristics of re-configurable hardware devices deserve special attention to ensure efficient implementations. This paper presents comparative analyses of implementations targeting Virtex and Virtex II FPGA technology. With respect to the algorithms, selected comparison criteria are operand widths, remainder representations and radix options. At the implementation level, latency, throughput, and area consumption have been traded-off within array, pipelined and sequential options; the results have been compared with previous ones. Area and speed improvements have been obtained thanks to careful implementation techniques. With respect to non-restoring implementations, significant delay improvements - up to 42% - have been achieved.

1 Introduction

In the recent past, division operation has been hard to implement on FPGAs, not only because of the complexity of the algorithms itself but mainly for the limited capacity of the first available FPGA devices. Thanks to a fast growing density feature, efficient division units on FPGA turned to be more feasible. This paper focuses on dividers within floating-point (FP) operations, namely with operands in range [1,2).

A simple and widely implemented class of division algorithm is based on digit recurrence. The most common implementation of digit-recurrence division in modern CPU´s is *SRT-division*, taking its name from the initials of Sweeney, Robertson, and Tocher, who developed the algorithm at approximately the same time. Digit recurrence, specifically SRT, and other division algorithm surveys can be found in [1-5].

Many implementations of SRT dividers on FPGA were recently presented in [6-8]. In [6], dedicated Virtex II multipliers are used to implement radix 2, 4 and 8 SRT dividers; a C++ generator is used to produce a synthesizable VHDL code. Paper [7] presents a minimally redundant radix-8 SRT division scheme; previous results of a radix-4 SRT are pointed out. In [8], radix 2, 4 and 8 SRT division schemes are iteratively implemented as fully combinational and pipelined circuits; they use module generator in JHDL. This paper departs from previous works by several points. First low-level component instantiations in parameterized VHDL code are used in order to keep control over implementation details. Then analyses are presented on maximally redundant radix 2, 4, 8 and 16 SRT dividers with 2's complement expression for the remainder, as well as a radix-2 carry-save remainder. Furthermore array, pipelined, and iterative architectures are evaluated, and then results are compared with the non-

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 209–220, 2004.

restoring division algorithm: one of the easiest to implement, and the one used in CoreGen [9] and others cores. Finally designs are implemented on Virtex and Virtex II devices. Differences in routing architecture, for each device family, lead to slightly different conclusions.

Restoring division algorithm

```
r(0) := X;                      rest_step (a, b, q, r)
for i in 1 .. p loop            z := 2*a - b;
   rest_step(r(i-1),            if z < 0 then
          D,q(i), r(i));           q := 0; r := 2*a;
end loop;                       else  q := 1; r := z;
                                end if;
```

Non-restoring division algorithm

```
r(0) := X;                      nonr_step (a,b,q,r)
for i in 0 .. p-1 loop          if a < 0 then
   nonr_step(r(i),                q := 0; r := 2*a+b;
          D,q(i), r(i+1));      else
end loop;                         q := 1; r := 2*a-b;
q(p):=1; q(0):=1-q(0);          end if;
```

Fig. 1. Restoring and non-restoring division algorithms

2 Algorithms

Given two non-negative real numbers the dividend X and the divisor D ($D\neq 0$), the quotient q and the remainder r are non-negative real numbers defined by the following expression: $X = q.D + r$ with $r < D.ulp$, where *ulp* is the unit in the least significant position. If X and D are the (unsigned) significant of two IEEE-754 floating-point numbers, then they belong to the range [1,2), and q lies in the range [0.5, 1). This result can be normalized by shifting the quotient then adjusting the exponent.

Division generally does not provide finite length result. The accuracy must be defined beforehand by setting the allowed maximum length of the result (p). The number of algorithmic cycles will therefore depend upon the aimed accuracy, not upon the operand length (n).

Restoring and Non-Restoring Algorithm: To divide two integers, the most well known procedures are the restoring and non-restoring digit-recurrence algorithms [3],[4]; the corresponding FPGA implementations are easy and the area/speed results are always better for non-restoring. Figure 1 depicts restoring and non-restoring division algorithm. In the latter one, a correction step should be added in order to correct the last remainder whenever negative.

SRT Division: As other digit-recurrence algorithms, SRT generates a fixed number of quotient bits at every iteration. The algorithm can be implemented with the standard radix-r ($r = 2^k$) SRT iteration architecture presented at figure 2. n-bit integers division requires $t = n/k$ iterations. Two additional steps are required to check input values (division by zero and scaling) and to convert the signed-digit quotient representation to a standard radix-2 notation. The division x/d produces k bits of the quotient q per iteration.

The quotient digit q_j is represented using a radix-r notation (radix complement or sign-magnitude). The first remainder w_0 is initialized to X. At iteration j, the residual w_j is multiplied by the radix r (shifted by k bits on the left, producing $r.w_j$). Based on a few most significant bits of $r.w_j$ and d (n_r and n_d bits respectively), the next quotient digit q_{j+1} can be inferred using a quotient digit selection table (*Qsel*). Finally, the product $q_{j+1} \times D$ is subtracted to $r.w_j$ to form the next residual w_{j+1}.

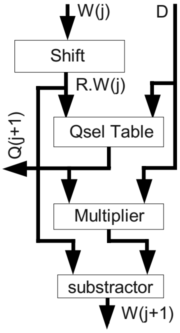


Fig. 2. General SRT division step

Figure 3 exhibits the general architecture; the last step *cond_adder* is necessary for the last remainder adjustment only (not essential in FP implementations). For the hardware implementation of an SRT divider, some important parameters have to be traded off.

Radix $r = 2^k$: For large values of k the iteration number t decreases but each step is getting more complex (larger *Qsel* tables, complex products $q_{j+1} \times D$). Higher radices (larger than 16, k=4) lead to very huge quotient digit selection tables and seem to be impracticable within present FPGA technology.

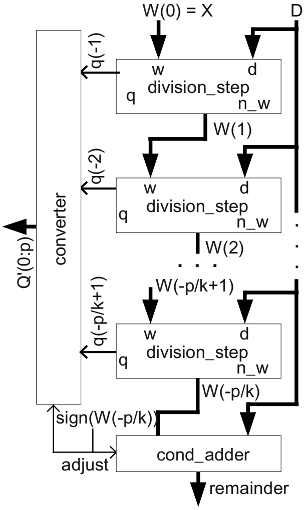


Fig. 3. General SRT array implementation

Residual representation (w_j): Traditionally in ASIC implementation, a redundant number system such as carry-save is used for w_j to accelerate the operation $q_{j+1} \times D$ - $r.w_j$. In FPGA's dedicated carry logic makes traditional ripple-carry faster than carry-save for small bit widths additions or subtractions. Non-redundant system (2's complement) and redundant (carry-save) format are analyzed in what follows.

Quotient representation: For speed up subtractive division redundant digit sets of the form of $\{-\alpha,-\alpha+1,...,0,...,\alpha-1,\alpha\}$ is used. The radix-2^k quotient is represented by a signed-digit redundant number system. It ensures that the next quotient digit determination is possible referring to just a few most significant bits of the remainder and divisor (*nr* and *nd* respectively). Higher values of α lead to simpler quotient digit selection (smaller values of $nr + nd$ for the address of the *Qsel* table) but also to more complex products $q_{j+1} \times d$.

3 FPGA Implementations

Details of FPGA implementation are discussed here. The division algorithms are implemented (i) in a fully combinational way, (ii) pipelined with different logic depth and finally (iii) as sequential implementations with different granularity.

3.1 Array Circuits

The array implementations of different division architectures are analyzed in this section. Traditional restoring and non-restoring algorithms are presented first, then SRT radix 2, 4, 8, and 16 with 2's complement remainder. Finally, a novel implementation of SRT radix-2 with carry-save remainder representation is worked out. Areas and delays are studied for each case. Same division step described in this section are employed later in pipelined and sequential implementation.

3.1.1 Radix-2 Restoring and Non-restoring

Integer division is traditionally dealing with restoring or non-restoring algorithms. Adjusting to fractional operands is trivial. The restoring division algorithm, implemented with the algorithm depicted in figure 1, needs *p restoring_division_step* cells. Each cell uses an (n+1)-bit sustractor and an n-bit 2-1 multiplexer, that means (n+1) slices and a delay of $2.T_{lut} + n.T_{mux\text{-}cy} + T_{net}$. Then, an n-bit divider with p-bit accuracy has an area of $C_{rest}(n, p) = p.(C_{rest\_div\_step}(n)) = p.(n{+}1) = p.n + p$ slices, and a delay of $T_{rest}(n, p) \approx p.(\ T_{rest\_div\_step}(\mathrm{n}) + \mathrm{T}_{net}) = 2.p.\mathrm{T}_{lut} + p.n.\mathrm{T}_{mux} + 2.p.\mathrm{T}_{net}$

Non-restoring division implemented in Virtex / Virtex II FPGA is more efficient. The *nonrestoring_division_cell* is implemented with p (n+1)-bit adder-subtractor. Each cell needs n/2+1 slices and a delay of $T_{lut} + n.T_{mux\text{-}cy}$. If the final remainder is required, an additional conditional adder (n/2 slices) will be needed to adjust the remainder whenever negative.

$$C_{nr}(n, p) = p.(C_{nonr\_step}(n)) + C_{cond\_adder}(n) =$$
$$p.(n/2{+}1) + n/2 = p.n/2 + p + n/2$$

slices.

$$T_{nr}(n,p) \approx p.(T_{nonr\_step}(n) + T_{conec}) + T_{cond\_adder}(n) + T_{net} =$$
$$(p{+}1).T_{lut} + p.(n{+}1).T_{mux} + (n{+}1).T_{net}$$

3.1.2 Radix-2 SRT with 2´s Complement Remainder

If the remainder is represented in 2´s complement notation, the *Qsel* table is trivial (actually doesn't exist). The most significant two bits of the remainder are utilized to settle on the operation to be executed in the next division step (figure 4.a).

Therefore, *Qsel*, the multiplier (by –1, 0 or 1) and the subtractor can be integrated in a single cell (*srt_step_r2*) that means (n+1)/2 slices only. The slice detail of *srt_step_r2* is shown at figure 4.b; the *carry* (-1) is filled with *srn*(0) (the second most significant bit of the previous remainder). The logic function *g(i)* implemented in each LUT is $s_0\overline{d}\,\overline{w} + \overline{s_0}\,\overline{d}w + \overline{s_1}dw + s_1 d\overline{w}$. Delay of *srt_step_r2* cell is given by: $T_{srt\_step\_r2}(n) = T_{lut} + (n{+}1).T_{mux\text{-}cy}$. The complete array implementation of SRT divider is shown at figure 4.c. The *cond_adder* cell adds one dividend D if the last remainder is negative; it uses n/2 slices. The *adjust* signal is the most significant bit of the last remainder. The *converter* cell transforms the quotient digits $q(i) \in \{-1,0,1\}$ in a p-bits 2's complement number; one is subtracted further if the remainder needs adjustment. This cell is implemented as a traditional subtractor, but the first carry is the *adjust* signal, so it takes (p/2+1) slices.

The total area of the SRT radix-2 corresponds to a *p srt_step_r2* cell (p.(n+1)/2 slices), a conditional

W(n:n-1)= srn(1:0)	Remainder Value	Operation	Q(i)
0 0	0 ≤ r < 1/2	Nothing	0
0 1	1/2 ≤ r < 1	Subtr Div	1
1 0	-1< r ≤-1/2	Add Div	-1
1 1	-1/2≤ r < 0	Nothing	0

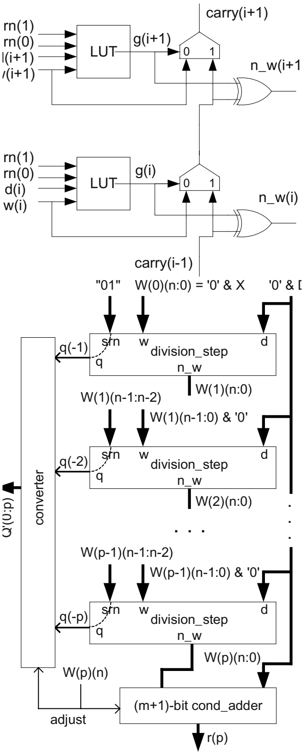


Fig. 4. SRT radix 2 with 2´s complement remainder.
a. Operations to be done in a radix-2 SRT divider cell.
b. Configuration of a Virtex / Virtex II slice for *srt_step_r2* cell. c. Array structure.

adder (*n*/2 slices), and a converter cell (*p*/2+1 slices), which similar to the area of the non-restoring divider.

$C_{srt\_r2}(n,p) = p.(n+1)/2 + n/2 + p/2+1 = (pn+n+2p)/2+1$ slices.

$T_{srt\_r2}(n,p) \approx p.(T_{lut}+(n+1)T_{mux\text{-}cy}+t_{net})+T_{lut}+max(p+1,n+1).T_{mux\text{-}cy}=(p+1).T_{lut}+(p+1)(n+1).T_{mux\text{-}cy}+(p+1).T_{net}$

3.1.3 Radix-4, Radix-8, and Radix-16 SRT with 2's Complement Remainder

In these implementations maximally redundant digit set is used. Therefore, in radix-4 the *Qsel* table uses 5 bits to determine the quotient digit in range {-3,-2,-1,0,1,2,3} according to the PD plot of [3]. The quotient digit code uses 3 bits in a signed value representation, but the sign assumes coding 1 for positive and 0 for negative. The sign bit is calculated directly from the most significant remainder bit *W*(*n*+2), the quotient value is calculated from the next 3 bits of the remainder *W*(*n*+1:*n*-1) and the second bit of the divisor *d*(*n*-2). The *Qsel* table uses a slice configured as a 5-input function generator (using a *F5mux*) to calculate the quotient bit *q*(0), and a LUT to calculate bit *q*(1).

The *srt_step_r4* cell (Figure 5.a), uses the dedicated *mul_and* gate for the 2-bit by *n*-bit multiplier ((*n*+2)/2 slices); the *Qsel* function uses 3 LUTs (2 slices) while the additional adder requires (*n*+2)/2 slices. The critical path includes the *Qsel* calculus, the multiplier carry propagation, the addition cell and connections. The area and delay are $C_{div\_step\_r4} = n+4$ slices, and $T_{div\_step\_r4} = 3.T_{lut} + (n+2).T_{mux\text{-}cy} + 2.T_{net}$ respectively.

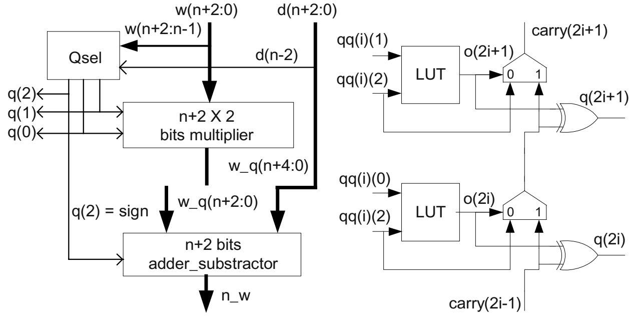


Fig. 5. SRT radix-4 divider a. *srt_step_r4* cell b. Slice detail for converter cell.

The *converter* cell converts signed representation of radix-4 digits to 2's complement. Efficient implementation can be achieved, using inverted logic for the sign digit *qq*(*i*)(2) (0 for negative, 1 for positive). Figure 5.b exhibits the slice detail of the converter, the *carry*(*-1*) input uses the *adjust* signal: zero triggers the subtraction of one from the result. LUT tables implement *o*(2*i*) = *not* (*qq*(*i*)(0)) and *o*(2*i*+1) = *not* (qq(i)(1)) respectively. Finally the cost and delay of the SRT radix-4 divider are:

$C_{srt\_r4}(n,p)=p/2.(C_{step\_r4}(n))+C_{cond\_add}(n)+C_{conv}=p/2.(n+4)+n/2+p/2+1=p.n/2+n/2+3/2.p +1$ slices

$T_{srt\_r4}(n,p)\approx p/2.(T_{step\_r4}(n)+T_{net})+T_{cond\_add}(n)+T_{net}= (3/2.p+1).T_{lut}+(p/2+1).(n+2).T_{mux\text{-}cy} +(2/3.p+1).T_{net}$

In radix-8, *Qsel* table uses 9 bits to resolve the quotient in the digit-set range {-7,..., 7}. The quotient digit is coded in 4-bit sign-magnitude. The first quotient digit bit (sign) *qq*(*i*)(*3*) is directly derived from the most significant remainder digit *W*(*n*+3). Quotient selection is achieved with five digits from the remainder *W*(*n*+2:*n*-2) and three from the divisor D(*n*-2:*n*-4). A total of 33 LUTs packed in 17 slices is needed for *Qsel* table. For *str_step_r8* a 3-bit by (*n*+3)-bit multiplier (*n*+3 slices) and an (*n*+3) adder-subtractor (*n*/2+2 slices) are used. Area and delay of division step in radix 8 are: $C_{div\_step\_r8}$ =3/2.n+22 slices, and $T_{div\_step\_r8}$=7.T_{lut}+(n+3).T_{mux-cy}+6.T_{net}. The circuit, as in radix-2 and -4 needs an (n+1)-bit conditional adder for the remainder (n/2+1 slices) and a 2's complement converter for the quotient ((p+1)/2 slices).

$$C_{srt\_r8}(n,p)=p/3.(C_{step\_r8})+C_{cond\_add}+C_{conv}=p/3.(3/2.n+22)+n/2+(p+1)/2+1=p.n/2+n/2+47/6.p+2 \text{ slices}$$

$$T_{srt\_r8}(n,\ p)\approx p/3.(T_{step\_r8}+T_{net})+T_{cond\_adr}+T_{net}=(7/3.p+1).T_{lut}+(p.n/3+n+2).T_{mux-cy}+(7/3.p+1).T_{net}$$

Radix-16 *Qsel* table uses 12 bits to resolve the quotient digit in range {-15,...,15}. The quotient digit is coded in 5-bit sign-magnitude. The first quotient digit (sign) $qq(i)(4)$ is directly derived from the most significant remainder digit $W(n+5)$. Quotient selection is achieved with seven bits from the remainder $W(n+4{:}n-2)$ and five from the divisor D(n-2:n-6). A total of 268 LUTs packed in 141 slices is needed for *Qsel* table in Virtex, and 221 LUTs packed in 181 slices in Virtex II. The differences between FPGAs families are mainly due to the availability of *muxF7* and *muxF8* in Virtex II. For *str_step_r16* a 4-bit by (n+4)-bit multiplier (3/2.n+6 slices) and an n+4 adder-subtractor (n/2+4 slices) are used. Finally $C_{div\_step\_r16}$=2.n+144 slices and $T_{div\_step\_r16}$ = 8.T_{lut}+(n+4).T_{mux-cy}+6.T_{net}. As in previous radices, a final (n+1)-bit conditional adder, and a sign-magnitude to 2's complement representation *converter* for the quotient are needed. The complete implementation has the following costs.

$$C_{srt\_r16}(n,p)=p/4.(C_{step\_r16})+C_{cond\_add}+C_{conv}=p/4.(2.n+150)+n/2+p/2+2=p.n/2+n/2+36.p+2 \text{ slices}$$

$$T_{srt\_r16}\ (n,p)\approx p/4.(T_{step\_r16}+T_{net})+T_{cond\_adder}+T_{net}=(2.p+1).T_{lut}+(p.n/4+n+2).T_{mux-cy}+(7/4.p+1).T_{net}$$

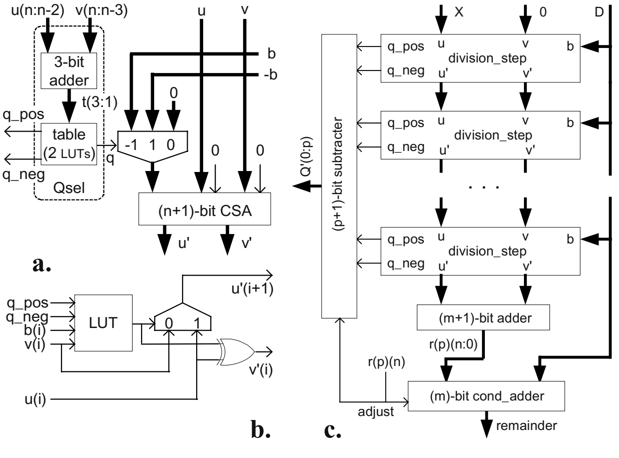


Fig. 6. SRT radix-2 with remainder in carry-save format. a. First version of *Division_step*. b. Detailed implementation of carry-save adder for *Division_step*. c. Array structure.

3.1.4 Radix-2 SRT with Carry-Save Remainder

Block diagram of radix-2 SRT carry-save remainder is shown at 6.c. Two *division_step* alternatives are analyzed. The first implementation is the one suggested in figure 6.a; the first (leftmost) 3 bits of u and v are added, the most significant 3 bits of the result address a table from where *q_pos* and *q_neg* are extracted. It has been established that 3+3 bits from the carry-save representation are adequate to make a proper selection of the quotient digit [11], although 4+4 bits are suggested in [3] and [4]. The multiplexer and carry-save adder (CSA) of figure 6.a can be implemented within (n+1) slices using the cell of figure 6.b. Observe that each CSA digit is calculated with one LUT only (together with a *muxcy* and a *xorcy*), but, due to routing limitations, only one CSA digit can be calculated per slice. Therefore, the division cell area and delay figures are C_{cell_cs1} = n+4 slices; $T_{cell\_cs1} = 3.T_{lut}+3.T_{mux\text{-}cy} +2.T_{ne}+2.T_{xor}$. So, this first version of the radix-2 SRT divider with carry-save remainder, has an area and delay costs given by

$$C_{cs\_v1}(n,p)=p.C_{cell\_cs1}+C_{cond\_add}+C_{conv}=p.n + 4.p + n/2 +1 + n/2+1 = (p+1).n + 4.p + 2 \text{ slices.}$$

$$T_{cs\_v1}(n,p)\approx p.(T_{cell\_cs1}+T_{net})+2.T_{net}+T_{add}+T_{cond\_add}=(3p+2).T_{lut}+(2n+3p+2).T_{mux\text{-}cy}+2p.T_{xor}+(3p+2).T_{net}$$

Area and delay can be improved using a single table to calculate *q_pos* and *q_neg*. A 2^6 x 1 bit memory is necessary for each signal. The *Qsel* cell is implemented using 8 LUTs (4 slices) together with *F5mux*, *F6mux* multiplexers. The second version of SRT carry-save remainder division cell has an area and delay of: $C_{cell\_cs\_lut} = (n + 4)$ slices, and $T_{cell\_cs\_lut} \approx 2.T_{lut} + T_{mux6} + T_{net} + T_{xor} \approx 3.T_{lut} + T_{net} + 2.T_{xor.}$ So, the SRT radix-2 carry-save remainder division with *Qsel* fully implemented in LUTs (*srt_cs_L*) has an area and delay costs given by

$$C_{srt\_cs\_L}(n,p) = p.C_{cell} + C_{con\_adder} + C_{converter} = (p+1).n + 4.p + 1 \text{ slices.}$$

$$T_{srt\_cs\_L}(n,p)=p.(T_{cell}+T_{net})+2.T_{net} +T_{add} +T_{cond\_add}=(3p+2).T_{lut}+(2n+2).T_{mux\text{-}cy}+p.T_{xor}+(2p+2).T_{net}$$

Observe that T_{cs_v1} as well as $T_{cell\_cs\_lut}$ are constant values; so, the computation time of the total divider is a linear function of p (or n if $n > p$).

3.2 Pipeline Circuits

For speed improvement, pipeline is a fruitful technique whenever a great batch of data is dealt with. In this architectural approach each k successive division_step, storage elements (called pipeline registers) are added, so that the longest delay, overall the entire circuit, is shopped down and the frequency can be improved.

Pipeline implementations of non-restoring, SRT radix-2 and -4 with 2´s complement remainder and SRT carry-save remainder for several logic depths have been implemented. In what follows, LD stands for the maximum number of division_step's between successive register banks. The division_step's for each implementation are the ones described in section 3.1.2, 3.1.3, 3.1.4, and 3.1.5 respectively.

In order to implement the register stages flip-flops (FF) distributed in slices are used. Additionally, the quotient digit de-skewing can be implemented using shift-register (SR) implemented in LUT (called SRL16). Look-up table based shift-registers allow the designer to compress up to 16 bits SR in a single LUT unit. Pipelining in FPGA has a low impact in area due to the embedded register distributed into the slices and the SRL characteristics of LUT.

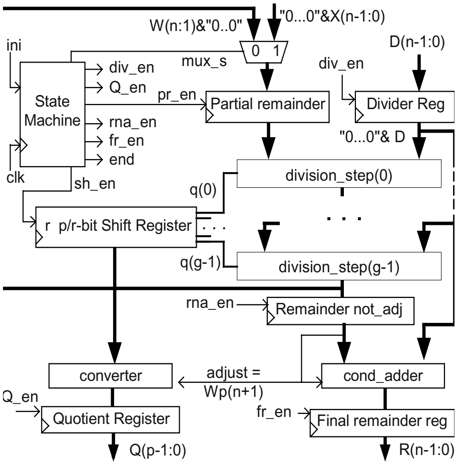


Fig. 7. Sequential implementation of SRT.

3.3 Sequential Circuits

The general architecture adds a state machine to control the datapath it is made of *g* consecutive *division_step*'s with the corresponding register to store intermediate values (figure 7). At each clock cycle, the circuit calculates *g.r* bits, while an extra cycle is necessary to compute the remainder; then a total of *p/*(*g.r*) cycles are used for the complete calculation, with an eventual extra cycle for remainder adjustment.

4 Implementation Results

The circuits are implemented in a Virtex (XCV800hq 240-6) and in a Virtex II (XC2V1000bg575-6). The circuits have been described in VHDL instantiating, when necessary, low level primitives such as LUTs, *muxcy*, *xorcy*, … [12]; Xilinx ISE 6.1 tool [13] and XST [14] for synthesis were used. Same pin assignment, preserving hierarchy option, speed optimization and timing constraints were part of the design strategy. Area and delay results presented below are those reported by Xilinx tools.

The major differences between Virtex and Virtex II implementations are observed in the routing delay. In Virtex, relation logic-route in worst path is averaging (55%, 45%), while in Virtex II, this relation is (63%, 37%). Other differences are observed due to fast connection between slices and *muxF7* and *muxF8* availabilities in Virtex II; this makes the large combinational blocks more efficient. Large *Qsel* tables as in radix-8 and -16 are implemented faster and require less area.

4.1 Results in Array Implementations

Table 1 shows, for Virtex devices, areas (LUTs and slices) and delays (total, due to logic and routing) expressed in ns. The circuits are the ones detailed in section 3.1: restoring and non-restoring (*rest*, *nonrest*); SRT radix-2, 4, 8, and 16 with 2´s complement remainder (srt_r2, . . . , srt_16) and finally, two SRT radix-2 with carry-save remainder representation (with adder *srt_cs_ad* and look-up table *srt_cs_L*).

Up to 24 bits, non-restoring and SRT radix-2 shows best results in delays; for greater operand sizes SRT carry-save remainder (*srt_cs_L*) and SRT radix-4 are the best choices. With respect to areas requirements, SRT radix-2 and non-restoring are always the best. On the opposite, restoring and SRT radix-16 are the worst area consumers. Best results in area × delay merit relation are provided by SRT radix-2 up to 24 bits and SRT radix-4 for greater sizes.

Table 2 exhibits the results for Virtex II. The circuits implemented are the same as above. *Restoring* and *srt_cs_ad* are not shown because of poor results. The architecture proposed in section 3.1.4, i.e. the SRT carry-save remainder (*srt_cs_L*), holds the best delay performance, followed by SRT radix-16 (*srt_r16*). Speed improvement, with respect to non-restoring algorithm, is up to 42.5 %. In area, like in Virtex, SRT radix-2 and non-restoring need the fewest resources. Results in area × delay (#slice × ns) relation point to SRT radix-4 (*srt_r4*) as the best choice.

Table 1. Results for array implementations in Virtex

	N	Area	Delay				N	Area	Delay				N	Area	Delay		
	P	slices	total	logic	route		P	slices	total	logic	route		P	slices	total	logic	route
nonRest	40	880	251.7	129.2	122.5	srt_r4	40	940	243.7	114.7	129.0	srt_cs_ad	40	1802	336.9	132.1	204.8
	32	576	180.6	93.3	87.3		32	624	187.8	87.3	100.5		32	1186	259.2	105.0	154.1
	24	336	118.7	62.5	56.2		24	372	125.7	63.4	62.3		24	698	192.8	80.6	112.2
	16	160	68.8	37.8	30.9		16	184	82.4	39.8	42.6		16	338	119.6	54.6	65.2
rest	40	1640	329.1	146.4	182.7	srt_r8	40	1112	277.3	101.1	176.1	srt_cs_L	40	1779	238.6	103.5	135.0
	32	1056	238.3	108.2	130.1		32	804	224.6	83.0	141.6		32	1183	179.2	81.1	98.1
	24	600	158.4	74.5	83.8		24	487	154.6	60.2	94.4		24	695	141.4	59.9	81.4
	16	272	91.5	44.2	47.3		16	243	83.9	41.4	42.5		16	335	87.9	42.8	45.1
srt_r2	40	861	293.2	124.3	168.8	srt_r16	40	2258	245.7	95.5	150.2						
	32	561	198.1	90.8	107.3		32	1666	191.1	75.9	115.2						
	24	325	125.5	60.8	64.6		24	1137	138.4	56.4	82.0						
	16	153	69.2	37.0	32.2		16	676	81.8	36.6	45.2						

Table 2. Results for array implementations in Virtex II

	N	Area	Delay				N	Area	Delay				N	Area	Delay		
	P	slices	total	logic	route		P	slices	total	logic	route		P	slices	total	logic	route
Non-Restoring	56	1680	304.0	205.2	98.8	srt r4	56	3417	231.6	160.1	71.5	srt r16	56	4237	219.9	120.0	99.9
	48	1248	244.9	157.3	87.6		48	2545	194.3	128.5	65.9		48	3429	173.9	99.5	74.5
	40	880	175.5	120.2	55.2		40	921	150.6	101.2	49.5		40	2685	144.3	78.5	65.8
	32	576	123.2	85.4	37.9		32	609	114.4	76.0	38.3		32	2004	109.6	62.7	47.0
	24	336	82.4	56.4	26.0		24	361	82.5	53.9	28.6		24	1388	79.7	44.5	35.1
	16	160	50.1	33.1	17.0		16	177	49.8	33.4	16.4		16	834	53.0	28.9	24.1
srt r2	56	1653	294.7	201.3	93.5	srt r8	56	2508	266.9	145.9	121.0	srt cs L	56	3467	174.8	104.1	70.7
	48	1225	235.8	159.0	76.8		48	1916	190.2	122.1	68.1		48	2545	152.0	92.5	59.5
	40	861	178.8	119.0	59.8		40	1399	152.7	95.1	57.6		40	1802	122.8	74.6	48.2
	32	561	131.4	84.5	46.9		32	1026	129.0	77.1	51.9		32	1186	99.7	62.6	37.1
	24	325	89.1	54.8	34.3		24	652	89.8	53.0	36.8		24	698	74.8	45.6	29.2
	16	153	47.5	32.7	14.8		16	334	54.5	33.0	21.5		16	338	50.4	32.0	18.4

4.2 Results for Pipeline Implementations

The following pipeline implementations have been achieved: non-restoring algorithm, SRT radix-2, SRT radix-2 using SRL (LUT's configured as shift-register), SRT radix-4, and SRT radix-2 using carry-save in remainder. Table 3 shows area expressed in slices, register count and maximum bandwidth in MHz, for 32-bits divider implementations and different logic depth LD (the maximum division steps between successive register banks) in Virtex II. Both architectures show similar results in speed improvement vs. area overhead.

The SRT radix-4 exhibits best results for both device families. In Virtex II, SRT with carry save remainder shows results good in speed but poor in area, meanwhile traditional non-restoring gives valuable results. A maximally pipelined architecture (LD=1) speeds up the system up to more than 20 times with respect to the fully combinational architecture (LD=32) with an area overhead lower than three times.

Table 3. Logic depth, clock cycles, area in slices, register utilization, and maximum bandwidth in MHz for the different architectures in Virtex II.

LD	C	Non-Restoring			SRT radix 2			SRT radix 2 SRL				SRT radix 4				SRT carry save			
		slices	FF	Mhz	slices	FF	Mhz	slices	FF	srl	Mhz	slices	FF	srl	Mhz	slices	FF	srl	Mhz
1	33	2000	2705	182.8	2182	3202	182.2	1720	2244	90	170.1	-	-	-	-	3356	3298	90	119.5
2	16	1286	1328	94.0	1429	1682	85.7	1171	1152	56	78.8	1305	1265	39	118.5	2257	1647	52	69.8
3	11	1067	933	79.8	1123	1089	77.3	1014	835	48	77.5	-	-	-	-	1939	1164	48	71.0
4	8	943	688	55.4	971	794	51.4	916	641	48	52.3	988	631	30	50.5	1745	871	40	48.0
5	7	908	617	50.1	932	717	50.8	891	583	40	50.9	-	-	-	-	1689	780	40	50.7
6	6	869	538	42.1	884	624	43.6	861	521	36	43.6	919	508	36	47.4	1629	685	36	53.0
8	4	779	368	30.1	775	414	28.8	788	385	32	29.6	849	414	12	30.1	1508	483	16	32.0
12	3	748	292	22.1	730	327	22.3	-	-	-	-	751	222	-	23.3	1462	392		28.3
16	2	697	208	15.9	677	224	15.7	-	-	-	-	768	226	-	17.1	1355	256		19.1
32	1	656	128	8.0	627	128	8.2	-	-	-	-	849	414	-	8.9	1251	147		10.0

4.3 Results for Iterative Implementations

Table 4 shows results for iterative implementations in Virtex and Virtex II. The amount of bits calculated at a time (G) and the total clock cycles necessary (C) are presented together with slices and register utilization and minimum period and maximum frequency in MHz. Finally the latency in *ns* (L = clock period × number of clock cycles) and the area × latency (AxL) in *ms* × slice are reported. As G grows, the latency decreases at the expense of area. Best latency results are provided by SRT radix-4 in both device families. Minimum value for area × latency figure is obtained for G = 2.

Table 4. Results for iterative circuits in Virtex and Virtex II.

	G	C	Virtex						Virtex II					
			slic	FF	P(ns)	F	L(ns)	AxL	slic	FF	P(ns)	F	L(ns)	AxL
non_rest	1	32	113	203	8.9	112	285.7	32.3	109	203	4.9	205	155.5	17.0
	2	16	124	202	13.7	72	219.8	27.3	123	202	8.7	115	138.7	17.1
	4	8	155	200	23.8	42	190.2	29.5	155	200	16.4	61	131.2	20.3
	8	4	219	196	44.9	22	179.6	39.3	219	196	31.8	31	127.2	27.8
srt_r2	1	32	135	240	8.0	124	256.9	34.7	127	237	4.9	204	156.4	19.9
	2	16	139	236	13.4	74	214.8	29.9	141	236	8.5	117	136.7	19.3
	4	8	169	236	24.1	41	193.0	32.6	171	236	15.8	63	126.4	21.6
	8	4	229	236	47.9	20	191.7	43.9	231	236	30.1	33	120.3	27.8
srt_r4	2	16	134	221	12.7	78	202.9	27.2	184	219	7.9	127	125.8	23.2
	4	8	169	221	21.9	45	175.2	29.6	170	221	14.5	68	116.2	19.7
	8	4	240	222	41.2	24	164.6	39.5	240	221	28.6	34	114.5	27.5
_r16	4	8	336	255	23.0	43	183.9	61.8	366	251	14.0	71	111.7	40.9
	8	4	603	294	41.9	23	167.5	101.0	635	293	28.3	35	113.1	71.8
srt_cs	1	32	179	269	11.9	83	382.0	68.4	174	267	7.0	143	223.0	38.8
	2	16	210	267	17.7	56	282.6	59.3	210	267	9.9	101	157.7	33.1
	4	8	282	267	29.9	33	239.3	67.5	312	267	15.0	66	119.8	37.4
	8	4	426	267	52.5	19	210.2	89.5	425	267	29.7	33	118.8	50.5

4.4 Comparison

The results of the implementations presented in this paper are compared with other recent contributions (figure 8). From the experiments, array implementations of SRT radix-2. -4 and -16 with 2's complement remainder (*srt_r*2, *srt_r*4, *srt_r*16) and the carry-save remainder representation (*srt_cs_L*) are charted. Finally, radix-4 and -8 SRT implementation results from [7] (*l_r*4 and *l_r*8), best result of [6], a radix-8 SRT using embedded Virtex II *mult*18x18 blocks (*b_r*8), and a 24-bit array implementation of radix-2 and -4 SRT dividers of [8] (*w_r*2. *w_r*4) complete the figure 8. Figure 9 shows the area-delay-latency trade-off for some of the 32-bit dividers presented in this paper. The sequential implementation shows best area performance, while pipeline exhibits best delay with a relatively low area penalty with respect to the array implementation, but with an initial latency.

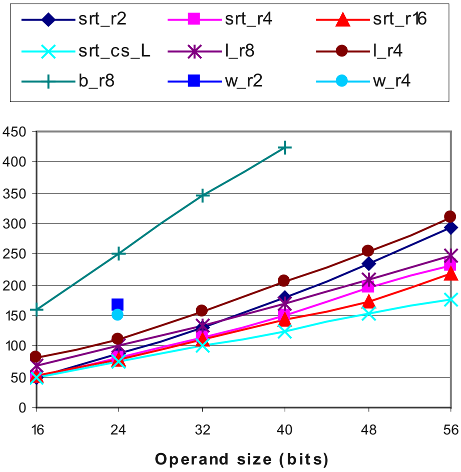


Fig. 8. Delay in *ns* as a function of operand size for some divider implementations.

5 Conclusions

This paper has presented improved architectures and implementations of SRT dividers. The optimizations have been targeted for Virtex and Virtex-II FPGAs families. The slight difference in slice architecture, and the more important one in routing characteristics of the FPGAs families under review, makes the specific conclusions somewhat different from each others.

The circuits presented have been implemented in VHDL instantiating low-level primitives whenever necessary. Latency, throughput, and area are traded off within arrays, pipelined and sequential implementations. Improvements have been shown with respect to recent contributions where high-level techniques were used.

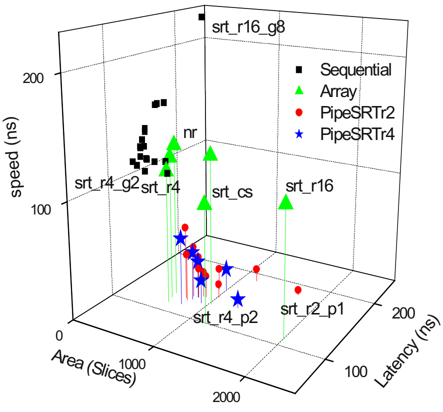


Fig. 9. Delay-latency-area trade-off for different architectural approaches.

In Virtex II for array implementation, the lower influence of routing delays makes SRT with carry-save remainder - proposed in section 3.5.1 - the best choice, improving the delays with respect to non-restoring algorithms by up to 42.5 %.

Further research is needed to explore carry-save remainder representation with higher radices. The power consumption features are currently under study. An optimized fully IEEE compliant floating-point unit is another key research interest.

Acknowledgement. This work has been supported of project number 07T/0052/2003-3 of the *Consejería de Educación de la Comunidad Autónoma de Madrid*. Spain.

References

1. S.F. Oberman and M.J. Flynn. "Division algorithms and implementations". *IEEE Transactions on Computers*. 46(8):833–854. August 1997.
2. M.D. Ercegovac and T. Lang. *Division and Square-Root Algorithms: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic. 1994.
3. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Design*. Oxford University Press. 2000.
4. Milos.D.Ercegovac and Tomas Lang. *Digital arithmetic* San Francisco. California: Morgan Kaufmann. cop. 2004
5. P.Soderquist and M.Leeser. *Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementatio*ns. ACM Computing Surveys. Vol. 28. No. 3. Sept 1996.
6. J-L Beauchat and A. Tisserand. "Small Multiplier-Based Multiplication and Division Operators". *12th Conference on Field Programmable Logic and Applications*. pp. 513-522. 2002.
7. B.R. Lee and N. Burgess. "Improved Small Multiplier Based Multiplication. Squaring and Division" *11th Annual IEEE symp. on Field-Program. Custom Computing Machines.* 2003
8. X. Wang and B.E. Nelson. "Tradeoffs of Designing Floating Point Division and Square Root on Virtex FPGAs" *11th IEEE symposium on FCCM'03*. 2003
9. Xilinx Inc. "Logicore: Pipelined Divider V2.0" available at www.xilinx.com. June 30. 2000.
10. Xilinx Inc. Development System Reference Guide: chp10 PAR (Place & Route). 2003
11. G.Sutter. "FPGA implementation of SRT dividers". UAM. Technical Report. Dec. 2003.
12. Xilinx Inc. Libraries Guide for ISE 6.1 available at www.xilinx.com. 2003.
13. Xilinx Inc. Xilinx ISE 6 Software Manuals. available at www.xilinx.com. 2003.
14. Xilinx Inc. XST User Guide 4.0. available at www.xilinx.com. June 2003.

Second Order Function Approximation Using a Single Multiplication on FPGAs

Jérémie Detrey and Florent de Dinechin

Laboratoire de l'Informatique du Parallélisme
École Normale Supérieure de Lyon
46, allée d'Italie, 69364 Lyon cedex 07, France
{Jeremie.Detrey,Florent.de.Dinechin}@ens-lyon.fr
http://www.ens-lyon.fr/LIP/Arenaire/

Abstract. This paper presents a new scheme for the hardware evaluation of elementary functions, based on a piecewise second order minimax approximation. The novelty is that this evaluation requires only one small rectangular multiplication. Therefore the resulting architecture combines a small table size, thanks to second-order evaluation, with a short critical path: Consisting of one table lookup, the rectangular multiplication, and one addition, the critical path is shorter than that of a plain first-order evaluation. Synthesis results for several functions show that this method outperforms all the previously published methods in both area and speed for precisions ranging from 12 to 24 bits and over.

1 Introduction

The evaluation in hardware of elementary functions such as sine/cosine, exp, log, or more complex functions has been an active research subject over the last decade. Applications include digital signal processing, but also neural networks [15], logarithm number system [5], and the initialization of Newton-Raphson iterations for hardware division [11] among many others.

The simplest hardware evaluator is a lookup table storing precomputed values. Its size grows exponentially with the size of the input word, which confines this solution to input precisions smaller than 10 bits. The table size can be reduced by using a piecewise linear approximation of the function. The hardware now requires a multiplier [10], but the bipartite trick and its variations [2,14,12, 3] allow to replace the multiplier with an adder, which improves both area and speed. These methods allow for practical input precisions up to 20 bits. For more precision, approximations by higher order polynomials are needed [7], using either more multipliers, or iterations over a single multiplier, with an increased delay. Variations on these higher order methods include partial product arrays [6], parallel powering units [13,9], and difference formulas [8,1].

This article presents a second order method which involves only one small rectangular multiplication. This method is well suited to input precisions from 10 to 24 bits and over. It is simpler and more flexible than previous similar work [4],

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 221–230, 2004.

allowing complete automation of the synthesis of operators for arbitrary functions and arbitrary input and output precision. This scheme also outperforms the other previously published methods listed above in both area and speed.

Notations. Throughout this paper, we discuss the implementation of a function whose inputs and outputs are in fixed-point format. We note w_I and w_O the required input and output size (in bits). Without loss of generality, we will focus in this paper on functions with both domain and range equal to $[0;1[$. Thus any input word X is written $X = .x_1x_2\cdots x_{w_I}$ and denotes the value $\sum_{i=1}^{w_I} 2^{-i}x_i$. Similarly an output word is written $Y = .y_1y_2\cdots y_{w_O}$.

2 The SMSO Approximation Scheme

2.1 General Idea

The main idea behind the Single Multiplication Second Order method (SMSO) is to consider a piecewise degree 2 polynomial approximation of the function f. The input word X is thus split into two sub-words A and B of respective sizes α and β, with $\alpha + \beta = w_I$ (see Figure 1) :

$$X = A + 2^{-\alpha}B = .a_1a_2\cdots a_\alpha b_1b_2\cdots b_\beta.$$

The input domain is split in 2^α intervals selected by A. On each of these intervals, f is approximated by a second order polynomial:

$$\begin{aligned} f(X) &= f(A + 2^{-\alpha}B) \\ &\approx K_0(A) \;+\; K_1(A) \times 2^{-\alpha}B \;+\; K_2(A) \times 2^{-2\alpha}(B - \tfrac{1-2^{-\beta}}{2})^2. \end{aligned}$$

Remark: we need the parabolic component to be centered in the interval so that we can exploit symmetry later on.

We can then split B into two sub-words B_0 and B_1 of respective sizes β_0 and β_1, with $\beta_0 + \beta_1 = \beta$ (see Figure 1). In other words $B = B_0 + 2^{-\beta_0}B_1$. This gives:

$$\begin{aligned} f(X) \approx\; & K_0(A) \\ & + K_1(A) \times 2^{-\alpha}B_0 \;+\; K_1(A) \times 2^{-\alpha-\beta_0}B_1 \\ & + K_2(A) \times 2^{-2\alpha}(B - \tfrac{1-2^{-\beta}}{2})^2. \end{aligned} \qquad (1)$$

We decide to tabulate as follows:

- A *Table of Initial Values*: $\mathrm{TIV}(A) = K_0(A)$;
- A *Table of Slopes*: $\mathrm{TS}(A) = 2^{-\alpha}K_1(A)$;
- Two *Tables of Offsets*: $\mathrm{TO}_1(A, B_1) = 2^{-\alpha-\beta_0}K_1(A) \times B_1$ and $\mathrm{TO}_2(A, B) = 2^{-2\alpha}K_2(A) \times (B - \frac{1-2^{-\beta}}{2})^2$.

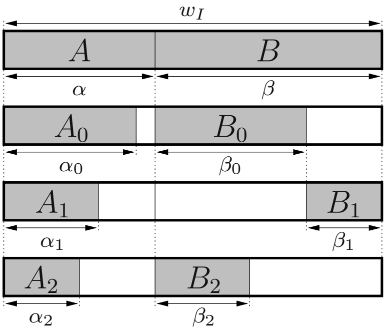


Fig. 1. Decomposition of the input word X.

We then have:

$$f(X) \approx \ \mathrm{TIV}(A) \ + \ \mathrm{TS}(A) \times B_0 \ + \ \mathrm{TO}_1(A, B_1) \ + \ \mathrm{TO}_2(A, B)$$

where there is only one multiplication, the rest being table lookups and additions.

In this scheme so far, the approximation error is only due to the initial polynomial approximation. Remark, however, that the relative accuracies of the various terms are different, due to the powers of two in Eq. 1. We may therefore degrade the accuracy of the most accurate terms (the least significant ones), to align it on the least accurate terms. This is achieved by reducing the number of bits addressing the various tables, which will reduce their size:

- The TS is addressed by $A_0 = .a_1a_2 \cdots a_{\alpha_0}$ the $\alpha_0 \leq \alpha$ most significant bits of A.
- The TO_1 is addressed by $A_1 = .a_1a_2 \cdots a_{\alpha_1}$ the $\alpha_1 \leq \alpha$ most significant bits of A and B_1.
- The TO_2 is addressed by $A_2 = .a_1a_2 \cdots a_{\alpha_2}$ the $\alpha_2 \leq \alpha$ most significant bits of A, and $B_2 = .b_1b_2 \cdots b_{\beta_2}$ the $\beta_2 \leq \beta$ most significant bits of B.

Section 3 will quantify this relation between the approximation error and the various parameters which determine the table and multiplier sizes.

Finally, we get the SMSO approximation formula below, which can be implemented as the architecture depicted Fig. 3:

$$f(X) \approx \mathrm{TIV}(A) + \mathrm{TS}(A_0) \times B_0 + \mathrm{TO}_1(A_1, B_1) + \mathrm{TO}_2(A_2, B_2)$$

2.2 Exploiting Symmetry

As remarked by Schulte and Stine in [14] in the case of the multipartite method, the tables present some symmetry. We have $\mathrm{TO}_1(A_1, B_1) = 2^{-\alpha-\beta_0} K_1(A_1) \times B_1$, which can be rewritten:

$$\begin{aligned} \mathrm{TO}_1(A_1, B_1) &= 2^{-\alpha-\beta_0} K_1(A_1) \times B_1 \\ &= 2^{-\alpha-\beta_0} \left(K_1(A_1) \times (B_1 - \tfrac{1-2^{-\beta_1}}{2}) + K_1(A_1) \times \tfrac{1-2^{-\beta_1}}{2} \right) \end{aligned}$$

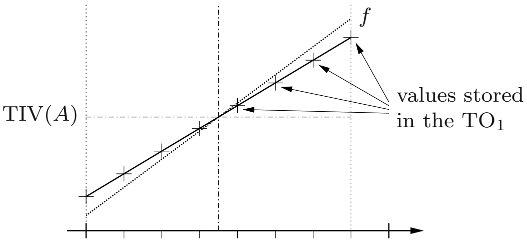


Fig. 2. Example of segment symmetry.

where the term $2^{-\alpha-\beta_0}K_1(A_1) \times \frac{1-2^{-\beta_1}}{2}$ can be added to the value of the TIV. This allows us to use the segment symmetry as depicted in Fig. 2, saving a bit in addressing the TO$_1$ at the expense of a few XOR gates needed to reconstruct the other half of the segment.

The values of TO$_2$ also present symmetry, which allows to divide its size by two as well. In this case the output of the table should not be XORed, as TO$_2$ holds an even function (see Fig. 3).

2.3 Architecture

An example of SMSO operator architecture is given Fig. 3. All the table lookups are performed in parallel. One should also notice that two of the three additions of the adder tree can be performed in parallel to the multiplication. Therefore the critical path is the TS table lookup, the multiplication and the last addition.

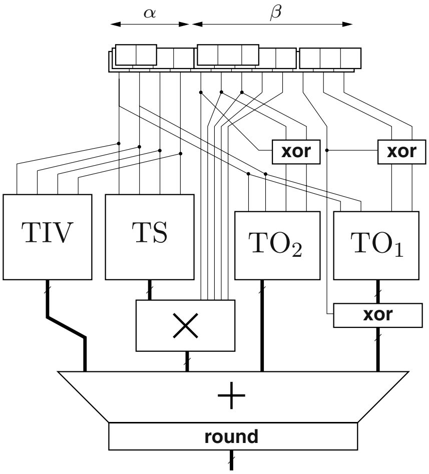


Fig. 3. Architecture of the SMSO operator for $\alpha = 4$, $\beta = 8$, $\alpha_0 = \alpha = 4$, $\alpha_1 = \alpha_2 = 2$, $\beta_0 = 5$, $\beta_1 = 3$ and $\beta_2 = 3$

An important advantage of this scheme is that the multiplier is kept small and rectangular due to the splitting of B. This will lead to efficient implementation on current FPGA hardware with fast carry circuitry (and even more efficient if block multipliers are available).

The remainder of this article shows how to choose the numerous parameters introduced here to ensure a given accuracy bound.

3 Optimisation of SMSO Operators

In the following, we want a SMSO architecture to (classically) provide *faithful* accuracy: The result returned must be one of the two numbers surrounding the exact mathematical result, or in other terms, the total error of the scheme should always be strictly smaller than 2^{-w_O}. However all the following is easily adapted to other error bounds.

The bound on the overall error ϵ of the SMSO operator is the sum of several terms:

$$\epsilon = \epsilon_{\text{poly}} + \epsilon_{\text{tab}} + \epsilon_{\text{rt}} + \epsilon_{\text{rm}} + \epsilon_{\text{rf}},$$

where:

- ϵ_{poly} is the error due to the polynomial approximation, studied in 3.1;
- ϵ_{tab} is the approximation error due to removing bits from the table inputs as shown previously; It is studied in 3.2;
- ϵ_{rt} and ϵ_{rf} are rounding errors, when filling the tables, the product and the final sum; They are studied in 3.3;

In the following, we show how these terms can be computed, depending on the design parameters. An heuristic for optimising a SMSO operator then consists in enumerating the parameter space, computing the error for each value of the parameters, keeping only those which ensure faithful accuracy, and selecting among them the optimal either in terms of speed or of area.

3.1 Polynomial Coefficients – ϵ_{poly}

The coefficients $K_0(A)$, $K_1(A)$ and $K_2(A)$ are computed on each of the 2^α intervals as a minimax approximation based on the Remez algorithm[11]. This method provides us with the 3 coefficients along with the value of ϵ_{poly}. To cut the exploration of the parameter space, we may remark that this error is obviously bounded by the second order Taylor approximation error:
$\epsilon_{\text{poly}} \leq \frac{1}{6} 2^{-3\alpha-3} \max_{X\in[0;1[} |f'''(X)|$

3.2 Reducing Table Input Sizes – ϵ_{tab}

Removing $\alpha - \alpha_i$ bits from the input of one table means imposing a constant table value over an interval of size $2^{\alpha-\alpha_i}$. As the content of the table is usually monotonous, the value that minimises the error due to this approximation is the mean of the extremal values on this interval, and the error induced is then the half of the distance between these extremal values, suitably scaled according to Eq. 1.

The symmetry reduction described in Section 2.2 to halve the size of the TO_is entails no additional approximation error.

3.3 Rounding Considerations – ϵ_{rt} and ϵ_{rf}

Unfortunately, the tables cannot be filled with results rounded to the target precision: Each table would entail a maximum rounding error of 2^{-w_O-1}, exceeding the total error budget of 2^{-w_O}. We therefore fill the TIV and the TO_is with a precision greater than the target precision by g_0 bits (guard bits). Thus rounding errors in filling one table is now $2^{-w_O-g_0-1}$ and can be made as small as desired by increasing g_0. For consistency of the final summation we chose to round the output of the multiplier to g_0 bits as well, by truncating it and adding half a bit to the value in the TIV before rounding . Thus the total error due to these four roundings is bounded by $4 \times 2^{-w_O-g_0-1} = 2^{-w_O-g_0+1}$.

The output of the TS table is not concerned by the previous discussion, and we may control its rounding error by another number of guard bits g_1. This entails another rounding error that adds up with the summation errors. Finally we have:

$$\epsilon_{\mathrm{rt}} = 2^{-w_O-g_0+1} + 2^{-w_O-g_1-1} \quad .$$

The final summation is now also performed on g_0 more bits than the target precision. Rounding the final sum to the target precision now entails a rounding error up to $\epsilon_{\mathrm{rf}} = 2^{-w_O-1}$. A classical trick due to Das Sarma and Matula [2] allows to improve it to $\epsilon_{\mathrm{rf}} = 2^{-w_O-1}(1 - 2^{-g_0})$.

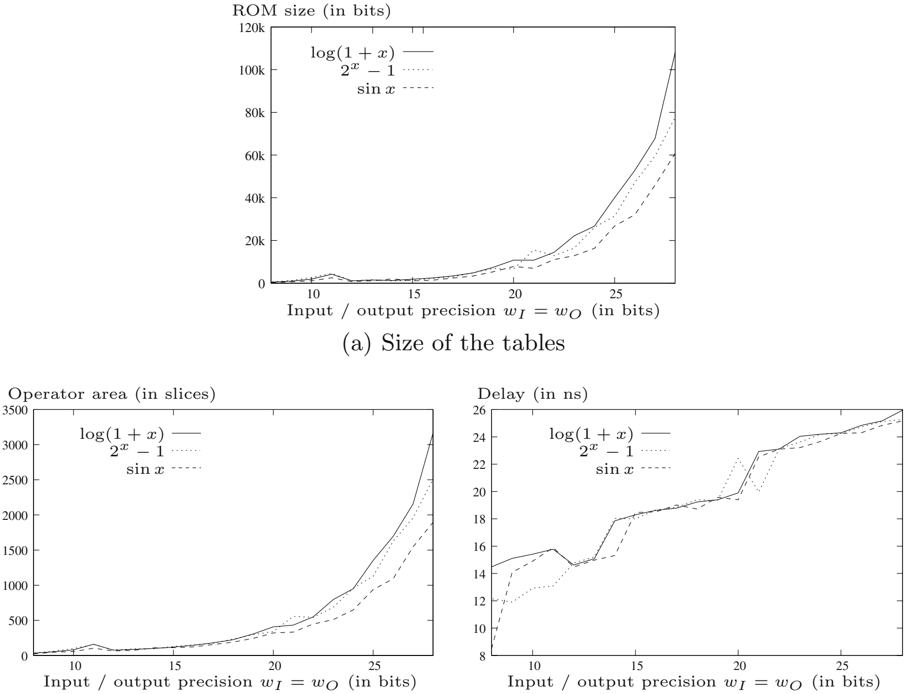
Note that this discussion has added another two parameters g_0 and g_1 to the SMSO architecture, but setting $g_1 = g_0$ (so that the result of the multiplication does not need any rounding) gives a formal expression for g_0. A trial-and-error method can be then applied to decrease g_0 and g_1 to finely tune the operator.

There is an implicit implementation choice in the previous error analysis, which is that we use an exact, full-precision multiplier. Another option would be to truncate the multiplier hardware directly. Our choice is obvious when targetting FPGAs with small multipliers, like the Virtex-II. It also makes sense in the other cases, as it allows to cleanly express the error as a function of the parameters. Besides, the expected gain in using a truncated multiplier is less than half the size of the multiplier, which is itself small compared to the tables as Section 4 will show. Therefore this choice seems justified *a posteriori*.

4 Results

4.1 ROM Size, Area, and Delay Estimations

In this section we give estimations of area and critical path delay for varying precisions (with $w_I = w_O$) for the following three functions:



(a) Size of the tables

(b) Operator area

(c) Critical path delay

Fig. 4. Area and delay of some SMSO operators (without using block multipliers).

Table 1. Impact of using the Virtex-II 18 × 18 block multipliers.

Function		$\log(1+x)$			$\sin x$		
Precision ($w_I = w_O$)		16 bits	20 bits	24 bits	16 bits	20 bits	24 bits
Multiplier bit size		8 × 11	8 × 14	14 × 17	8 × 13	8 × 14	14 × 19
not using block multipliers	area (slices)	148	419	981	124	332	671
	delay (ns)	21	22	27	19	21	25
using block multipliers	area (slices)	102	362	855	71	275	540
	delay (ns)	18	21	25	19	21	25

- The natural logarithm: $\log(1+x) : [0;1[\to [0;1[$;
- The power of 2: $2^x - 1 : [0;1[\to [0;1[$;
- The sine: $\sin(\frac{\pi}{4}x) : [0;1[\to [0;1[$.

These estimations were obtained using Xilinx ISE v5.2 for a Virtex-II XC2V-1000-4 FPGA. We performed synthesis with and without using the small multipliers embedded in those FPGAs, to compare our results with those of other published works. Only results for combinatorial operators are detailed, as the estimations for pipelined circuits present only slight differences.

Table 2. Compared table size, area and delay of the multiparitite table method [3] and of the SMSO for the $\sin x$ and $2^x - 1$ functions.

Function		$\sin x$					$2^x - 1$
Precision ($w_I = w_O$)		8 bits	12 bits	16 bits	20 bits	24 bits	16 bits
Multipartite	table size (bits)	—	—	7808	—	189440	8704
	area (slices)	19	76	258	1209	4954	283
	delay (ns)	17	18	24	34	43	23
SMSO	table size (bits)	280	720	1376	7808	16288	2208
	area (slices)	21	63	123	321	648	149
	delay (ns)	8	14	19	19	24	19

Fig. 4(a) shows that the combined size of the four tables (TIV, TS and TO$_i$s) grows exponentially with the precision, as expected for a table-based method. Fig. 4(b) closely resembles Fig. 4(a), which indicates that the adders and multiplier contribute only by a small amount to the overall area of the operators. This fact is also underlined by Table 1, which studies the impact on area and delay of using block multipliers: the difference in area corresponds roughly to the area of the multiplier implemented in slices.

Fig. 4(c) shows that the delay of the SMSO operators grows linearly with the precision, as it is dominated by the table lookup delay which is logarithmic in the size of the tables. For precisions up to 28 bits, the SMSO operators can run at frequencies higher than 33 MHz. Pipelined designs in 3 to 4 stages have been successfully tested at 100 MHz. Table 1 shows that using the small multipliers provided by Virtex-II FPGAs speeds up the whole circuit by 10 to 20%.

As a conclusion, implementing these operators on FPGAs provided with small multipliers will bring improvements in both area and speed, but performance is still very close without embedded multipliers, so the method is also well-suited to multiplier-less FPGA families.

4.2 Comparison with Previous Works

We first compare our SMSO scheme to the state of the art in multipartite method [3]. Table 2 shows that, thanks to its 2nd-order approximation, a SMSO operator is always much smaller than its (first-order) multipartite counterpart. On Virtex FPGAs, this gain in size also allows our method to outperform the multipartite method in terms of delay, despite the multiplier in the critical path where the multipartite scheme have only additions.

We also compare our method to the lookup-multiply units developed by Mencer et al. in [10]. The results they publish are obtained on XC4000 FPGAs, which prevents comparing delays. As XC4000 CLBs can be compared to Virtex-II slices, Table 3 shows that the SMSO operators are much smaller than lookup-multiply units, which actually seem less efficient than multipartite ones.

Finally, we want to compare the SMSO scheme with the faithful powering computation developed by Piñero et al. in [13], once again implemented on

Table 3. Area compared to the lookup-multiply method [10] for the $\log(1+x)$ function.

Precision ($w_I = w_O$)	8 bits	12 bits	16 bits	20 bits	24 bits
Lookup-multiply area (XC4000 CLBs)	80	180	560	2000	8900
SMSO area (Virtex-II slices)	33	76	145	407	949

XC4000 FPGAs. Their method uses a squarer unit and a multiplier to compute a second-order approximation, which is probably more generally applicable than what they publish: Their architecture is hand-crafted for powering functions with a precision of 23 bits. Their area estimation (1130 slices, but it is unclear for which function) is roughly the same as those of our method (about 1000 slices, depending on the function), but their critical path is larger, as their operator performs all the additions after the multiplications. Besides the strong point of our method here is its flexibility.

5 Conclusion

We have presented a new scheme for elementary function approximation, based on a piecewise degree 2 minimax approximation involving only one small rectangular multiplication. The method is simple and leads to architectures well suited to modern FPGAs, is suitable for arbitrary differentiable functions and any precision, and performs better in terms of area and speed than all previously published methods for hardware function evaluation in the precision range from 12 to 24 bits and over. For smaller precisions, a simple table or the multipartite method may be more efficient.

We have also developed a simple method to explore the huge parameter space depicted in Section 3, as exhaustively as possible: maximum error, area and delay estimations are quickly computed for each possible choice of parameters, and all the acceptable solutions are sorted according to a user-specified score function. Eventually, the best solutions are effectively built to choose the optimal one. This method runs in less than a minute for a precision of 24 bits.

This work will also lead to improvements in our LNS operator library [5].

References

1. J. Cao, B.W.Y. Wei, and J. Cheng. High-performance architectures for elementary function generation. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, June 2001.
2. D. Das Sarma and D.W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, 1995. IEEE Computer Society Press.
3. F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 128–135, Vail, Colorado, June 2001. Updated version of LIP research report 2000-38.

4. D. Defour, F. de Dinechin, and J.M. Muller. A new scheme for table-based evaluation of functions. In *36th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 2002.
5. J. Detrey and F. de Dinechin. A VHDL library of LNS operators. In *37th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, USA, October 2003.
6. H. Hassler and N. Takagi. Function evaluation by table look-up and addition. In S. Knowles and W.H. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 10–16, Bath, UK, 1995. IEEE Computer Society Press.
7. D-U Lee, W. Luk, J. Villasenor, and P. Cheung. Hierarchical segmentation schemes for function evaluation. In *IEEE Conference on Field-Programmable Technology*, Tokyo, dec 2003.
8. D.M. Lewis. Interleaved memory function interpolators with application to an accurate LNS arithmetic unit. *IEEE Transactions on Computers*, 43(8):974–982, August 1994.
9. A.A. Liddicoat. *High-performance arithmetic for division and the elementary functions*. PhD thesis, Stanford University, 2002.
10. O. Mencer, N. Boullis, W. Luk, and H. Styles. Parametrized function evaluation on fpgas. In *Field-Programmable Logic and Applications*, Belfast, September 2001.
11. J.M. Muller. *Elementary Functions, Algorithms and Implementation.* Birkhauser, Boston, 1997.
12. J.M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, 1999.
13. J. A. Piñeiro, J. D. Bruguera, and J.-M. Muller. Faithful powering computation using table look-up and a fused accumulation tree. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 40–47, Vail, Colorado, June 2001.
14. J.E. Stine and M.J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2):167–177, 1999.
15. S. Vassiliadis, M. Zhang, and J. G. Delgado-Frias. Elementary function generators for neural-network emulators. *IEEE transactions on neural networks*, 11(6):1438–1449, nov 2000.

Efficient Modular Division Implementation
ECC over GF(p) Affine Coordinates Application

Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater

Université Catholique de Louvain, UCL Crypto Group,
Laboratoire de Microélectronique (DICE),
Place du Levant 3, B-1348 Louvain-La-Neuve, Belgium.
{gmeurice,bulens,quisquater}@dice.ucl.ac.be

Abstract. Elliptic Curve Public Key Cryptosystems (ECPKC) are becoming increasingly popular for use in mobile appliances where bandwidth and chip area are strongly constrained. For the same level of security, ECPKC use much smaller key length than the commonly used RSA. The underlying operation of affine coordinates elliptic curve point multiplication requires modular multiplication, division/inversion and addition/substraction. To avoid the critical division/inversion operation, other coordinate systems may be chosen, but this implies more operations and a strong increase in memory requirements. So, in area and memory constrained devices, affine coordinates should be preferred, especially over GF(p).
This paper presents a powerful reconfigurable hardware implementation of the Takagi modular divider algorithm. Resulting 256-bit circuits achieved a ratio throughput/area improved by at least 900 % of the only known design in Xilinx Virtex-E technology. Comparison with typical modular multiplication performance is carried out to suggest the use of affine coordinates also for speed reason.

1 Introduction

Modular arithmetic plays an important role in cryptographic systems. In mobile appliances, very efficient implementations are needed to meet the cost constraints while preserving good computing performances. In this field, modular multiplication has received great attention through different proposals: Mongtomery multiplication, Quisquater algorithm, Brickell method and some others. The modular inversion problem has also been extensively studied. It can be performed using the well-known Euclid algorithm (or any of the binary variants like the Montgomery inverse [10,8]), Fermat little theorem or the recently GCD-free method [6].

The modular division is believed to be slow and has not received a lot of attention because it can be replaced by a modular inversion followed by a modular multiplication. Despite their slowness, these operations are needed in several cases: when creating public-private key pairs for RSA and when deciphering in an ElGamal cryptosystem. Although, the main bottleneck arises when we talk about Elliptic Curve Cryptosystems (ECC).

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 231–240, 2004.

This paper is structured as follows: section 2 reminds the theoretical bases of ECC over GF(p) and the different coordinate systems. We introduce the rewritten algorithm and the reason why it has been chosen in section 3. The main contribution of this paper lies in section 4 where we present our implementation. The opportunity of special adders is discussed and a pipelined architecture is described. Practical results and comparisons with the only known published design are in section 5. A typical modular division design is also introduced, suggesting the use of affine coordinates for their memory and area requirements as much as their computational time. Finally, section 6 concludes the article.

2 Elliptic Curve Operations over GF(p)

An elliptic curve E over $GF(p)$, with p a prime number, is defined as the set of points (x, y) verifying the reduced Weierstraß equation:

$$E : f(X, Y) \triangleq Y^2 - X^3 - aX - b \equiv 0 \bmod p$$

for $a, b \in GF(p)$, each choice of these parameters leading to a different curve. Such a curve is called “non-singular” if its discriminant is different from 0 (this corresponds to three distinct roots). The condition is then rewritten as $4a^3 + 27b^2 \not\equiv 0 \bmod p$.

In ECC, the data to be encrypted is represented by a point P on a chosen curve. The encipherment by the key k is performed by computing $Q = P + P + \cdots + P = kP$. This operation, called scalar multiplication, is usually achieved through the “double and add” method (the adaptation of the well-known “square and multiply” to elliptic curves).

2.1 Point Addition and Doubling in Affine Coordinate

In most cases, the addition of points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ leads to the resulting point $R = (x_R, y_R)$ through the following computation:

$$\begin{cases} x_R = \lambda^2 - x_P - x_Q \bmod p \\ y_R = -y_P + \lambda(x_P - x_R) \bmod p \end{cases}, \lambda = \begin{cases} \frac{y_P - y_Q}{x_P - x_Q} \bmod p & \text{if } P \neq Q \\ \frac{3x_P^2 + a}{2y_P} \bmod p & \text{if } P = Q \end{cases}$$

In some cases, exceptions may arise. When we try to add P to its inverse $-P = (x_P, -y_P \bmod p)$, there is an obvious problem in the computation of λ. This is theoretically tackled by the definition of the point at infinity $\mathcal{O}$. The result is defined as $P + (-P) = \mathcal{O} \Leftrightarrow P + \mathcal{O} = -P$. This means that $\mathcal{O}$ is the identity element. Another issue is the doubling of P when it lies on the x-axis, i. e. when its y-coordinate is 0. As might expected, the result is the point at infinity.

2.2 Using Other Coordinate Systems

In the computation of $Q = kP$, the “double and add” method requires on average $(m - 1)$ doubling steps and $(m - 1)/2$ addition steps according to the Hamming weight of k, where m is the bit length of k. For each of these steps, we need to know the value λ. It can be computed by a modular division or a modular inversion followed by a modular multiplication. As those operations are believed to be slow, one may prefer representing points in another coordinate system [3].

Those kinds of systems allow point addition and point doubling with only multiplications and additions. Inversion is no more intertwined in the elliptic curve operations, but we still need it to convert the final result back to affine representation.

Table 1 contains the timings in modified jacobian[1] and in affine coordinates, where M denotes the time for a modular multiplication, S for a squaring and I for the inversion. $I + M$ can be replaced by a modular division D.

Table 1. Timings of different coordinate systems

	Doubling	Addition
modified jacobian	$t(2\mathcal{J}^m) = 4M + 4S$	$t(\mathcal{J}^m + \mathcal{J}^m) = 13M + 6S$
affine	$t(2\mathcal{A}) = 2M + 2S + I$	$t(\mathcal{A} + \mathcal{A}) = 2M + S + I$

It is now possible to issue the upper bound for the ratio I/M under which the affine coordinates are theoretically more useful. We extract it by asking the average timings for scalar multiplication in affine to be quicker than in jacobian coordinates.

$$\begin{aligned}(m-1)\cdot t(2\mathcal{J}^m) + \tfrac{m-1}{2}\cdot t(\mathcal{J}^m + \mathcal{J}^m) &\geq (m-1)\cdot t(2\mathcal{A}) + \tfrac{m-1}{2}\cdot t(\mathcal{A}+\mathcal{A})\\ 2\cdot(4M+4S) + 13M + 6S &\geq 2\cdot(2M+2S+I) + 2M + S + I\\ 8 &\geq I/M\end{aligned}$$

The last result is obtained assuming a square is performed using the same circuitry as a multiplication. The same kind of criteria may be adapted to the ratio D/M. In this case, we reach $9 \geq D/M$. Of course, those timing considerations must be weighted by their area and memory requirements.

3 Algorithm

All the algorithms practically used for modular division are based on the Extended Binary GCD. The main problem is that a comparison is generally needed in order to determine the next operations to compute. For this reason, some efforts have been made to speed up the operation [2], but we believe that the most clever idea is due to the work of Takagi [11], based on the plus-minus algorithm of Brent and Kung [1]. He reaches the goal of avoiding comparison by replacing it with the inspection of Least Significant Bit (LSB) from shift registers and variables.

Takagi's algorithm is rewritten in algorithm 1. For a m-bit modulus M, it takes between $m + 4$ and $2m + 4$ clock cycles to perform the loading of the

[1] We choose this system since it is the fastest at point doubling.

operands, the modular division and the last correction step. The operations between brackets are performed in parallel.

Algorithm 1 Algorithm for modular division computation.

Inputs: $2^{n-1} < M < 2^n$; $-M < X, Y < M$
Output: $Z = X/Y \bmod M$

```
STEP 1:
  $A \leftarrow 0,\ B \leftarrow Y,\ U \leftarrow 0,\ V \leftarrow X,\ P \leftarrow n,\ D \leftarrow 1$
STEP 1BIS:
  $A \leftarrow A + B,\ B \leftarrow M,\ U \leftarrow U + V,\ V \leftarrow 0$
STEP 2:
  while $P \geq 0$ do
    if $[a_1a_0] = 0$ then $A \leftarrow A/4,\ U \leftarrow MQRTR(U, M)$
      if $D < 2$ then
        if $D = 1$ then $P \leftarrow P - 1$
        else $P \leftarrow P - 2$
      $D \leftarrow D - 2$
    elsif $a_0 = 0$ then $A \leftarrow A/2,\ U \leftarrow MHLV(U, M)$
      if $D < 1$ then $P \leftarrow P - 1$
      $D \leftarrow D - 1$
    else
      if $([a_1a_0] + [b_1b_0]) \bmod 4 = 0$ then $q \leftarrow 1$ else $q \leftarrow -1$
      if $D \geq 0$ then $A \leftarrow (A + qB)/4,\ U \leftarrow MQRTR(U + qV, M)$
        if $D = 0$ then $P \leftarrow P - 1$
        $D \leftarrow D - 1$
      else $D \leftarrow -D - 1,\ A \leftarrow (A+qB)/4,\ B \leftarrow A,\ U \leftarrow MQRTR(U + qV, M),\ V \leftarrow U$
STEP 3:
  $U \leftarrow 0$
STEP 4:
  if $[b_1b_0] \bmod 4 = 3$ then
    if $V \geq 0$ then $Z \leftarrow U - V + M$ else $Z \leftarrow U - V$
  else
    if $V \geq 0$ then $Z \leftarrow U + V$ else $Z \leftarrow U + V + M$
```

The presented algorithm is slightly different from the original. The initialization step has been duplicated (STEP 1BIS) and a reset step (STEP 3) has been added before the correction step (STEP 4) to spare resources within the targeted devices. The extra clock cycles added are negligible with respect to the global executing time.

Instead of using redundant binary representation like Takagi, we decided to use classical binary representation. Indeed, we want to focus on the smallest area requirements and the redundancy roughly twice the amount of hardware needed in the FPGA. With this choice, the timings of the P and D shift registers used in the control part are less critical. So, we decided to replace those registers by small counters, like suggested in [11]. Their sizes are in $\log_2(\log_2(M))$, so the propagation delay is very short. Another fact in aid of counters is that shift registers with different directions and steps consume a lot of resources on FPGA. Finally, counters also enable a slight reduction of the algorithm complexity, leading to a more efficient control structure.

The main feature of this algorithm is the use of P and D counters. The comparison between A and B in commonly used algorithms is replaced by $D = \alpha - \beta$, where α and β are values such that 2^α and 2^β represent the minimums of the upper bounds of $|A|$ and $|B|$ respectively. This substitution reduces the comparison between A and B in the "while" loop to "$|A| > 0$". Instead of investigating all the bits of A, it is replaced by a counter P with sign detection, indicating the minimum of the upper bounds of $|A|$ and $|B|$.

The halving operation $MHLV(T, M)$ of $T \bmod M$ is performed either by $T/2$ or $(T + M)/2$ regarding the parity of T (the LSB). The quartering operation $MQRTR(T, M)$ of $T \bmod M$ depends on the value of $M \bmod 4$. If it equals 1, then the operations to be carried out are $T/4$, $(T - M)/4$, $(T + 2M)/4$ or $(T + M)/4$, accordingly as $T \bmod M$ is $0, 1, 2$ or 3. If $M \bmod 4$ equals 3, the operations are $T/4$, $(T + M)/4$, $(T + 2M)/4$ or $(T - M)/4$ with respect to the value of $T \bmod M$, respectively $0, 1, 2$ or 3.

It should be finally noticed that, when using binary GCD, a division does not slow down the computation compared to the inversion, since U or V are not used in the control part[2].

4 Implementation

In this section, we present two different kinds of implementation. We first introduce the basic sequential architecture and the opportunity of flags precomputation. After, we present an improvement of this architecture by tackling the critical path: the carry chain of the adders. We will show that the best compromise lies in the use of pipelinening instead of carry conditional and select adder.

4.1 Basic Division Architecture

We present here the basic sequential architecture. It is naturally broken up in different distinct parts: the *ControlStage*, the *ABstage*, the *UVstage* and the *P,D* counters.

The ControlStage: The main advantage of the algorithm we use lies in the absence of comparison, serially wired with the main operative part. Fortunately, this improvement implies only a small complexity increase with the use of P and D counters. In the main loop, all the flags only depend on parity bits of variables and on sign of small counters. So, all the flags can be efficiently precomputed, leading to high working frequency.

[2] U is set to 1 when performing an inversion, to X otherwise.

The ABstage: The ABstage operative part is shown in Fig. 1. It consumes mainly 3 Logic Elements (LE, half of a slice) on FPGA for each bit of the modulus. One is used for the two's complement adder/substractor, another is used for the shift right selection and the last one is used for the loading and the swap of registers.

Since the loading step has been duplicated, we can spare one multiplexor, implemented by one LE per bit length, between the logical shifter and the register.

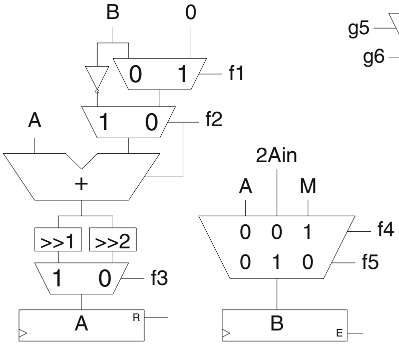


Fig. 1. Stage AB

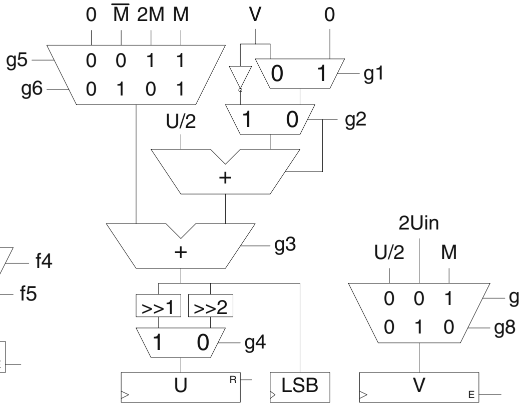


Fig. 2. Stage UV

The UVstage: The UVstage operative part is shown in Fig. 2. This stage mainly consumes 4 Logic Elements per bit length. Two are used for two's complement adder/substractor (M is considered constant), one is used for the shift right selection and the last one is used for the loading and the swap of registers.

We also spare a multiplexor thanks to the duplication of the loading step. The reset step (STEP 3) of the U signal enable resource sharing for the last correction step (STEP 4). However, we always right shift the input of the U register. The shifted bit must be saved in one register to provide the final result.

We should notice that the flags $g3$, $g5$ and $g6$ depend on the 2 LSB computed by the first adder. An efficient solution is to replicate this 2-bit adder to allow flags precomputation in the control part.

The *P* and *D* counters: The P counter is based on an adder and is negative allowed (Fig. 3). The end criteria can be checked with only the sign bit. The D counter is also adder based (Fig. 4). We introduced two additional adder which always compute $D-1$ and $D-2$ to compute all the tests needed in the algorithm.

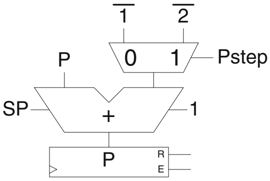


Fig. 3. Counter P

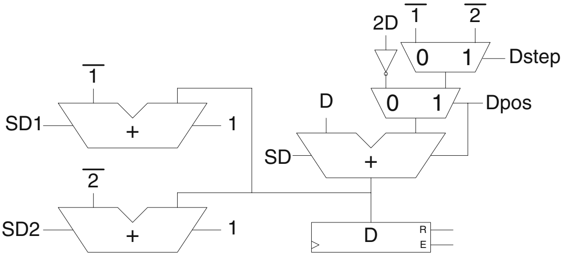


Fig. 4. Counter D

4.2 Special Adder

ECC need computation over important length modulus (typically 160 bits). Area constrained modular division designs are adder based, so it is obvious that the critical path resides in the carry chain of the simple carry-propagate adders.

Modern FPGA are built to efficiently implement the addition operation. High speed performance is achieved by the use of a dedicated optimized carry chain. While FPGAs are structured as an array of programmable logic and routing resources, the carry chain is physically wired and therefore exhibits very low delays. It can be used without constraining the routing as long as the carry chain size does not exceed the column height. For this purpose, adders must be adapted to keep satisfactory speed performance.

This optimized carry chain explains why well known methods like carry-look-ahead, carry-bypass, carry conditional, carry-select and carry-save are practically not really attractive on area constrained reconfigurable devices. Of course, some of them can speed up work frequency, but it is at the cost of a lot of chip area.

Another solution is the pipelining of the addition. It is simply achieved by inserting register in the carry chain and by properly handling the operands. This way, almost no additional hardware is consumed. The drawback is the number of clock cycles needed to fill the pipeline. Nevertheless, this overhead can be negligible if the number of repeated additions is great with respect to the number of pipelined stages. These advantages lead us to choose this method for our improved design.

4.3 Pipelined Division Architecture

In order to speed up the computation, we choose the pipelined architecture. As said in the previous section, this method requires small area increase and can lead to interesting results. It can be used because the flags of the control part are only LSB for the main loop. MSB are only required for the last correction step. So, the total overhead will be twice the number of pipeline stages added. This is negligible compared to the number of clock cycles required for the whole computation.

Number of pipeline stages: The number of pipeline stages must still be determined. Putting the overhead aside, we cannot cut the adders in a lot of small parts. This is due to the behaviour of the algorithm: the two right shifts impose that the LSB of the next stage must be available for the MSB of the current stage. Because the computation is achieved in one clock cycle, we must bring back the LSB asynchronously. This increases the number of Logic Elements through the path leading to the computation of the carry-out.

This new design will be quite place & route dependent. So, it is not easy to theoretically determine the best number of pipeline stages. Nevertheless, approximation can be made to decide after how many bits the carry chain must be cut. After inspecting Fig. 5 and Fig. 6, the carry chain propagation time must be shorter than the delay of the U and V operands arrival plus the carry-in arrival and three serial Logic Element (two for the adders and one for the conditional shifter).

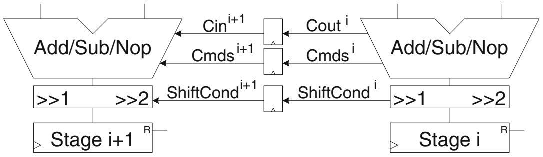


Fig. 5. Typical pipeline organisation

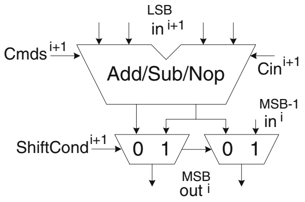


Fig. 6. Modification of the 2 MSB input of each stage

Architecture: To improve readability, all stage features are not reproduced in Fig. 5. The main concern is the conveyance of the shifted bits between pipelined stages. Indeed, the 2 MSB input of each stage (except the last) must be modified with the circuit shown in Fig. 6. The two LSB adders and the conditional shifter have been duplicated to avoid the two serial delays of carry-in arrival and sum return. These two delays appear when we simply take the output from the next pipeline stage adder. With this negligible increase of logic element, we expect to only suffer from the operand arrival delay. All these modifications ensure a faster working design, with only a small increase of hardware and clock cycles.

5 Results

Speed and area comparison of 64, 128, 160 and 256-bit basic and pipelined designs are presented in table 2. The VHDL synthesis and place & route have been achieved on Xilinx ISE 6.2.02i. The first device selected is a Xilinx Virtex-E XCV2000e-6bg560 FPGA (V-E) to guarantee fair comparison with the design [4]. The second is a Xilinx Spartan3 XC3S200-4pq208 (S3) to exhibit performances over a small and low cost FPGA. The two operands are loaded by two 32 bits clocked interface and stored in 32-bit shift registers. The result is registered and then moved in 32-bit shift registers to access the 32-bit clocked output interface.

As explained in section 4.3, the pipelined stages must have a minimal length to be interesting. We decided to pipeline 32-bits adders. The pipelining leads

Table 2. FPGA implementation results

Design	Area V-E (Slices)	Freq. V-E (MHz)	Thr./Area (Kbits/s. Slices)	Area S3 (Slices)	Freq. S3 (MHz)	Thr./Area (Kbits/s. Slices)
64-bit	420 (2 %)	77	88.8	424 (22 %)	77	88
32-bit ×2	460 (2 %)	83	86.1	461 (23 %)	83	86
128-bit	778 (4 %)	55	34.8	873 (45 %)	48	27
32-bit ×4	842 (4 %)	75	42.8	927 (48 %)	79	41
160-bit	951 (4 %)	45	23.3	1112 (57 %)	44	19.5
32-bit ×5	1022 (5 %)	77	36.3	1180 (61 %)	78	31.8
256-bit	1457 (7 %)	29	9.8	1920 (90 %)	29	7.5
32-bit ×8	1612 (8 %)	77	23	1847 (96 %)	80	20.9

to great improvements, especially for the 256-bit basic design, where the carry chain length is too big for the selected FPGA column height.

Throughput/area ratio comparison between our best design and the best design of [4] is given in table 3. To compute the throughput, we consider the worst case of $(2m-1)$ clock cycles for their design and $(2m+4)+2\,ps$ for our design, where ps is the number of pipeline stages. Unfortunately, they do not give the 160-bit design results.

Table 3. Performance comparisons

Design	Freq (MHz)	Throughput (Mbits/s)	Area (Slices)	Thr./area (Kbits/s. Slices)	Improvement
64-bit	45	22.7	1212	18.7	
Our 64-bit	**83**	**39.6**	**460**	**86.1**	**360 %**
128-bit	31	15.6	2215	7	
Our 128-bit	**75**	**36**	**842**	**42.8**	**511 %**
Our 160-bit	**77**	**37**	**1022**	**36.3**	
256-bit	27	13.53	5846	2.3	
Our 256-bit	**77**	**37**	**1612**	**23**	**900 %**

As previously said, it is interesting to compare our modular division implementation with modular multiplication in terms of design performances. We take the 120-bit architecture presented in [5] for its reasonable area requirements.

To enable fair comparison, we implemented our design over the same Virtex1000. Our 128-bit design exhibits a frequency of 74 Mhz and an area of 852 Slices but requires, in the worst case, two times more clock cycles. So, we can roughly say that, with a multiplier work frequency of 88.5 Mhz and an area of 603 Slices, we reached a ratio D/M of 2.5. This result is more than three times as good as the threshold ratio of section 2.2.

6 Conclusion

A powerful modular division implementation has been presented. It has minimal hardware requirements and high work frequency. In addition to the accurate description of our architecture, we discussed the opportunity to use special adder structures (e.g. carry-save, carry-look-ahead) and exhibit that a pipelined carry-propagate adder seems to be the best choice for an area constrained FPGA implementation of the modular division.

We achieved a throughput/area ratio of 23 kbit/(s.Slices) for a 256-bit design. It represents an improvement by at least 900 % of the only known design in Xilinx Virtex-E technology. Using our implementation, affine coordinates for ECC over GF(p) seem to be attractive for their memory and area requirements but also for speed reason. This suggests the use of our design in Elliptic Curve cryptoprocessor of embedded devices.

References

1. R. P. Brent and H. T. Kung, *Systolic VLSI arrays for linear time GCD computation*, VLSI'83, pages 145-154, 1983.
2. S. Chang-Shantz, *From Euclid's GCD to Montgomery Multiplication to the Great Divide.* Technical report, Sun Microsystems Laboratories TR-2001-95, June 2001.
3. H. Cohen, A. Miyaji and T. Ono, *Efficient Elliptic Curve Exponentiation using Mixed Coordinates*, ASIACRYPT 1998, Springer-Verlag, 1998, LNCS 1423, pp. 51–65.
4. A. Daly, W. Marnane, T. Kerins and E. Popovici, *Fast Modular Division for Application in ECC on Reconfigurable Logic*, The 13th International Conference on Field Programmable Logic and Applications — FPL 2003, Portugal, Lisbon, September 1–3, 2003.
5. A. Daly, W. Marnane, *Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic*, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, 2002.
6. M. Joye and P. Paillier, *GCD-free Algorithms for Computing Modular Inverses.* Cryptographic Hardware and Embedded Systems — CHES 2003, Springer-Verlag, 2003, LNCS 2779, pp. 243–253.
7. M. E. Kahaira and N. Takagi, *A VLSI Algorithm for Modular Mulitplication/Division*, The 16th IEEE Symposium on Computer Arithmetic — ARITH 16, Spain, Santiago de Compostela, June 15–18, 2003.
8. B. S. Kaliski Jr., *The Montgomery Inverse and its Applications*, IEEE Transactions on Computers, 44(8), pp. 1064–1065, August 1995.
9. D. E. Knuth, *The Art of Computer Programming*, vol. 2 : Seminumerical Algorithms , 2nd ed., Addison-Wesley, 1981.
10. E. Savaş and Ç. K. Koç, *The Montgomery Modular Inverse - Revisited*, IEEE Transactions on Computers, 49(7), pp. 763–766, July 2000.
11. N. Takagi, *A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm*, IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences, Vol. E81-A, n° 5, pp. 724–728, May 1998.

A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management

Jesus Tabero[1], Julio Septién[2], Hortensia Mecha[2], and Daniel Mozos[2]

[1]Instituto Nacional de Técnica Aeroespacial, 28850 Madrid, Spain
taberogj@inta.es
[2]Universidad Complutense de Madrid, 28040 Madrid, Spain
{jseptien,horten,mozos}@dacya.ucm.es

Abstract. A novel technique is proposed for the management of a two-dimensional run-time reconfigurable device in order to get true hardware multitasking. The proposed technique uses a Vertex List Set to keep track of the available free area, and of the candidate locations to place the arriving tasks. Each Vertex List describes the contour of each unoccupied area fragment in the reconfigurable device. Several heuristics are proposed to solve the problem of selecting one of the vertices to place the task. The heuristic that gives best results is based on a novel fragmentation metric. This metric estimates for each alternative location the suitability of the resulting free device area to accept future incoming tasks. Finally, we show that our approach, with a reasonable complexity, gives better results, in terms of device fragmentation and efficiency, than other techniques.

1 Introduction

The increase in size and density of modern reconfigurable hardware (HW), such as Field-Programmable Gate Arrays (FPGA), together with the appearance of new operational facilities that are summarized in [1], such as the partial run-time reconfiguration (RTR) ability, has made possible in recent years to consider true hardware multitasking. This HW multitasking would be possible not only through time-multiplexing, as it happens with usual SW multitasking, but through space multiplexing. It becomes obvious that resources involved in such HW multitasking should be managed by the same OS that manages SW resources, and that many of the problems involved should be alike to those of SW multitasking, though some others would be specific as [2] adequately shows.

A modern HW resource with partial RTR can be viewed as a large two-dimensional processing area, capable of holding a set of HW tasks. Each task has been previously compiled to a relocatable HW bitmap with the available compilation tools, and can be loaded when asked for execution at a free section of the FPGA. Each HW task can enter or leave the FPGA without affecting the other executing tasks. If the task needs parameters or generates results, their transmission to or from the FPGA should be also dealt with.

One of the main problems stated above is the decision of where to locate each arriving HW task. This decision must be made on-line, and should take into account

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 241–250, 2004.

the resultant area fragmentation to favor the insertion of tasks arriving later-on. In order to make such a decision, some information on the free device area must be kept, and some selection criteria to choose among available alternatives must be devised. The complexity of the techniques used to solve these problems must be kept low in order to be useful.

Next sections will show the main contributions found in the related work and the limitations we find in their approaches, a detailed description of our own approach to RTR HW management and some experimental results that show the validity of our proposal.

2 Related Work

The problems of managing 2D RTR resources, such as allocation, fragmentation or relocation, have been dealt with recently by several research teams.

Diessel et al. [3] have developed a quad-tree structure to store the information of the available FPGA area. Such structure can be travelled and updated quite fast, but it does not guarantee that an adequate place is found, even if there is enough area to store the task, but split among different branches of the tree. This solution doesn't take into account the resultant free area fragmentation to select the position where the task is mapped to. On the contrary, it deals with fragmentation by proposing several alternative high-cost defragmentation processes (local repacking and ordered compaction).

Bazargan et al. [4] deal with the area allocation problem by using a bin-packing approach and applying some of the classical algorithms for such theoretical problem. They propose several strategies for on-line 2D bin-packing of the arriving rectangular tasks. These strategies differ mainly in the way the free area is managed.

One of them keeps track of all the maximum empty rectangles (MER) where an arriving task could be placed. Such approach guarantees that, if an adequate place exists, it can be found. But the complexity of the updating algorithm is too high.

A second approach tries to use heuristics in order to reduce the number of rectangles considered when updating the rectangle list. When a free rectangle is selected to store the arriving task, the excess area is divided in only two, non-overlapping, new rectangles. Bazargan offers several criteria to do this splitting, but does not decide clearly for one of them. Anyway, by selecting some of the possible rectangles, situations can arise where existing room cannot be used to store a task, because it is split among several rectangles.

Finally, Walder et al. use in [5] a typical 2D bin-packing First Fit algorithm, and a Best Fit using a fragmentation formula that is applied to every available location when looking for a place for an arriving task. Thus each alternative location is evaluated, and the one generating the FPGA state with the lowest fragmentation is chosen. This algorithm is time consuming. As an interesting novelty, they consider non-rectangular tasks, with a hierarchical task model made of one or several rectangular sub-tasks, whose relative position can be modified ("footprint transform") during allocation. No clear results on the benefits of such transformation on the algorithm's performance are given, and in later works as [6] and [7] they substitute such task model by a rectangular one. In these papers they also propose an enhanced version of Barzagan's partitioner with the same efficiency but improved placement quality. This enhanced method delays the basic vertical/horizontal split decision and manages overlapping rectangles in a restricted form.

3 Our Approach

Our approach to RTR HW management will keep track of the available free area with a vertex-list structure, and will decide where the arriving tasks are mapped to by selecting one of the vertices of the list. Next we will explain our FPGA and task models, as well as the main features of the HW manager we propose, placing more emphasis on the vertex-list structure used to manage the available FPGA area and to make the task allocation decisions.

3.1 FPGA and Task Models

Our partially reconfigurable FPGA model is an homogeneous two dimensional grid formed by W*H basic RTR blocks, that we will use as "area units" all along. In our model we suppose that each block, made of a certain number of CLBs, includes all the interconnection resources needed for routing and data I/O. A task can be made of an arbitrary number of such RTR blocks. In order to simplify our algorithms, we suppose that tasks are always rectangular.

The tasks are relocatable and can be inserted at arbitrary positions with different row and column offsets. The tasks are independent, with no precedence constrains between them, but there can be real-time constrains that must be satisfied. Each task is defined by the following tuple of parameters:

$T_i = \{ w_i, h_i, t\_ex_i, t\_arr_i, t\_max_i \}$, where:

- w_i is the task width,
- h_i is the task height,
- t_ex_i is the task execution time,
- t_arr_i is the task arrival time,
- t_max_i is the maximum time for the task to finish execution. It must be satisfied that $t\_arr_i + t\_ex_i < t\_max_i$.

3.2 HW Manager Characteristics

Two main goals were set for our HW management algorithm: reduced execution time, that is, small overhead, and minimal FPGA fragmentation.

As fig. 1 shows, the HW manager is made of three main components: the Scheduler, the Allocator, and the Area Manager, and uses three important data structures:

- A Running Task List, L, where the information on the tasks currently running is stored.
- A Waiting Task Queue, Q where the arriving tasks are stored when there is not enough room for an immediate allocation.

- A Vertex List Set, VLS, that describes all the available FPGA free space. Each component of VLS is a Vertex List VL_i describing an independent fragment of the FPGA free space, or “hole”.

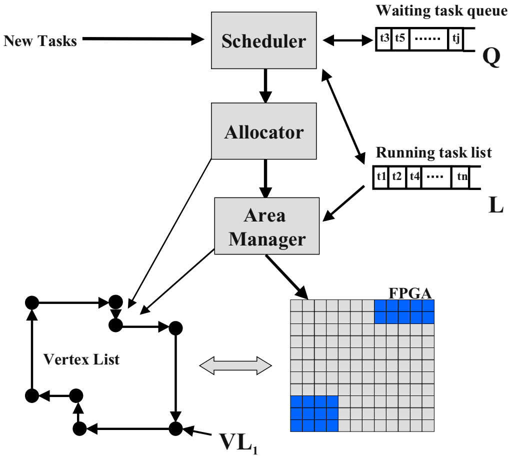


Fig. 1. HW manager structure

The HW manager operates as follows: When a new task arrives, the Scheduler picks it up and calls the Allocator to check whether a feasible position exists where the task could be mapped to. The Allocator consults the VLS to perform this checking.

When a feasible task insertion at a given candidate vertex is selected, the Allocator calls the Area Manager to update the FPGA free area description with the newly inserted task and also inserts this task in the list L. If the Allocator can not find a feasible insertion vertex in any VL_i, then the task goes to the waiting task queue, Q.

When the Scheduler detects that a task finishes execution, extracts this task from L and calls the Area Manager to update VLS. If the freed area is adjacent to an already existing hole, the corresponding VL_i is updated accordingly. If not, a new VL_j is created for the new hole. Section 3.3 will explain the possible situations.

Every time a task finishes execution and there is a free area increment, the Scheduler tries to insert as many tasks of Q as possible. The queue Q is sorted according to the timeout value of each task, computed as t_max_i - t_ex_i. Therefore when a task extraction happens, the Scheduler first tries to place the task which is closest to timeout. If timeout happens for a task (current time becomes greater that t_max_i-t_ex_i), the task should be rejected.

3.3 Vertex List Set Structure and Management

The problem of allocating the HW tasks inside the 2D FPGA is alike to the classical 2D bin-packing problem, consisting of packing rectangles inside a rectangular bin. This theoretical problem has been deeply studied and generic solutions with First Fit (FF) and Best Fit (BF) approaches can be found in [8].

As fig. 2 shows, the VLS structure is a geometrical description of the whole FPGA free area perimeter. This figure shows an example VLS with a single VL for the unique hole available. Some of the vertices are marked as valid candidates for task placement while others are not. During a task insertion process, the allocation algorithm travels clockwise each VL_i in VLS and looks for a feasible task insertion at all the vertex marked as candidates, until the VLS is finished. Each VL_i contains all the candidate vertex to locate the task inside hole i, with Bottom-Left (BL), Top-Right (TR), Bottom-Right (BR) or Top-Left (TL) approaches.

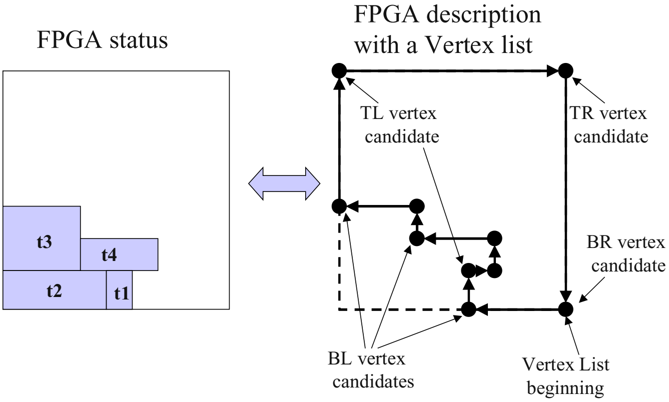


Fig. 2. Example FPGA status and associated vertex list

To perform the feasibility checking on each candidate vertex, the Allocator looks for intersections between the VL edges (formed by successive pairs of VL vertices) and the task edges. When the Allocator detects an edge intersection, it rejects the candidate vertex and continues searching the VL, looking for the next candidate.

The main aspects of the vertex list management are the task insertion and extraction processes.

For task insertion, the Allocator passes the candidate vertex and the task data to the Area Manager to update the hole shape by modifying the corresponding VL_i. The Area Manager shifts then some of the already existing vertices, and creates new vertices if necessary.

When a task finishes execution, the Scheduler extracts it from L and the VLS is updated. Several situations can arise then, depending on the number of holes the leaving task is adjacent to, that must be dealt with separately. Other specific problems can appear when merging several holes into a single one, or when integrating an isolated occupied fragment (and “island”) into the perimeter. A more detailed description of all the Vertex List Set management aspects can be found in [9].

4 Heuristics for Location Selection

Once our basic management algorithm and VLS structure have been explained, the problem of selecting a given vertex among all the feasible candidates to locate the task must be faced. To make this choice, a simple approach can be used based on a First-Fit criteria as it can be found in [9]. A more interesting alternative, though, is a Best-Fit approach that tries to find the best location according to a given criterion. We have developed two different heuristics based on adjacency and fragmentation criteria, that are shown next. With this new alternative, each VL_i is traveled completely and a value for each Candidate Vertex is computed according to the selected heuristic. Finally, the task is allocated in the most suitable Candidate Vertex.

4.1 Adjacency-Based Heuristic

This heuristic inserts the task on the vertex position where the arriving task achieves the higher contact level between the task edges and the envelope defined by the VL_i. This adjacency is computed in terms of RTR block length units.

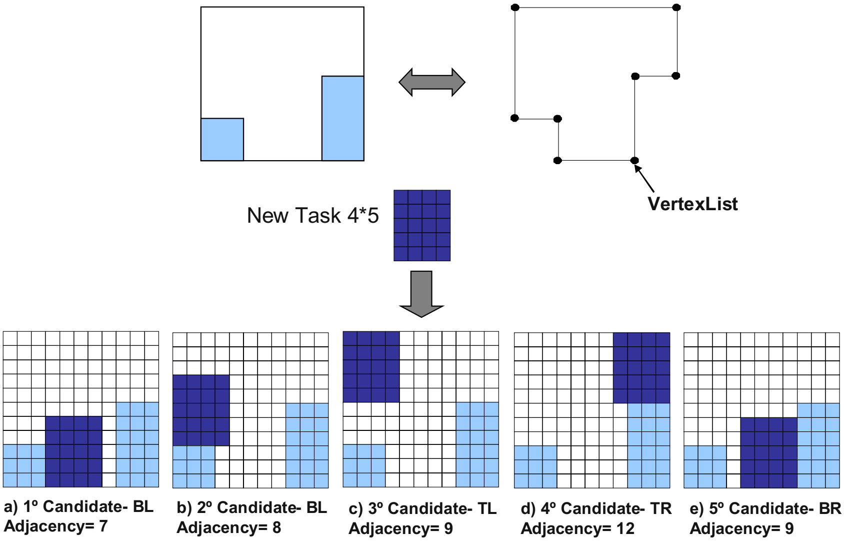


Fig. 3. Candidate locations in adjacency-based heuristic

Figure 3 shows a simple example to illustrate this heuristic. The FPGA status is shown on top, with two currently running tasks placed at the bottom and the corresponding VL. When a new task of 4*5 basic cells arrives, this approach computes the adjacency value for each feasible candidate position. Therefore a BF algorithm using the adjacency criteria would place the task at the fourth candidate.

4.2 Fragmentation-Based Heuristic

This heuristic estimates the fragmentation produced in the FPGA free area for each feasible candidate vertex, and finally inserts the task on the vertex position where a lower fragmentation level is produced. The fragmentation level is estimated with the following metric:

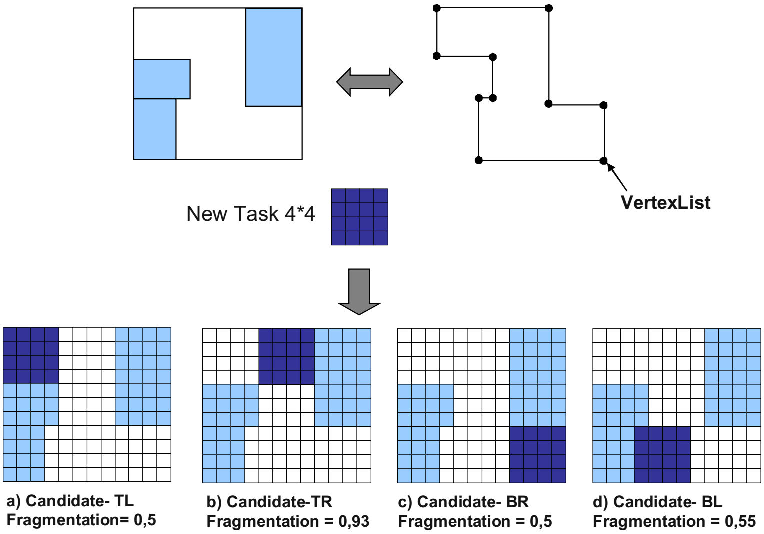
$$F = 1 - \Pi_i \ [(4/V_i) * (A_i/A_F)] \qquad (1)$$

Where the term between brackets represents a kind of "suitability" for a given hole i, with area A_i and V_i vertices:

- $(4/V_i)$ represents the suitability of the shape of the hole i to accommodate rectangular tasks. Notice that any hole with four vertices has the best suitability.
- (A_i/A_F) represents the relative hole normalized area. A_F stands for the whole free area in the FPGA. That is $A_F = \quad A_i$.

This fragmentation metric penalizes the proliferation of holes in the FPGA, as well as the task placements that generate holes with complex shapes and small sizes.

It is important to notice that the algorithm complexity estimation must cover both the task insertion/extraction method and the updating of the data structure that reflects the FPGA status. As a general rule, the simpler the task insertion method is, the more complex is the data structure update. Taking this into account, the global complexity of our algorithm is of $O(N^2)$, with N the number of running tasks in the FPGA.



a) Candidate- TL
Fragmentation= 0,5

b) Candidate-TR
Fragmentation = 0,93

c) Candidate- BR
Fragmentation = 0,5

d) Candidate- BL
Fragmentation = 0,55

Fig. 4. Candidate locations in fragmentation based heuristic

Figure 4 shows an example of this heuristic, with the FPGA status shown on the top. When a new task of 4*4 basic cells arrives to the FPGA, the fragmentation level is calculated for all the feasible locations. The A and C candidates produce the lowest fragmentation of the resultant free area, so one of them would be chosen by this heuristic for the task insertion.

5 Experimental Results

To evaluate the quality of our approaches, we have made experiments using four different algorithms for area management:

a. *Classical FF with BL heuristic(FF_BL).* When a new task arrives to the FPGA, it performs an exhaustive search, from left to right and from bottom to top, in order to find a feasible location for the arriving task.

b. *Vertex List with FF (VL_FF).* It uses the Vertex List structure presented in 3.3 and places the arriving task at the first feasible location.

c. *Vertex List with BF-Adjacency heuristic (BF_ADJ).*

d. *Vertex List with BF-Fragmentation heuristic (BF_FRAG).*

An example of how these four different algorithms process a new task by selecting different feasible locations is shown in figure 5. Additionally, the fragmentation metric is used to characterize the quality of the placements made by these algorithms. Notice that the classical FF shown in 5A can consider successful locations inside the free area perimeter, but not included in our corresponding VL. In this example, the BF with the fragmentation heuristic produces the lowest fragmentation level.

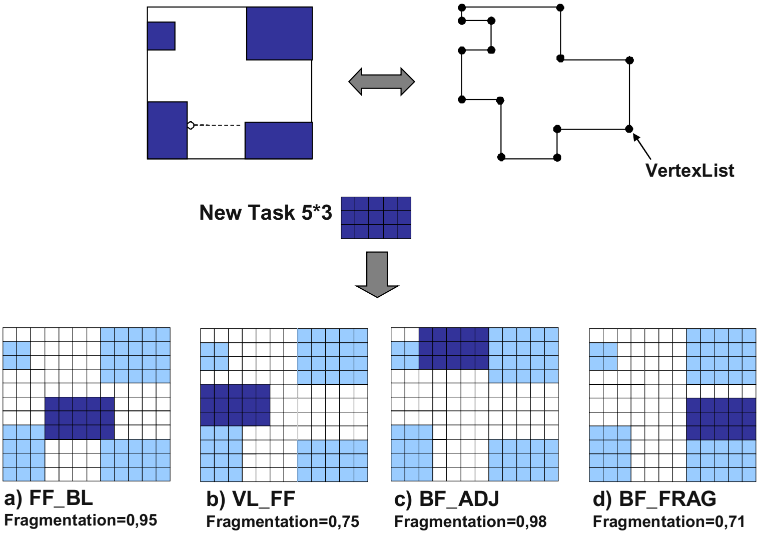


Fig. 5. Candidate locations for different management algorithms

These four algorithms, developed in C++, have been tested with an FPGA of 100*100 basic blocks, and different data sets of 100 tasks each that have been randomly generated with a task size range and ratio, similar to others found in many multitasking environments. These data sets have been classified in four classes, depending on the task size ranges, maximum waiting times, and arrival frequencies.

Table 1. Data set classes

Data Sets	Min. task area	Max. task area	Waiting time	Data Set features
A1, A2	5*5	55*55	1 to 5	High Critical - Medium size
B1, B2	10*10	60*60	1 to 10	Critical - Medium size
C1, C2	15*15	75*75	11 to 21	Low Critical Big - Med size
D1, D2	10*10	80*80	21 to 31	No Critical - Big size

We have used three different parameters to evaluate the results obtained. First, the computing "volume" rejected by the management algorithms for each data set. This volume represents all the tasks that were rejected because the manager was not able to find a proper location in time to meet the task time constraint. For each task, the volume is the product of the task area and its execution time. As figure 6 shows, the BF algorithms achieve better results for all the data sets, especially the version with fragmentation heuristic, that for most data sets is able to allocate all the tasks in time.

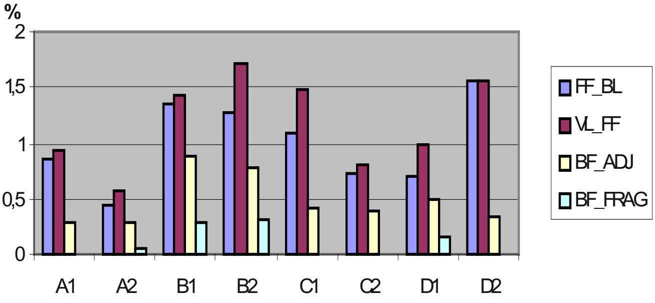


Fig. 6. Computation volume rejected (%)

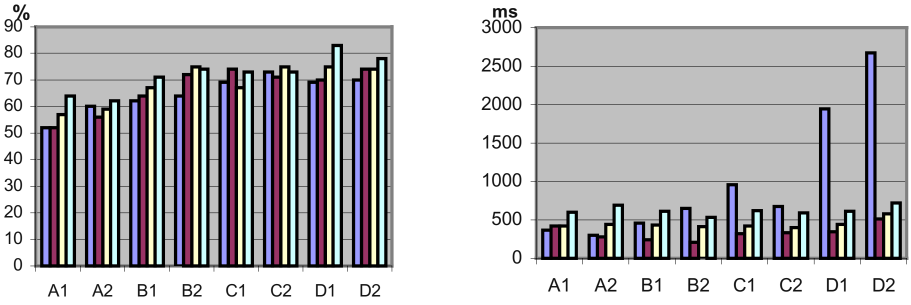


Fig.7. Average FPGA occupation level and algorithm execution times

The other parameters shown in fig 7 are the average FPGA occupation maintained for each set, and the time used by each algorithm to process the different data sets. As before, BF algorithm with fragmentation heuristic gives better results. It gets better FPGA occupation levels for most examples, at the cost of a minimal computing overhead, compared with the other algorithms. Thus, it becomes clear that it is worthy to use a fragmentation-based heuristic to decide where the tasks must be placed.

6 Conclusions

We have presented a new approach to HW multitasking, with an area manager that uses a novel approach to task insertion based on a vertex-list structure. Several heuristics for selecting task locations have been presented, and the heuristic based on a new fragmentation metric has clearly shown a better behavior.

As an adequate management of the fragmentation problem has revealed itself crucial, among our future work we are considering the incorporation of defragmentation strategies for exceptional situations, and of architectural modifications in order to speed up the defragmentation processes.

Acknowledgements. This work has been supported by Spanish Government research grant TIC2002-00160.

References

1. K. Compton, S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software”, ACM Computing Surveys, Vol. 34, No. 2, pp 171-210. June 2002.
2. O. F. Diessel, G. Wigley, "Opportunities for Operating Systems Research in Reconfigurable Computing", Technical report ACRC-99-018. Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.
3. O. F. Diessel, H. Elgindy, "On Dynamic Task Scheduling for FPGA-based Systems", International Journal of Foundations of Computer Science, IJFCS'01, Vol. 12, No. 5, 2001
4. K. Bazargan, R. Kastner, M. Sarrafzadeh, “Fast Template Placement for Reconfigurable Computing Systems”, IEEE Design and Test of Computers, Vol. 17, pp 68–83, 2000.
5. H. Walder, M. Platzner, “Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform”, ERSA’02, Las Vegas, US, pages 24-30, June 2002
6. H. Walder, C. Steiger, M. Platzner, “Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing”, RAW’03, Munich, Germany, April 2003
7. C. Steiger, H.Walder, M. Platzner , L. Thiele “Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices“ RTSS'03, Cancun, Mexico, Dec 2003
8. E.Coffman, J. Csirik, G. Woeginger, “Approximate Solutions to Bin-Packing Problems", in Handbook of Applied Optimization, P. Pardalos and M Resende, eds, Oxford University Press, 2002
9. J. Tabero J. Septién, H. Mecha, D. Mozos, “A vertex-list approach to 2D HW multitasking management in RTR FPGAs”, DCIS 2003, Ciudad Real, Spain, pages 545-550, Nov 2003

The Partition into Hypercontexts Problem for Hyperreconfigurable Architectures

Sebastian Lange and Martin Middendorf

Parallel Computing and Complex Systems Group
Department of Computer Science, University of Leipzig
Augustusplatz 10/11, D-04109 Leipzig, Germany
{langes,middendorf}@informatik.uni-leipzig.de

Abstract. Hyperreconfigurable architectures adapt their reconfiguration abilities during run time in order to achieve fast dynamic reconfiguration. Models for such architectures have been proposed that change their ability for reconfiguration during hyperreconfiguration steps and in ordinary reconfiguration steps reconfigure the actual contexts for a computation within the limits that have been set by the last hyperreconfiguration step. In this paper we study algorithmic aspects of how to optimally decide what hyperreconfiguration steps should be done during a computation in order to minimize the total time necessary for hyperreconfiguration and ordinary reconfiguration. It is shown that the general problem is NP-hard but fast polynomial time algorithms are given to solve this problem on different types of hyperreconfigurable architectures. These include newly introduced architectures that use a cache to store hypercontexts. We define an example hyperreconfigurable architecture and illustrate the introduced concepts for three application problems.

1 Introduction

The increasingly higher integration and flexibility of dynamically reconfigurable hardware lead to a large amount of information which has to be transferred onto the hardware for reconfiguration to define the new state of the system. This large amount of data transfer makes run time reconfigurations time critical operations, especially, for computations which exploit the full capacity of dynamically reconfigurable architectures by frequent reconfigurations. Different approaches have been proposed in the literature to cope with this problem, e.g., compression methods for the stream of reconfiguration bits ([4,6]), multi-context architectures [1,12]), self-reconfigurability ([8,15,17]) and hyperreconfiguration ([9]) which means that the reconfiguration potential of an architecture itself is reconfigurable.

In this paper we study algorithmic aspects of single task hyperreconfigurable architectures as they have been proposed in [9] (algorithmic aspects of multi-task hyperreconfigurable architectures are studied in [10]). Such architectures use two types of reconfiguration steps: i) reconfiguration steps where the reconfiguration potential of the architecture is defined ii) standard reconfiguration steps which are used to reconfigure the actual contexts which are used by the algorithm. The first type of reconfiguration steps are called hyperreconfiguration steps. Moreover, we extend hyperreconfigurable architectures by introducing a cache for storing hypercontexts.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 251–260, 2004.

A central problem that emerges on hyperreconfigurable architectures is to determine when hyperreconfiguration steps should be taken and how the reconfiguration potential should be defined in these steps in order to minimize the total time necessary for (hyper)reconfiguration of a computation. We call this problem Partition into Hypercontexts (PHC) problem and show that it is NP-hard. We also describe polynomial time algorithms for several variants of PHC on the so called Switch model of hyperreconfigurable architectures ([9]). Unfortunately, it is also shown that the introduction of a cache for hypercontexts makes the PHC problem NP-hard even for the Switch model. To illustrate the ideas in this paper we present an example for the PHC problem on the Switch model. An optimal solution for the PHC problem is provided when the example architecture has no cache and a heuristic solution when a cache for hypercontexts is used.

The paper is organized as follows. In the next Section 2 we describe hyperreconfigurable architectures and introduce the Partition into Hypercontexts (PHC) problem. In Section 4 we discuss polynomial time solvable cases of the PHC problem. A variant of the PHC problem with changeover costs is studied in Section 5. In section 6 we introduce hyperreconfigurable architectures with a cache for hypercontexts and study PHC for these architectures. Experimental results for a test architecture are presented in Section 7. The paper ends with a conclusion in Section 8.

2 The Partition into Hypercontexts Problem

Hyperreconfigurable architectures allow to alter the reconfiguration potential during run time and use two types of reconfiguration steps ([9]). The ordinary reconfiguration steps are used to actually define a new configuration of the system. The actual state of the system that can be changed by reconfiguration is called the context of a computation. Hyperreconfiguration steps are used for defining the actual reconfiguration potential of the architecture that is activated for reconfiguration in the ordinary reconfiguration steps. Thus, a hyperreconfiguration step defines the set of contexts that can potentially be reconfigured in (ordinary) reconfiguration steps. Such a set of possible contexts is called a hypercontext. A reconfiguration into a new context might be dependent on external and internal parameters of the computation and can be characterized by the set of all possible contexts that it defines depending on the data. Hence, a reconfiguration can in general only be executed during run time when the machine is in a hypercontext that contains this set of possible contexts. A set of possible contexts is called a context requirement and a hypercontexts that contains it *satisfies* the corresponding context requirement. It is assumed that a reconfiguration step requires reconfiguration information for all activated resources (even when the information is that an activated resource is not used in the corresponding context). Formal models for hyperreconfigurable architectures where the cost (e.g., the time or the amount of bits necessary to be loaded onto the architecture) of a reconfiguration step depends on the actual hypercontext have been given in [9] and are described in the following.

Let $\mathcal{C}$ be the set of possible context requirements for a reconfigurable machine and $C = c_1 \ldots c_m$, $c_i \in \mathcal{C}$ be the sequence of context requirements that characterizes an algorithm/computation. A *hypercontext* is a state of the machine which is characterized by the subset of $\mathcal{C}$ context requirements that are satisfied when the machine is in this state. At any time exactly one hypercontext is realized on the machine. Let $\mathcal{H}$ be the set of possible hypercontexts. For a hypercontext $h \in \mathcal{H}$ let $h(\mathcal{C}) \subset \mathcal{C}$ be the subset of context requirements that are satisfied by h. The set $h(\mathcal{C})$ is called the *context set* of h. For a sequence $c_1 \dots c_k$ of context requirements and a hypercontext h let $c_1 \dots c_k \subset h(\mathcal{C})$ denote the fact that for each context requirement c_i, $i \in [1:k]$ $c_i \in h(\mathcal{C})$ holds. In order to change the machine's current hypercontext a *hyperreconfiguration step* is necessary. For each hypercontext $h \in \mathcal{H}$ two cost measures are defined: i) $init(h)$ is the cost of performing a hyperreconfiguration that brings the machine into hypercontext h ii) $cost(h)$ denotes the cost of an ordinary reconfiguration step when the machine is in hypercontext h. Then a computation is characterized by a partition of C into substrings $S_1, \dots, S_r$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$, $r \geq 1$ such that $S_i \subset h_i(\mathcal{C})$ and $\sum_{i=1}^{r}(init(h_i)+cost(h_i)\cdot|S_i|)$ are the costs where $|S_i|$ is the length of S_i, i.e., the number of context requirements in S_i. When the algorithm/computation is executed the machine performs the following reconfiguration operations: $h_1 S_1 \dots h_r S_r$ where S_i stands for a sequence of $|S_i|$ reconfigurations which use only those parts of the machine which are available within the hypercontext h_i. It is assumed that a hyperreconfiguration is always performed before the first reconfiguration step.

An important problem that emerges for a hyperreconfigurable machine and a given algorithm (i.e. a sequence of context requirements) is to define when hyperreconfigurations are done and how corresponding hypercontexts are defined such that the context requirements of the algorithm are satisfied and the total costs for the hyperreconfiguration steps and the ordinary reconfiguration steps are minimized. Formally we define,

Partition into Hypercontexts (PHC) problem : Given a hyperreconfigurable machine (as described above) and a sequence $C = c_1 \dots c_m$ of context requirements. Find a partition of C into substrings $S_1, \dots, S_r$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$, $r \geq 1$ swith $S_i \subset h_i(\mathcal{C})$ and minimal total (hyper)reconfiguration.

Two variants of the model for hyperreconfigurable architectures have been introduced in [9]. The *DAG model* is for coarse grained reconfigurable machines where different reconfigurable submachines (hypercontexts) can be defined that can be ordered with respect to their computational power (this model is not considered in this paper due to space limitations). The second variant called *Switch model* is for fine grained machines where a set of small (similar) reconfigurable units (also called switches) exists. The reconfigurable machine that is available during a hypercontext is defined by the subset of available units. For reconfiguration the state of each available switch has to be defined. Thus the cost for reconfiguration is the number of available units plus some overhead cost. Formally, let $X = \{x_1, \dots, x_n\}$ be a set of switches and define $\mathcal{C} = \mathcal{H} = 2^X$, i.e., the set of possible context requirements $\mathcal{C}$ and the set of possible hypercontexts $\mathcal{H}$ equal the set of all subsets of X. For context $x \in X$ the relation $x \in h(\mathcal{C})$ holds, when $x \subset h$. Let $cost(h) = |h|$, where $|h|$ is the size of h, i.e., the number of switches available in h. Let $init(h) = n$ for $h \in \mathcal{H}$, which reflects the fact that for each switch it has to be defined during hyperreconfiguration whether it is available in the new hypercontext. A computation is characterized by a partition of C into substrings $S_1, \dots, S_r$, $r \geq 1$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are $r \cdot n + \sum_{i=1}^{r} |h_i| \cdot |S_i|$.

PHC-Switch problem: Given a hyperreconfigurable machine in the Switch model with the set of switches $X = \{x_1, \ldots, x_n\}$ and a sequence of context requirements $C = c_1 \ldots c_m$. Find a partition of C into substrings $S_1, \ldots, S_r$, $r \geq 1$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are minimal. Note that for the PHC-Switch problem there exist 2^n hypercontexts but this number is not part of the size of the problem instance which is $n + m$.

3 NP-Hardness

In this section we show that the general PHC problem is NP-hard which means it is unlikely that the problem can be solved in polynomial time.

Theorem 1. *The PHC problem is NP-complete.*

We only give the proof idea. For a proof one can encode an instance of an NP-hard problem, say 3-SAT, in a sequence of contexts C. Then a cost function and a set of hypercontexts can be defined such that there exists a cheap partition into hypercontexts of C if and only if the partition consists of a single hypercontext and the contexts in C encode an instance of 3-SAT that is solvable such that there exists no partition of C into substrings which can be covered by hypercontexts in a cheap way.

4 Polynomial Time Algorithm for PHC-Switch

In this section we describe a dynamic programming solution for the PHC-Switch problem. The algorithm computes a table $M = (M_{k,j})_{k \in [1:m], j \in [k:m]}$ where $M_{k,j}$ are the minimal costs for the prefix of length j of the sequence of context requirements $c_1 \ldots c_m$ when using k hypercontexts. The optimal solution for PHC-Switch can then be derived from this matrix. This algorithm is designed such that each row of the matrix can be determined in time $O(n \cdot m)$ so that the total run time is $O(n \cdot m^2)$.

In the following let h_{ij} be a cheapest hypercontext that satisfies the contexts requirements $c_i, \ldots, c_j$. First, we need some facts and definitions. It is not hard to show for each $k \in [1 : m]$: i) the value of $M_{k,p}$ is monotone decreasing in p, ii) for $j \in [k : m]$ the value of $cost(h_{i,j})$ is monotone decreasing in i. Let $j \in [k : m]$. It follows from the stated facts that there exists a partition $T_1, \ldots, T_h$ of the sequence of context requirements $c_k \ldots c_j$ such that $c_k \ldots c_j = T_1 \ldots T_h$ and for each string of contexts T_s, $s \in [1 : h]$ holds: For all contexts $c_t \in T_s$ the hypercontexts $h_{t,j}$ and therefor the costs $cost(h_{t,j})$ are the same. Recall, that $h_{t,j}$ for the PHC-Switch problem is defined as the hypercontext that consists of all switches that are element of at least one of the context requirements $c_t, \ldots, c_j$, i.e., $h_{t,j} = \bigcup_{i=t}^{j} c_i$. We call the partition $T_1, \ldots, T_h$ the *equal cost partition* of $[k : j]$. The corresponding intervals of indices of the contexts the *equal cost intervals*.

Let $[s : t]$ be an equal cost interval. For index $x \in [s : t]$ the values $\delta \in [1 : n]$ are determined for which $cM_{k,x-1} + \delta \cdot (t - (x - 1)) = \min\{M_{k,y-1} + \delta \cdot (t - (y - 1)) \mid y \in [s : t]\}$ holds. Clearly, for each index $x \in [s : t]$ the corresponding δ values form a subinterval of $[1 : n]$. This interval is called the *minimum cost interval of index*

x (within the equal cost interval $[s : t]$) and is denoted by I_x. It is not hard to show that $I_s, \ldots, I_t$ is a partition of $[1 : n]$ where all elements in I_i are smaller than all elements in I_{i+1} for $i \in [s : t-1]$.

In the following we describe the computation of a single matrix element in the main step of the algorithm. We assume that all elements in row 1 of $M_{k,j}$ and all elements $M_{k,k} = k \cdot w + \sum_{i=1}^{k} |c_i|$, $k \in [1 : m]$ have been computed during initialization. It is enough to consider the computation of an element $M_{k,j+1}$ for $k > 1$ and $j \in [1 : m-1]$ assuming that elements in row $k-1$ and element $M_{k,j}$ have already been computed.

In order to search efficiently for possible good places to introduce the kth hyperreconfiguration we introduce a pointer structure over parts of the sequence of context requirements $c_1 \ldots c_m$. First we describe the pointer structure over the sequence $c_k \ldots c_j$ for the computation of $M_{k,j}$ and then show how it can be extended to a pointer structure over the sequence $c_k \ldots c_{j+1}$ for the computation of $M_{k,j+1}$.

The first context requirements in each of the sequences of context requirements $T_h, \ldots, T_1$ are linked by so called *equal cost pointers*, i.e. there is a pointer to the first context requirement in T_h, from there to the first context requirement in T_{h-1} and so forth. Moreover, within each equal cost interval the indices x with a minimal cost interval that is empty or contains only values that are smaller than the actual costs $cost(h_{x,j+1})$ are linked in order of increasing value by so called *minimum cost pointers*. In addition, there is a pointer from the first context requirement of the interval to the last useful index in the interval. This pointer is called the *end pointer* of the equal cost interval. All indices with an equal cost interval that are linked by minimal cost pointers are called *useful*. All other indices are called *useless* and will be marked as useless by the algorithm. The following two facts which are not hard to show are used for run time analysis and to show the correctness of the algorithm (omitted due to space limitations).

Fact 1: It is easy to obtain from the equal cost partition $T_1, \ldots, T_h$ of $[k : j]$ and its corresponding pointers the equal cost partition $U_1 \ldots U_g$ of $c_k \ldots c_{j+1}$ of $[k : j+1]$ and the corresponding pointers in time $O(n)$.

To see that this is true observe that each string in $U_1, \ldots, U_g$ can be obtained by merging (or copying) neighbored strings from $T_1, \ldots, T_h$ and U_g contains in addition the context requirement c_{j+1}.

Fact 2: Consider an element T_s of the equal cost partition $T_1, \ldots, T_h$ of $[k : j]$. Let c_x (c_y) be the context in T_s (respectively from the element of the equal cost partition of $[k : j+1]$ that contains T_s) for which $M_{k,x-1} + cost(h_{x,j})$ (respectively $M_{k,y-1} + cost(h_{y,j+1})$) is minimal. Then it follows that $x \leq y$.

To compute $M_{k,j+1}$ the algorithm performs the following steps:

i) Extend the equal cost partition of $[k : j]$ by appending the (preliminary) equal cost interval c_{j+1} and let $[1 : n]$ be the (preliminary) minimal cost interval for $j+1$.

ii) Compute the equal cost partition of $[k : j+1]$ from the extended equal cost partition of $[k : j]$ by merging neighbored intervals when they have the same cost with respect to $j+1$.

iii) For each index within a merged interval the new equal cost interval is determined together with its minimal cost pointers and its end pointer. During this process all indices that have become useless are marked.

Clearly step (i) can be done in time $O(1)$. The determination of the intervals that have the same costs in step (ii) is done in time $O(n)$ by following pointers that connect the intervals. To determine the time for step (iii) consider an equal cost interval $[s_0 : s_h]$, $k \leq s_1 \leq s_h \leq j+1$ that was merged from $h \leq n$ old intervals $[s_0 : s_1], [s_1+1 : s_2], \ldots [s_{h-1}+1 : s_h]$. We now show that the computation of new pointers and the marking of useless indices takes time $O(h+q)$ where q is the number of marked indices.

a) For each of the h intervals consider the minimum cost interval of the index to which the first minimum cost pointer points. If the minimum cost interval does not contain a value that is at least as large as $cost(h_{s,j+1})$ then the index is marked as useless and the first pointer is merged with the next pointer. This process proceeds until every first minimum cost pointer points to a useful index.

b) Now it remains to update the minimum cost intervals by selecting for each cost value only the best index from the h merged intervals. This can be done in a left to right manner starting with the smaller cost values. Thereby always comparing the corresponding minimum cost intervals of indices between two neighbored of the h merged intervals, say $[s_{i-1}+1 : s_i]$ and $[s_i+1 : s_{i+1}]$, $i \in [1 : h-1]$. For ease of description we assume here that all values in one minimal cost interval are better than all values in the other interval. If this is not the case both minimum cost intervals are split so that each contains only the values for which it is better. Observe that the split value can be computed in constant time. When the minimum cost interval in the left interval $[s_{i-1}+1 : s_i]$ is better the corresponding index in the right interval is marked useless and the next minimum cost intervals are compared. When the minimum cost interval in the right interval $[s_i+1 : s_{i+1}]$ is better the index in the left interval is marked useless. Then the minimum cost interval in the old right interval (now the new left interval) is compared with the corresponding minimum cost interval of its right neighbor interval $[s_{i+1}+1 : s_{i+2}]$. During the search for the corresponding minimum cost interval all indices that are passed are marked useless. The process stops when the best minimum cost interval with value n is found. During the search a pointer is set from the rightmost useful index of an interval to the first useful index in its right neighbor. Thereby it might be necessary to jump over intervals that have no useful index left. The end pointer of the first interval is set to point to the last useful index of the merged intervals.

Since the total number of intervals in the equal cost partition for $[k : j+1]$ is at most n minus the number of merged intervals the time to compute $M_{k,j+1}$ is at most $O(n+q)$ where q is the total number of indices that are marked useless. Since at most $m-k$ indices exist in row k of matrix M it follows that the computation sum of all steps (iii) for computing the elements in this row is $O(n \cdot m + m)$.

Theorem 2. *The PHC-Switch problem can be solved in time $O(n \cdot m^2)$.*

5 PHC with Changeover Costs

In this section we study a variant of the PHC problem where the cost for a hyperreconfiguration depends not only on the new hypercontext but also on its preceding hypercontext. Parts of the hyperreconfiguration costs can then be considered as changeover costs and therefore we call this problem the PHC problem with changeover costs. This problem is used to model architectures where during hyperreconfiguration it is not necessary to specify the new hypercontext from scratch but where it is possible to define the new hypercontext through its difference to the old hypercontext. In the following we consider the problem only for the Switch-Model. For this problem the changeover costs between two hypercontexts are defined as the number of switches for which the state has to be changed for the new hypercontext (i.e., the state is changed from available to not available or vice versa). Formally, the problem can be stated as follows.

PHC-Switch problem with changeover costs: Given an instance of the PHC-Switch problem, where $init(h) = w$ for $h \in \mathcal{H}$, $w > 0$, the cost function $changeover$ on $\mathcal{H} \times \mathcal{H}$ is defined by $changeover(h_1, h_2) := |h_1 \triangle h_2|$ where $\triangle$ denotes the symmetric difference, and an initial hypercontext $h_0 \in \mathcal{H}$. Find a partition of C into substrings $S_1, \ldots, S_r$, $r \geq 1$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and $r \cdot w + \sum_{i=1}^{r}(|h_i \triangle h_{i+1}| + |h_i| \cdot |S_i|)$ is minimized.

The next result shows that PHC-Switch with changeover costs is polynomially solvable (the algorithm is too involved for the available space and omitted).

Theorem 3. *The PHC-Switch problem with changeover costs can be solved in time* $O(m^4 \cdot n)$.

6 Caches for Hypercontext and PHC

Multi-context devices allow to store the reconfiguration data that are necessary to specify a set of contexts. Such context caching on the device can lead to a significant speedup compared to single context devices where the reconfiguration bits have to be loaded onto the device from a host computer for every reconfiguration. In this section we introduce multi-hypercontext hyperreconfigurable architectures, which have a cache for hypercontexts so that they can switch between hypercontexts very rapidly. The concept of reconfigurable devices with context switching has been introduced a decade ago (e.g. the dynamically configurable gate array (DPGA) [1] or WASMII [12]). In [14] the reconfigurable computing module board (RCM) has been investigated which contains two context-switching FPGAs, called CSRC, where the context switching device can store four contexts.

A typical cache problem for many reconfigurable architectures is that the sequence of contexts for a computation is known in advance and the problem is then to find the best replacement strategies for the contexts that are stored in the cache. On a run time reconfigurable machine the problem is that the actual contexts might not be known in advance because they can depend on the actual results of a computation. But what might be known in advance are general requirements on the contexts, e.g. whether few or many routing resources are needed. The actual context, e.g. the exact routing, is then defined at a reconfiguration step. Therefore, it seems a promising concept for hyperreconfigurable architectures to introduce a cache for storing hypercontexts.

What makes the problem of using a cache for hypercontexts particularly interesting on a hyperreconfigurable machine is that different sequences of hypercontexts are possible which can satisfy the sequence of context requirements of a computation. Hence, the algorithm that computes the best sequence of hypercontexts should take the use of the cache into account. In general, it can be advantageous to use fewer but more comprehensive hypercontexts in order to increase the chances that a hypercontext which is to be used already exists in the cache and can therefore be loaded very fast. Thus, there is a trade-off between the increasing reconfiguration costs when fewer but more comprehensive hypercontexts are used and the shrinking costs for loading these hyperreconfigurations.

Here we consider a hyperreconfigurable machine with a cache for hypercontexts that can store a fixed maximal number of hypercontexts. It is assumed that a hypercontext has to be loaded from the host only when the hypercontext is not stored in the cache. Hence, the cost for loading a hypercontext h depends on whether it is in the cache or not. The value of $init(h)$ is smaller when the hypercontext is in the cache. For a machine with cache we define the PHC-Switch problem as follows.

PHC-Switch problem (for hyperreconfigurable machines with a cache for hypercontexts): Given a cache capacity $2n$, a set of switches $X = \{x_1, \ldots, x_n\}$, a set of context requirements $\mathcal{C}$ and a set of hypercontexts $\mathcal{H}$ defined as $\mathcal{C} = \mathcal{H} = 2^X$,i.e., $\mathcal{C}$ and $\mathcal{H}$ equal the set of all subsets of X. For a given sequence of context requirements $C = c_1 \ldots c_m$ find a partition of C into substrings $S_1, \ldots, S_r$, $r \geq 1$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and $r_1 \cdot n + r_2 \cdot c + \sum_{i=1}^{r} |h_i| \cdot |S_i|$ is minimized where r_2 is the number of hypercontexts that can be loaded from the cache, $r_2 := r - r_1$, and c the cost to load a hypercontext from the cache.

We can show the following theorem by a reduction from 3-SAT (the proof is somewhat technical and therefore omitted).

Theorem 4. *The PHC-Switch problem is NP-hard on a hyperreconfigurable machine with a cache for hypercontexts.*

7 Experiments and Results

We define a Simple HYperReconfigurable Architecture (SHyRA) as an example of a minimalistic model of a rapidly reconfiguring machine in order to illustrate our concepts. As depicted in Figure 1 it features 18 reconfigurable Look-Up Tables each with three inputs and one output. For storing signals a file of 73 registers is used. The registers are reconfigurably connected to the LUTs by a 73:54 multiplexer and 18:73 demultiplexer. The inability of the architecture to directly chain the LUTs for computation poses a bottle neck for the test applications we run on SHyRA and forces them to make extensive use of reconfigurations. The test applications therefore naturally lend themselves to profit from the use of hyperreconfigurations. This, however, does not limit the general validity of the experimental results, because although SHyRA implicitly imposes reconfiguration every reconfigurable application follows the same basic design, i.e. having a calculation phase (LUTs), transferring the information to some registers (DeMUX) and then have it reinjected into the next calculation phase (MUX). In order to evaluate the caching model, each reconfigurable component was equipped with a cache of up to 14 cache lines. Two sample applications (a 4 bit adder and a primitive ALU) were mapped to the modified SHyRA.

After mapping the design onto the reconfigurable resources (LUT contents, MUX switching information) a heuristic was employed to determine appropriate hypercontexts

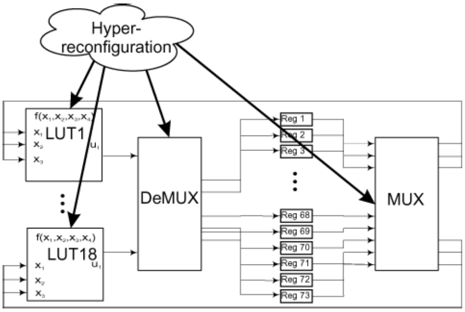


Fig. 1. Simple HYperReconfigurable Architecture: Principal System Design

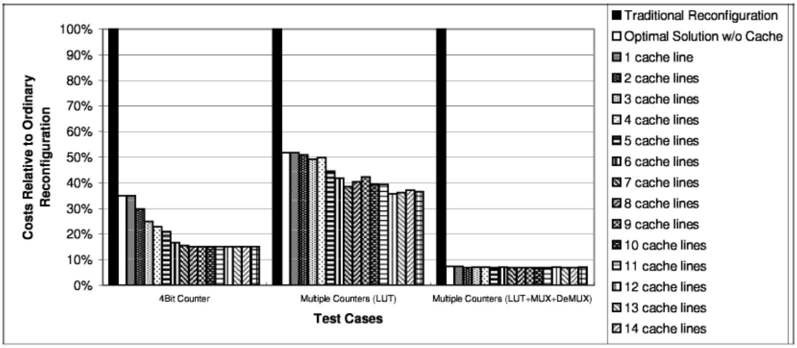


Fig. 2. Relative Costs of the Test Case Designs With Cache Size From 1 to 14 Lines

using the same costs as in the Switch model. For the case of not using caches the optimal hypercontexts were determined with the algorithm described in Section 4. For the case with caches for hypercontexts we used a greedy strategy which takes the optimal solution for the PHC-Switch problem without caches as starting point and subsequently improves this solution by randomly applying one of three operations:

1. Two randomly chosen hypercontexts are merged. 2. Two hypercontexts are chosen randomly. For each context c_j a penalty cost $(cost(c_j) = \sum_{k\in[0,n],c_{jk}=1}(|\{c_i|i \neq j, c_{ik} = 0\}|))$ is determined and the most expensive context is exchanged (this is repeated as long as the total costs become smaller). 3. One randomly chosen hypercontext is split into two hypercontexts and the same exchange procedure as in (2) is applied.

Figure 2 shows the resulting total hyperreconfiguration costs for the test designs without cache and with caches of sizes from one two 14 cache lines. For the test applications it can be observed that small caches for hypercontexts can significantly decrease the total hyperreconfiguration costs.

8 Conclusion

We have investigated a central algorithmic problem for hyperreconfigurable architectures, namely the Partition into Hypercontexts (PHC) problem. It was shown that the problem in NP-hard in general but can be solved in polynomial time for the Switch model under different cost measures. We have also introduced hyperreconfigurable architectures that use a cache to store hypercontexts and have shown that PHC becomes NP-hard even for the Switch model for this architectures. Applications of the PHC problem on an example architecture have been given. For the case when caches for hypercontexts are used a heuristic for solving the PHC problem was introduced.

References

1. M. Bolotski, A. DeHon, and Jr. T.F. Knight: Unifying FPGAs and SIMD Arrays. Proc. FPGA '94 – 2nd International ACM/SIGDA Workshop on FPGAs, 1-10, (1994).
2. K. Bondalapati, V.K. Prasanna: Reconfigurable Computing: Architectures, Models and Algorithms. In Proc. Reconfigurable Architectures Workshop, IPPS, (1997).
3. K. Compton, S. Hauck: Configurable Computing: A Survey of Systems and Software. ACM Computing Surveys, 34(2): 171–210, (2002).
4. A. Dandalis and V. K. Prasanna: Configuration Compression for FPGA-based Embedded Systems. In Proc. ACM Int. Symposium on Field-Programmable Gate Arrays, 173–182, (2001).
5. C. Haubelt, J. Teich, K. Richter, and R. Ernst: System Design for Flexibility. In Proc. 2002 Design, Automation and Test in Europe, 854–861, (2002).
6. S. Hauck, Z. Li, and J.D.P. Rolim: Configuration Compression for the Xilinx XC6200 FPGA. IEEE Trans. on CAD of Integrated Circuits and Systems, 8:1107–1113, (1999).
7. P. Kannan, S. Balachandran, D. Bhatia: On Metrics for Comparing Routability Estimation Methods for FPGAs. In Proc. 39th Design Automation Conference, 70–75, (2002).
8. M. Koester and J. Teich: (Self-)reconfigurable Finite State Machines: Theory and Implementation. In Proc. 2002 Design, Automation and Test in Europe, 559–566, (2002).
9. S. Lange and M. Middendorf: Hyperreconfigurable Architectures for Fast Runtime Reconfiguration. To appear in Proceedings of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM´04), Napa Valley, USA, 2004.
10. S. Lange and M. Middendorf: Models and Reconfiguration Problems for Multi Task Hyperreconfigurable Architectures. To appear in Proc. RAW 2004, Santa Fe, 2004.
11. K.K. Lee and D.F. Wong: Incremental Reconfiguration of Multi-FPGA Systems. In Proc. Tenth ACM International Symposium on Field Programmable Gate Arrays, 206–213 , (2002).
12. X. P. Ling, and H. Amano: WASMII: a Data Driven Computer on a Virtual Hardware. Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, 33-42, (1993).
13. T.-M. Lee, and J. Henkel, W. Wolf: Dynamic Runtime Re-Scheduling Allowing Multiple Implementations of a Task for Platform-Based Designs. In Proc. 2002 Design, Automation and Test in Europe, 296–301, (2002).
14. K. Puttegowda, D.I. Lehn, J.H. Park, P. Athanas, and M. Jones: Context Switching in a Run-Time Reconfigurable System. The Journal of Supercomputing, 26(3): 239-257,(2003).
15. R.P.S. Sidhu, S. Wadhwa, A. Mei, V.K. Prasanna: A Self-Reconfigurable Gate Array Architecture. Proc. FPL (2000) 106-120.
16. M. Teich, S. Fekete, and J. Schepers: Compile-Time Optimization of Dynamic Hardware Reconfigurations. Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, U.S.A., 1999.
17. S. Wadhwa, A. Dandalis: Efficient Self-Reconfigurable Implementations Using On-chip Memory. Proc. FPL, (2000) 443-448.

A High-Density Optically Reconfigurable Gate Array Using Dynamic Method

Minoru Watanabe and Fuminori Kobayashi

Department of Systems Innovation and Informatics,
Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka, 820-8502, Japan
Tel: +81-948-29-7749, Fax: +81-948-29-7749
{watanabe, koba}@ces.kyutech.ac.jp

Abstract. A high-density optically reconfigurable gate array (ORGA) is proposed to improve the gate density of conventional ORGAs, which are a type of Field Programmable Gate Array (FPGA). However, unlike FPGAs, an ORGA is reconfigured optically with external optical memories. A conventional ORGA has many programming elements, just as FPGAs do. One programming element consists of: a photodiode to detect an optical reconfiguration signal; a latch, a flip-flop or a bit of memory to temporarily store the reconfiguration bit; and some transistors. Among those components, the latch, flip-flop, or memory occupies a large implementation area on a typical VLSI chip; it prevents realization of a high-gate-density ORGA. This paper presents a high-density ORGA structure that eliminates latches, flip-flops, and memory using a dynamic method and a design of an ORGA-VLSI chip with four optically reconfigurable logic blocks, five optically reconfigurable switching matrices, and four optical reconfigurable I/O blocks including four I/O bits. It uses 0.35 μm 3-Metal CMOS process technology. This study also includes some experimental results.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have been used widely in recent years because of their flexible reconfiguration capabilities. Moreover, demand for high-speed reconfigurable devices has been increasing. If circuit information can be exchanged rapidly between the gate array and memory, an idle circuit can be evacuated into memory; other necessary circuits at that time can be downloaded from memory into the gate array, thereby increasing the activity of the gate array.

Nevertheless, it is difficult to realize a rapidly reconfigurable device using a structure in which a gate array VLSI and a memory are separated into different chips, with both of them connected by metal wiring, as in an FPGA [1][2]. Presuming that a reconfigurable device functions at 100 MHz, the total number of reconfiguration bits is 100,000. The device and the external memory are connected by single wiring, requiring a transmission rate of 10 Tbps. The transmission rate can not be realized using standard CMOS processes. Although a wide range of reconfiguration wiring should be used, nevertheless, the problem remains because available packages and bonding wiring are limited to several thousand pins. For those reasons, electrical reconfiguration bandwidth is not sufficiently large to accommodate the number of configuration bits.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 261–269, 2004.

On the other hand, high-speed reconfigurable processors have been developed, e.g. DAP/DNA chips and DRP chips [3][4]. They package reconfiguration memories and microprocessor array onto a chip. The internal reconfiguration memory stores reconfiguration contexts of 3 to 16 banks and the banks can be changed from one to the other on a clock. This process constitutes the so-called context-switching method. Thereby, the ALU of such devices can be reconfigured on every clock cycle in a few nanoseconds. However, increasing the internal reconfiguration memory while maintaining gate density is extremely difficult.

Recently, new devices that combine various optical and electrical techniques have been developed to supplement weak points [5]-[8]. In such devices, optically reconfigurable (programmable) gate arrays (ORGAs) [9]-[12] and optically differential reconfigurable gate arrays (ODRGAs) [13][14] have been developed to hasten the reconfiguration time of conventional FPGAs. Such devices are similar to FPGAs, but they can be reconfigured optically with external optical memories.

For optical reconfiguration, these devices require a photodiode to detect an optical reconfiguration signal, a latch, a flip-flop, or memory that temporarily stores the reconfiguration signal, and transistors. Conventional ORGAs and ODRGAs can be considered to have added photocircuits onto the gate array of FPGAs. However, the implementation area of the photocircuits of ODRGA reaches 47% of the implementation area of VLSI chip when the photodiode size is 25 um^2 and the gate count is 65 k. The large implementation area of the photocircuits prevents realization of a high-gate-density ORGA.

Therefore, this study presents a high-density ORGA structure eliminating latches, flip-flops, or memory using dynamic method and a design of an ORGA-VLSI chip with four optically reconfigurable logic blocks, five optically reconfigurable switching matrices, and four optical reconfigurable I/O blocks including four I/O bits using 0.35 μm 3-Metal CMOS process technology. Some experimental results are also explained.

2 ORGA with Dynamic Method

2.1 Dynamic Reconfiguration Circuit

Conventional ORGAs and ODRGAs consist of a VLSI part and an optical part that allows optical reconfiguration. In the optical part, holographic memories or spatial light modulators are used as memory. The reconfiguration contexts are stored in it and are read out by laser. In contrast, the VLSI part consists of gate array similar to that of FPGAs and a reconfiguration circuit including many programming elements that need photodiodes to detect the optical reconfiguration context, latches, flip-flops, or memory to temporarily store the reconfiguration context, and some transistors. However, it is considered that conventional ORGAs and ODRGAs have an excess function that memory functions exist on both sides of the optical part and the VLSI part. The memory function of VLSI part has an important function: keeping the states of gate array while refreshing photodiodes. However, if the memory function of VLSI part were eliminated, the gate density could be extremely large.

Therefore, this paper proposes a high-density ORGA that eliminates the static memory function of the VLSI part and uses dynamic method. The gate array information is stored in photodiodes instead of latches, flip-flops, or memory. An array of dynamic

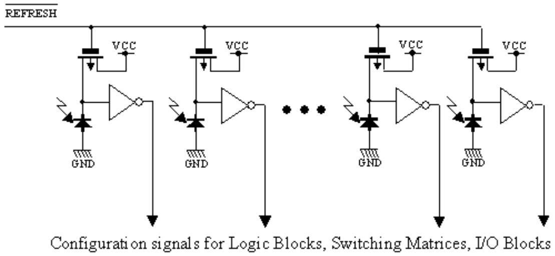

REFRESH
VCC
GND
Configuration signals for Logic Blocks, Switching Matrices, I/O Blocks

Fig. 1. Schematic diagram of an array of dynamic reconfiguration circuits eliminating a static memory function.

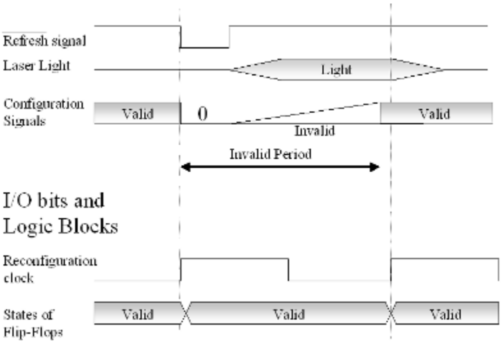


Fig. 2. Timing diagram of a reconfiguration cycle.

reconfiguration circuits eliminating the static memory function and a timing diagram of it are shown in Figs. 1 and 2, respectively. The reconfiguration circuit comprises refresh transistors, photodiodes, and inverters. Photodiodes not only detect light, but also serve as dynamic memory. The photodiode states are connected through inverters to the gate array portion.

The reconfiguration procedure of the reconfiguration circuit is initiated by activating the refresh signal to charge the junction capacitance of photodiodes. After charging is completed, a laser irradiates the photodiodes. If laser light penetrates the photodiode, the junction capacitance of the photodiode is discharged rapidly; if not, the voltage charged in the junction capacitance of photodiode is retained. The refresh cycle described above is completed instantaneously. The gate array states can then be maintained for a certain time before the junction capacitance is completely discharged by leak current. In the case of using high-density ORGAs, it is natural that the gate array is reconfigured dynamically so that the above reconfiguration cycle arises automatically. Of course, in the case where a reconfiguration cycle does not arise while the electric charge of the junction capacitance is maintained, the gate array must be refreshed with an identical reconfiguration context before the junction capacitance completely discharges. However, since the period between refreshes is typically a few dozen milliseconds - which is a relatively long time, this is not a common problem and such refreshing never affects the use of high-density ORGAs.

Table 1. Specification of a designed high-density ORGA with 4 Logic blocks, 5 switching matrices, and 16 I/O bits.

Technology	0.35 μm double-poly triple-metal CMOS process
Chip size	4.9 × 4.9 [mm]
Supply Voltage	Core 3.3V, I/O 3.3V
Photodiode size	25.5 × 25.5 [μm]
Distance between Photodiodes	h.=99 v.= 99 [μm]
Number of Photodiodes	605
Number of Logic Blocks	4
Number of Switching Matrices	5
Number of I/O bits	16

When using the high-density ORGA, each I/O bit with the role of an output must have a flip-flop to retain the output of the gate array while refreshing or reconfiguring, however, that has already implemented on almost all of currently available FPGAs. Also, all flip-flops included in I/O bits and logic blocks are synchronized with reconfiguration cycles to retain the output and internal states of the gate array while refreshing or reconfiguring. The timing diagram of reconfiguration clock and the outputs of flip-flops are shown in the lower portion of Fig. 2. As synchronized, internal states are kept in flip-flops of logic block and I/O bits while refreshing or reconfiguring, invalid states are blocked, and only valid states are outputted.

Using the dynamic method, the total number of flip-flops, latches, or memory bits on gate array is dramatically decreased compared to that of conventional ORGAs and ODRGAs.

2.2 VLSI Design

A high-density ORGA-VLSI chip was designed using 0.35 μm CMOS standard process, as shown in Table 1. The voltages of core and I/O cells were designed identically with 3.3 V. Photodiode and photodiode cell sizes including a refresh transistor and an inverter are 25.5 $\mu m\times$ 25.5 μm and 40.5 $\mu m\times$ 33.0 μm, respectively. The photodiodes were constructed between the N-well and P-substrate. The photodiode cells are arranged at 99 μm intervals in two dimensions. The total number of photodiodes is 605. The CAD layout of a photodiode cell is shown in Fig. 3. The implementation area of a reconfiguration circuit using dynamic method is reduced to 74.25 μm^2 compared with 618.75 $[\mu m^2]$ of the reconfiguration circuit of ODRGAs. The implementation area I of the reconfiguration circuit is determined as follows:

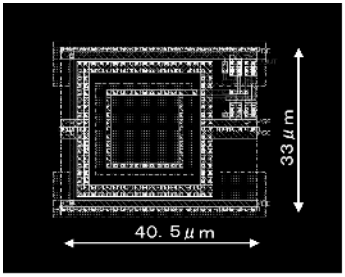


Fig. 3. CAD Layout of a photodiode cell.

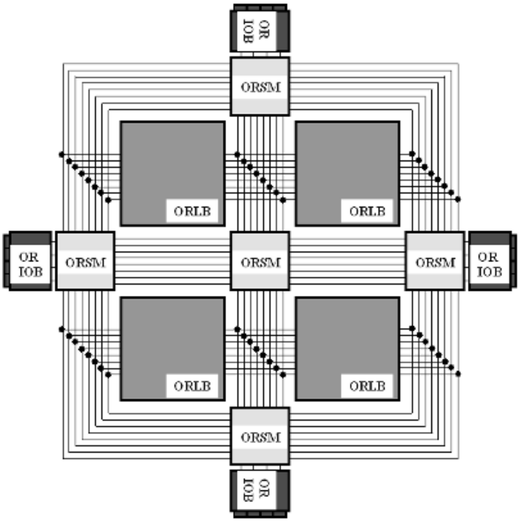


Fig. 4. Block diagram of high-density ORGA.

$$I = (P + R) \times N \ , \tag{1}$$

where P and R denote the implementation areas of photodiodes and other circuit components including flip-flops, latches, inverters, and other transistors; N is the number of programming elements. In this design, although photodiodes were designed for large size to allow easy justification, the positioning between a VLSI part and an optical part, in near future, the size will be less than 25 μm^2. At that time, because the relation P<<R exists, reducing R becomes important. The circuit reconfigured with dynamic method has a reduced implementation area: it can be less than 1/8 the conventional size. This size is very useful to increase gate density.

Next, the block diagram and CAD layout of high-density ORGA-VLSI chip are shown in Fig. 4 and Fig. 5, respectively. A high-density ORGA-VLSI chip was designed using the ROHM 0.35μm standard cell library, except for photodiode cells and transmission gate cells. For design, the Synopsys Design Compiler and Apollo were used as

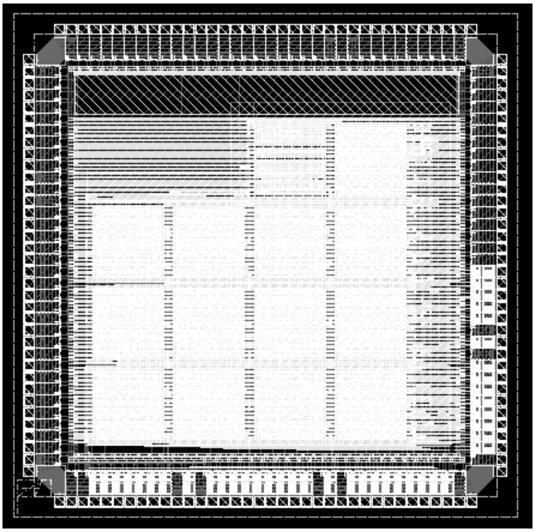


Fig. 5. CAD Layout of ORGA using 0.35 μm CMOS standard process.

the logic synthesis tool and the place and route tool, respectively. The top metal layer was used for guarding transistors from light irradiation; the other two layers were used for wiring.

The gate array of high-density ORGA-VLSI chip consists of 4 Optically Reconfigurable Logic Blocks (ORLBs), 5 4-direction Optically Reconfigurable Switching Matrices (4-ORSMs), and Optical Reconfigurable I/O Blocks including four I/O bits (ORIOBs). All routing channels include the same eight wires connected by 4-ORSMs. Gate array functionality is fundamentally identical to conventional FPGAs. The basic

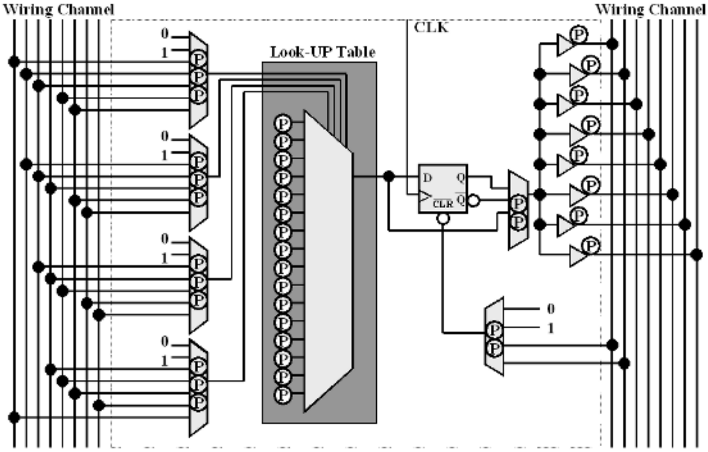


Fig. 6. Block diagram of an optically reconfigurable logic block (ORLB).

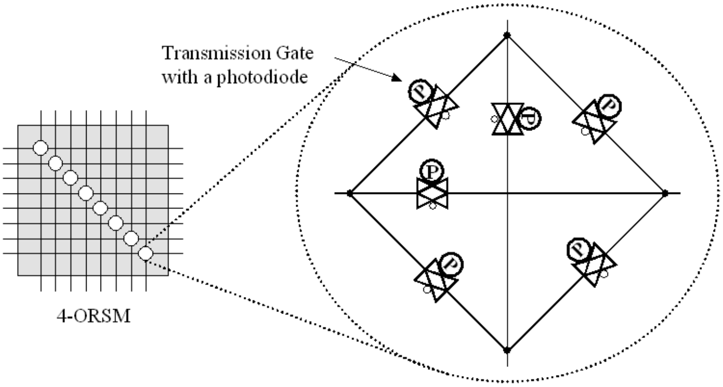


Fig. 7. Block diagram of a 4-direction optically reconfigurable switching matrix (4-ORSM).

building units of ORLBs is shown in Fig. 6. The ORLB consists of a 4-input Look-Up Table (LUT), six multiplexers and a D-Flip Flop (D-FF), and eight tri-state buffers. The LUT has 16 photodiodes - all the states of which can be programmed optically. Each bit of input of LUT is connected through a 7:1 multiplexer into the wiring channel. The multiplexer is connected to a logic 0 and a logic 1 in addition to five inputs from the wiring channel, the state of which can be optically programmed by three photodiodes. The inputs of logic 0 and logic 1 are used for converting the optical signal to an electrical signal. Keeping two photodiodes low, the state of other photodiode decides the multiplexer output state. Consequently, the inputs of LUT can be chosen to be either an optical signal or an electrical signal. The output of LUT is connected to a multiplexer and a D-FF. The outputs of LUT and $Q, \bar{Q}$ of D-FF are selected by the multiplexer. The output signal of the multiplexer is connected through tri-state buffers into wiring channel. Consequently, ORLB functionality is fundamentally identical to conventional FPGAs except for treating optical signals.

The block diagram of 4-ORSM is shown in Fig. 7. Although the structure of the ORSMs is basically same as those sold by Xilinx Inc., each transmission gate has an added photodiode and can be reconfigured optically. The transmission gate size of ORSMs is 99 μm^2.

3 Experimental System and Results

A dedicated high-density ORGA-subboard was developed as shown at the left side of Fig. 8 to evaluate the performance of a high-density ORGA-VLSI chip. The board that was used in experimentation was a combination of an FPGA board on the market, to which was implemented a USB2 interface, EP20KC200CF484C8 (Altera Corp.), a power supply circuit, and so on. The boards generate control signals for a high-density ORGA-VLSI chip. In addition, the boards are controlled using a personal computer (PC) through a USB2 interface. Mainly, communication between the PC and boards is required while fitting the position between an optical illumination system and boards with a high-density ORGA-VLSI chip.

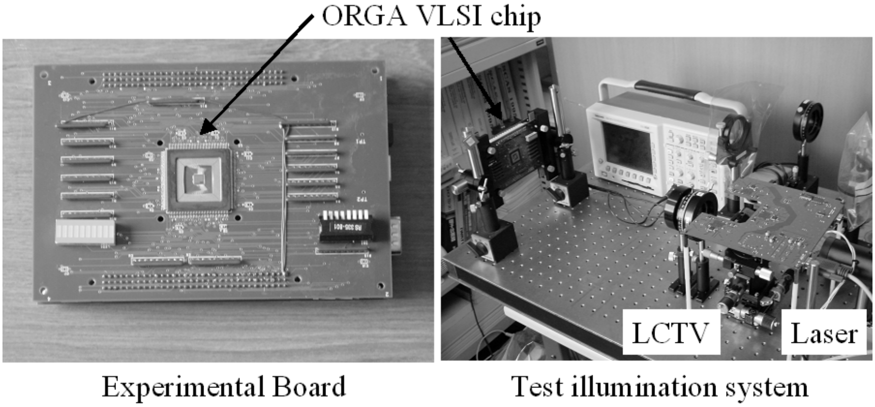


Fig. 8. Experimental board with ORGA chip.

Table 2. Results of reconfiguration circuit characteristics.

Reconfiguration circuit characteristics	
Reconfiguration Time	20.8 [ns]
Data Retention Time	93 [ms]

An optical illumination system with spatial light modulators for an ORGA is shown at the right side of Fig. 8. The system consists of a Liquid Crystal Television Panel-Spatial Light Modulator (LCTV-SLM), a 20 mW He-Ne-laser, a beam-expander, a lens, and polarizers. The LCTV-SLM resolution is 17 $\mu m\times$ 17 μm. A lens at the front of ODRGA-VLSI chip is employed for production and scale-down of the image on LCTV-SLM.

Using the FPGA boards and the optical illumination system, reconfiguration circuit characteristics were measured as shown in Table 2. The data retention time keeping the output state of an inverter connected to a photodiode shows the high-density ORGA refresh period. The refresh period of designed high-density ORGA is satisfied by larger than 93 ms. Moreover, using 20 mW He-Ne Laser, the reconfiguration time was confirmed as less than 20.8 ns. Results show that a high-density ORGA can be used with a low refresh rate and dynamic RAM; its reconfiguration speed is 1,000,000 times faster than that of FPGAs.

4 Conclusion

Conventional ORGAs and ODRGAs have an excess function: memory functions exist on both an optical and VLSI portions. This paper has proposed a high-density ORGA to improve gate density of conventional ORGAs and ODRGAs. It uses a method by which its gate array information is stored in photodiodes instead of latches, flip-flops, or memory in the VLSI portion. A structure using dynamic method could remove excess latches, flip-flops, or memory so that the implementation area of a reconfiguration circuit without photodiodes could be reduced to less than 1/8 its usual size.

A design of an ORGA-VLSI chip with 4 logic blocks, 5 switching matrices, and 4 optical reconfigurable I/O blocks including four I/O bits using 0.35 μm 3-Metal CMOS technology was presented. In addition, the refresh period and reconfiguration time were confirmed, experimentally, to be larger than 93 ms and to be less than 20.8 ns, respectively. Results demonstrate that a high-density ORGA can be used with a low refresh rate as well as dynamic RAM; its reconfiguration speed is 1,000,000 times faster than that of FPGAs.

Acknowledgments. The VLSI chip in this study has been fabricated in the chip fabrication program of VLSI Design and Education Center(VDEC), the University of Tokyo in collaboration with Rohm Corporation and Toppan Printing Corporation.

References

1. Altera Corporation, "Altera Devices," http:/www.altera.com/products/devices/dev-index.html
2. Xilinx Inc., "Xilinx Product Data Sheets," http://www.xilinx.com/partinfo/databook.html
3. H. Nakano, T. Shindo, T. Kazami, M. Motomura, "Development of dynamically reconfigurable processor LSI," NEC Tech. J. (Japan), vol. 56, no. 4, pp. 99–102, 2003.
4. U. Tangen, J. S. McCaskill, "Hardware evolution with a massively parallel dynamically reconfigurable computer: POLYP," Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98 Proc., pp.364–371, 1998.
5. T. H. Szymanski, M. Saint-Laurent, V. Tyan, A. Au, and B. Supmonchai, "Field-programmable logic devices with optical input-output," Appl. Opt., vol. 39, pp.721–732, 2000.
6. S.S. Sherif, S.K. Griebel, A. Au, D. Hui, T. H. Szymanski, and H.S. Hinton, "Field-programmable smart-pixel arrays: design, VLSI implementation, and applications," Appl. Opt., vol. 38, pp.838–846, 1999.
7. M. F. Sakr, S. P. Levitan, C. L. Giles, and D.M. Chiarulli, "Reconfigurable processor employing optical channels," Proc. SPIE - Int. Soc. Opt. Eng., vol. 3490, pp. 564–567, 1998.
8. M. Watanabe, J. Ohtsubo, "Digital associative memory neural network with optical learning capability," Opt. Commun., vol. 113, pp.31–38, 1994.
9. J. V. Campenhout, H. V. Marck, J. Depreitere, and J. Dambre, "Optoelectronic FPGAs," IEEE J. Sel. Top. Quantum Electron, vol. 5, pp. 306–315, 1999.
10. J. Mumbru, G. Panotopoulos, D. Psaltis, X. An, F. Mok, S. Ay, S. Barna, and E. R. Fossum, "Optically Programmable Gate Array," Proc. SPIE - Int. Soc. Opt. Eng., vol. 4089, pp. 763–771, 2000.
11. J. Depreitere, H. Neefs, H. V. Marck, J. V. Campenhout, R. Baets, B.Dhoedt, H. Thienpont, and I. Veretennicoff, "An optoelectronic 3-D field programmable gate array," FPL '94. Proc., pp.352–360, 1994.
12. J. Mumbru, G. Zhou, X. An, W. Liu, G. Panotopoulos, F. Mok, and D. Psaltis, "Optical memory for computing and information processing," Proc. SPIE - Int. Soc. Opt. Eng., vol. 3804, pp. 14–24, 1999.
13. M. Watanabe, F. Kobayashi, "An optically differential reconfigurable gate array and its power consumption estimation," IEEE International Conference on Field-Programmable Technology, pp. 197–202, 2002.
14. M. Watanabe, F. Kobayashi, "An Optically Differential Reconfigurable Gate Array with dynamic reconfiguration circuit," 10th Reconfigurable Architectures Workshop, pp. 188, 2003.

Evolvable Hardware for Signal Separation and Noise Cancellation Using Analog Reconfigurable Device

D. Keymeulen, R. Zebulum, A. Stoica, V. Duong, and M.I. Ferguson

Jet Propulsion Laboratory,
4800 Oak Grove Laboratory
Didier.keymeulen@jpl.nasa.gov
http://ehw.jpl.nasa.gov

Abstract. Evolvable systems in silicon are third generation hardware in terms of flexibility. The first generation was fixed silicon: once a device was fabricated its structure was forever fixed. Reconfigurable hardware came as a second generation: new configurations could be downloaded changing the function of the device and also bypassing faulty areas, if any. The third generation is that of self-configurable, evolvable hardware (EHW), and adds the automatic reconfiguration feature, enabling truly adaptive hardware. This paper addresses current efforts in building and using evolvable chips. The first section refers to evolutionary algorithms for evolvable hardware. The second section describes the JPL evolvable hardware testbed and the JPL Field Programmable Transistor Array (FPTA) chip designed and used for circuit evolution in silicon. The third section addresses the application of evolvable hardware for signal separation and noise cancellation. The final section concludes the work.

1 Evolutionary Algorithms for EHW

The application of evolution-inspired formalisms to hardware design and self-configuration leads to the concept of Evolvable Hardware (EHW) [5]. In the narrow sense, EHW refers to self-reconfiguration of electronic hardware by evolutionary/genetic reconfiguration mechanisms. In a broader sense Evolvable Hardware can be considered a tool for automatic circuit design.

Conventional design automation techniques explore a small fraction of the design space, consisting of standard circuit topologies (logic gates, OpAmps). In the case of digital design, there are many commercially available CAD tools that perform automatic synthesis of complex digital circuits from high level specifications; and it is based on standard heuristics and algorithms. These techniques used a library of well-known topologies of logic gates, flip-flops, etc. In the case of analog design, automation is a more challenging task because analog building blocks are more sensitive to the fabrication technology.

On the other hand, Evolutionary Algorithms' representational potential allows the exploration of a larger fraction of the design space compared to conventional tools. Instead of using standard topologies for digital gates and analog building blocks,

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 270–278, 2004.

evolutionary design can find new topologies and larger circuits can be built based on this new design library. The main steps for the evolutionary design of electronic circuits are the circuit representation and evaluation[6]. The genetic search in EHW is tightly coupled with a coded representation that associates each circuit to a "genetic code" or chromosome. The simplest representation of a chromosome is a binary string, a succession of 0s and 1s that encode a circuit. The status of the switches (ON or OFF) determines a circuit topology and consequently a specific response. Thus, the circuit topology can be considered as a function of switch states, and can be represented by a binary sequence, such as “1011…”, where a ‘1’ is associated to a switch turned ON and a ‘0’ to a switch turned OFF.

The main steps of evolutionary synthesis are illustrated in Figure 1 [5]. First, a population of chromosomes is randomly generated. The chromosomes are converted into circuit models for evaluation in SW (extrinsic evolution) or into control bit strings downloaded to programmable hardware (intrinsic evolution). Circuit responses are compared against specifications, and individuals are ranked based on how close they come to satisfying them. In preparation for a new iteration, a new population of individuals is generated from the pool of best individuals in the previous generation. This is subject to a probabilistic selection of individuals from a best individuals pool, followed by two operations: random swapping of parts of their chromosomes, the *crossover* operation, and random flipping of chromosome bits, the *mutation* operation. The process is repeated for several generations, resulting in increasingly better individuals. Randomness helps to avoid getting trapped in local optima. Monotonic convergence (in a loose Pareto sense) can be forced by unaltered transference to the next generation of the best individual from the previous generation [6]. There is no theoretical guarantee that the global optimum will be reached in a useful amount of time; however, the evolutionary/genetic search is considered by many to be the best choice for very large, highly unknown search spaces. The search process is usually stopped after a number of generations, or when closeness to the target response has reached a sufficient degree. One or several solutions may be found among the individuals of the last generation.

2 JPL Evolvable Hardware Testbed

A Stand Alone Board Level Evolvable System (SABLES) was developed as a testbed for autonomous portable experiments. SABLES is a stand-alone platform, integrating the Field Programmable Transistor Array (FPTA) chip and a digital signal processing (DSP) implementing the Evolutionary Platform (EP) as shown in Figure 2 The system is stand-alone and is connected to the PC only for the purpose of receiving specifications and communicating back the results of evolution for analysis.

The evolutionary algorithm was implemented on a DSP that directly controlled the FPTA, together forming a board-level evolvable system with fast internal communication ensured by a 32-bit bus operating at 7.5MHz [1]. Over four orders of magnitude speed-up of evolution was obtained on the FPTA chip compared to SPICE

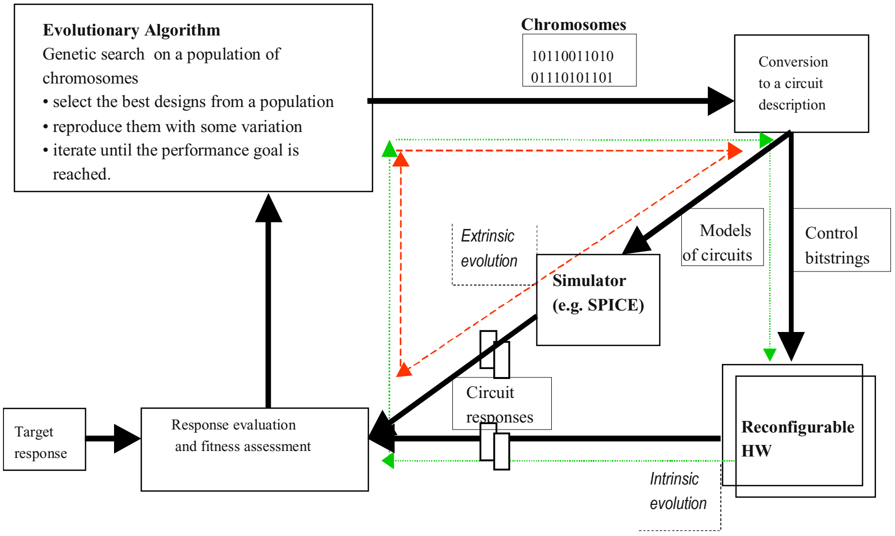

Evolutionary Algorithm
Genetic search on a population of chromosomes
• select the best designs from a population
• reproduce them with some variation
• iterate until the performance goal is reached.
Chromosomes
10110011010
01110101101
Conversion to a circuit description
Extrinsic evolution
Simulator (e.g. SPICE)
Models of circuits
Control bitstrings
Circuit responses
Reconfigurable HW
Target response
Response evaluation and fitness assessment
Intrinsic evolution

Fig. 1. Main steps for the evolutionary synthesis of electronic circuits

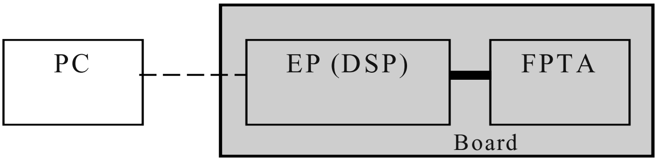


Fig. 2. Block diagram of our stand-alone board level evolvable system (SABLES)

simulations on a Pentium processor (this performance figure was obtained for a circuit with approximately 100 transistors; the speed-up advantage increases with the size of the circuit).

The FPTA is an implementation of an Evolution-Oriented Reconfigurable Architecture (EORA) that is in detail described in reference [2]. The lack of evolution-oriented devices, in particular for analog, has been an important stumbling block for researchers attempting evolution in intrinsic mode (with evaluation directly in hardware). Extrinsic evolution (using simulated models) is slow and scales badly when performed with accurate circuit models e.g. in SPICE. Less accurate models may lead to solutions that behave differently in hardware than in software simulations.

Several aspects necessary for EORA were considered during the FPTA design. The granularity of the programmable chip is an important feature. The first limitation of commercial FPGAs and field programmable analog array (FPAAs) is their coarse granularity for use in evolution. From the EHW perspective, it is interesting to have *programmable granularity*, allowing the sampling of novel architectures together with the possibility of implementing standard ones. It also allows to choose, depending of the task, the optimal choice of elementary block type and granularity. Virtual higher-level building blocks can be considered by imposing programming constraints. EORA should also be *transparent architecture*, allowing the analysis and simulation of the evolved circuits. It should also be robust enough not to be damaged by any bit-string configuration existent in the search space, potentially sampled by evolution. Finally EORA should allow evolution of both analog and digital functions.

The FPTA chips designed at JPL meet the above requirements, and are particularly targeted for EHW experiments. The first versions of the FPTA (FPTA-0 and FPTA-1) relied on a cell with 8 transistors interconnected by 24 switches. They were used to demonstrate intrinsic evolution of a variety of circuits, including logical gates, transconductance amplifiers, computational circuits, etc [2].

The most recent version, FPTA2, is a second generation reconfigurable mixed signal array chip whose cells can be programmed at the transistor level [3]. The chip architecture is described in details in [2]. It consists of an 8x8 matrix of reconfigurable cells. The chip can receive 96 analog/digital inputs and provide 64 analog/digital outputs. Each cell is programmed through a 16 bits data bus/9 bits address bus control logic, which provides an addressing mechanism to download the bit-string of each cell. A total of 5000 bits is used to program the whole chip. The FPTA-2 cell consists of 14 transistors connected through 75 switches and it is able to map different building blocks for analog processing, such as two- and three- stage Operational Amplifier (OpAmps), logarithmic photo-detectors, or Gaussian computational circuits. Figure 3 shows the details of the cell for the latest version of the FPTA chip.

3 Flexible Evolvable Hardware

This section describes two applications of SABLES in the domain of flexible hardware, encompassing the evolution of signal separators and elimination of noise from a voice signal in real time. The first experiment demonstrates that our evolvable hardware system is able to design a circuit able to separate two mixed signals using the knowledge of frequency of both signals to identify the correct circuit. The second experiment goes one step further by designing a circuit able to filter the noise from an unknown and dynamic signal.

3.1 Signal Separation Experiments

The goal of this experiment was to design a circuit able to separate two mixed signals, $E_1(t)$ and $E_2(t)$, obtained by the linear combination of pure sine waves, $e_1(t)$ and $e_2(t)$, of known frequencies. The circuit outputs are the original pure sine waves. We chose for the frequency of the pure sine wave $e_1(t)$, f_1 = 10kHz and for the pure sine wave $e_2(t)$, f_2 = 20kHz. These signals were linearly combined by a mixing matrix to produce the chip inputs $E_1(t)$ and $E_2(t)$ as shown below:

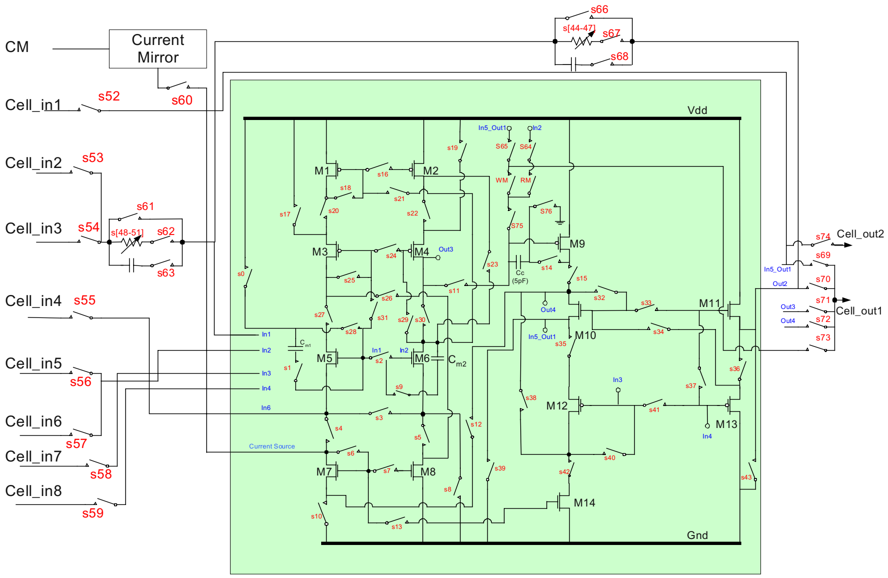


Fig. 3. Schematic of cell transistor array of the FPTA-2. Cell inputs are labeled Cell_in1, Cell_in2 until Cell_in8. Cell outputs are labeled Cell_out1 and Cell_out2. Switches are labeled s0, s1 until s75. Fixed transistors are labeled M1, M2 until M14

$$\begin{bmatrix} E_1(t) \\ E_2(t) \end{bmatrix} = \begin{bmatrix} 0.25 & 0.5 \\ 0.5 & 0.25 \end{bmatrix} \begin{bmatrix} \sin(2\pi 10{,}000t) \\ \sin(2\pi 20{,}000t) \end{bmatrix}$$

The GA parameters selected for this experiment were: 70% mutation rate; 20% crossover rate; replacement factor of 20%; population of 400; and 100 to 200 generations. These parameters are the results of multiple runs and have given the best result. A binary representation was used, where each bit determines the state (opened, closed) of a switch. Each evolution took about 5 minutes in the SABLE system. More than 20 different GA executions were performed. In order to compute the fitness function, the fast Fourier Transform (FFT) of the output signal(s) from the FPTA-2 was calculated. The fitness was a measure of the error of the FPTA-2 outputs to the target values in each experiment, as shown below.

$$Fitness = \sum_{i=1}^{N} | O_f(i) - T_f(i) |$$

where N is the number of samples used in the FFT (usually 64), $O_f(i)$ is the magnitude of the ith FFT component of the FPTA-2 output, and $T_f(i)$ is the target magnitude of the ith FFT component. Other fitness measures such as the sum of the squared deviations between the output and the target were tried, without significant improvement.

Another important feature of these experiments refers to the control voltages for operation of switches. The switches of the re-configurable chip are implemented as transmission gates. The control voltages that completely open or close the switches are 0 and 2V. However, through experimentation, it has been observed that the results significantly improve when the values of 0.4V and 1.6V are used to control the switches, meaning that they are now partly opened and closed (partly closed if the higher and lower control voltages are respectively applied to the NMOS and PMOS transistors of the switch, and partly opened in the other case).

Finally, another important issue is the search space size. If we allow a completely unconstrained evolution, we will end up with a very large search space size. One approach to reduce the search space size is to have the FPTA-2 cells constrained to a particular topology, so that only the interconnections among the cells are evolved. Through experimentation, it has been verified that the constrained approach delivered better results. This experiment was performed using 10 cells of the FPTA-2, and the cell topologies were fixed to one of inverting amplifiers. Figure 4 depicts a block diagram of the FPTA-2 chip, showing its 64 cells and the chip corner used in the experiment, circuit inputs (E1 and E2) and outputs (O1 and O2).

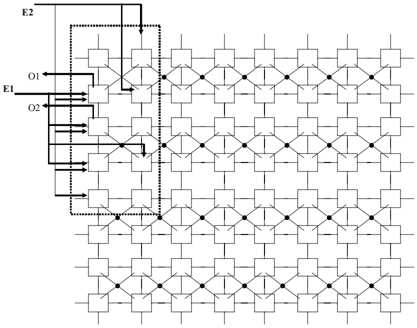


Fig. 4. Block diagram of the FPTA-2 chip for the signal separation experiment.

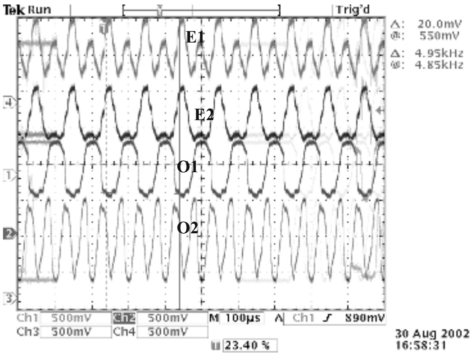


Fig. 5. Result of the signal separation experiment. At the top the inputs E1 and E2 are shown. At the bottom the outputs O1 (10kHz)) and O2 (20kHz) are shown.

Figure 5 depicts the best inputs and outputs of the evolved circuit achieved in this set of executions. Table 12-1 summarizes the evolved circuit performance in terms of the FFT and of the values measured in the frequency analyzer for 10kHz and 20kHz frequency component. The target FFT values used in the fitness function are also included in the table.

Table 1. Analysis of the evolved circuits in the signal separation experiment with signal frequencies of 10 & 20 kHz . Amplitude of 10kHz and 20kHz tones as measured by the spectrum analyzer and calculated by an FFT algorithm used during evolution

	10kHz Tone		**20kHz Tone**	
	FFT	**Spectrum Analyzer**	**FFT**	**Spectrum Analyzer**
Input E_1	33.6	-13dB	8.86	-20dB
Input E_2	7.6	-20dB	41.1	-15dB
Output O_1	36.9 *(Target: T_f>20)*	-13dB	1.07 *(Target: T_f= 0)*	-35.3dB
Output O_2	0.6 *(Target: T_f= 0)*	-30dB	84.5 *(Target: T_f>20)*	-13dB

From Table 1, it can be observed (output O_1) that the evolved circuit attenuates the input signal component e_2 (20kHz) by –15.3dB (from –20dB to –35.3dB), while keeping the input signal e_1 (10kHz) at the same level. On the other hand (output O_2), the evolved circuit attenuates the input signal component e_1 by -10dB (from –20dB to –30dB), and amplifies e_2 by 2dB (from –15dB to –13dB).

This experiment is a first approach to tackle the independent component analysis problem, which consists of recovering the original source signals from signals acquired by a set of sensors that pick up different linear combinations of source signals.

3.2 Real-Time Noise Elimination

The objective of this experiment was to evolve a circuit that can automatically filter the noise from a dynamic radio signal. Two signals (radio signal from 0.5kHz to 10kHz and the noise signal at 7kHz) were linearly combined to produce the input

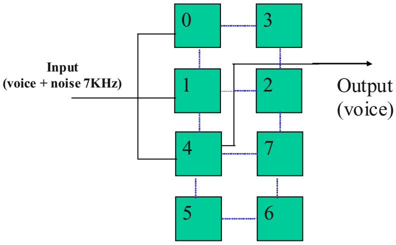


Fig. 6. FPTA-2 cells used in the noise elimination experiment

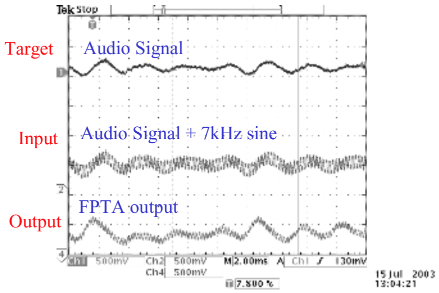


Fig. 7. Response in the time domain of the circuit evolved for real-time noise elimination

signal E1(t). The challenge of the experiment was to evolve a circuit using a non-stationary voice signal acquired by the microphone. The evolved circuit must thus be able to generalize to voice signals not exposed during evolution/training. The evolved circuit was obtained in less than 1 minute after 50 generations of evolution on the SABLES platform. The GA parameters selected for this experiment were: 20% mutation rate; 70% crossover rate; replacement factor of 10%; population of 500; and 100 to 200 generations.

The fitness is computed by taking the FFT of the circuit output. The 7kHz component of the FFT of the circuit output is found and the objective is to minimize it. The evolution algorithm uses 6 cells of the FPTA-2 as shown in Figure 6.

Figure 8 depicts the platform used in the experiment. Besides SABLES, a Field Programmable Analog Array (FPAA) chip from Anadigm [4] was used for audio signal conditioning before applying it to the FPTA chip. The microphone acquires a

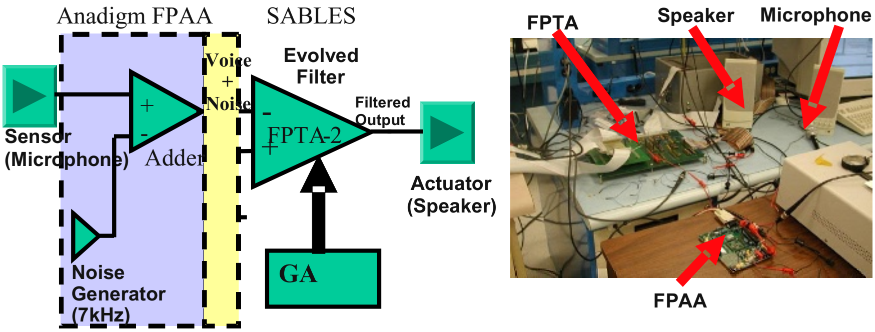


Fig. 8. Platform for real-time noise elimination experiment. Block diagram is on the left and picture of the apparatus is to the right

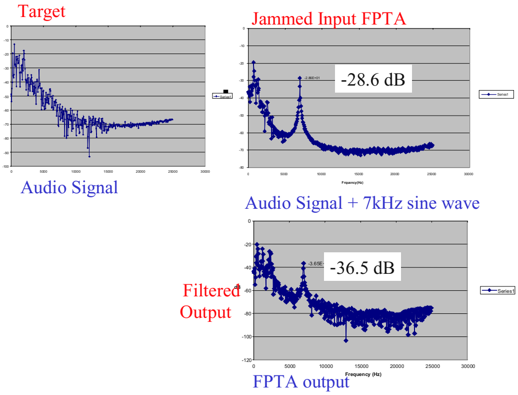


Fig. 9. Response in the frequency domain of the circuit evolved for real-time noise elimination.

voice signal coming from a radio in real-time which is mixed with a noise signal and conditioned for the FPTA-2 (shift the DC value). The FPTA-2 is evolved to separate the two signals.

Figure 7 shows the input and output signals in the time domain. It can be seen that the FPTA output signal has the 7 kHz noise attenuated compared to the input, while keeping the audio signal level. Figure 9 displays the same information in the frequency domain, where it can be seen that the 7kHz noise signal is attenuated by -8dB (from -28.6 to -36.5dB).

These are preliminary results, and there is still some room for improvement, such as evolving a circuit that can adapt to jamming signals at different frequencies..

4 Conclusion

The main objective of this paper was to demonstrate new capabilities for flexible electronics using evolvable hardware. We demonstrated the evolution of circuits using the FPTA cells that can automatically filter the noise signal from a dynamic radio signal. Evolvable hardware technology is particularly significant for NASA's future autonomous systems, providing on-board resources for reconfiguration to self-adapt to situations, environments, and mission changes. It would enable future space missions using concepts of spacecraft survivability in excess of 100 years that poses difficult challenges to current electronics

Acknowledgement. The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology and was sponsored by the National Aeronautics and Space Administration.

References

[1] Ferguson, M.I., Stoica A., Zebulum R., Keymeulen D. and Duong, V. "An Evolvable Hardware Platform based on DSP and FPTA", Proceedings of the Genetic and Evolutionary Computation Conference, July 9-13, 2002, pp145-152, New York.

[2] Stoica, A. et al., "Reconfigurable VLSI Architectures for Evolvable Hardware: from Experimental Field Programmable Transistor Arrays to Evolution-Oriented Chips", *IEEE Trans. on VLSI*, IEEE Press , V. 9, N. 1, pp. 227-232, February 2001.

[3] Stoica, A. et al.."Evolving Circuits in Seconds: Experiments with a Stand-Alone Board Level Evolvable System", 2002 NASA/DoD Conf. on Evolvable Hardware, July 15-18, 2002, IEEE Computer Press, pp.67-74.

[4] Anadigm, Inc., "Evaluation Board User Manual", (http://www.anadigm.com)

[5] Stoica, A., Lohn J., Keymeulen D., Zebulum R. Proceedings of "NASA/DoD Conference on Evolvable Hardware", July 1999 – June 2003. IEEE Computer Society

[6] Zebulum R., Pacheco M., Vellasco M.. "Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms". CRC Press, 2002

Implementing High-Speed Double-Data Rate (DDR) SDRAM Controllers on FPGA

Eduardo Picatoste-Olloqui, Francisco Cardells-Tormo[1],
Jordi Sempere-Agullo[1,2], and Atilà Herms-Berenguer[2]

[1] Hewlett-Packard, R&D Technology Lab, Digital ASICs,
08174 Sant Cugat del Valles (Barcelona), Spain
{francisco.cardells,jordi.sempere}@hp.com

[2] University of Barcelona (UB), Department of Electronics,
Marti i Franques 1, 08028 Barcelona, Spain
herms@el.ub.es

Abstract. This paper deals with the FPGA-implementation of a high-speed interface for DDR SDRAMs. We aim to achieve a performance, in terms of bandwidth, comparable to ASIC implementations. The novelty of this paper is to present the design techniques that lead to high performance memory controllers. First of all, we compile the specific hardware features available in FPGA families. In the second place, we depict a memory interface data path architecture adapted for implementation on Xilinx and Altera FPGAs. Finally, we explain the design rules to meet timing requirements (round trip delay) for successful operation. The discussion is complemented with timing measurements for a Virtex-II based memory interface and with timing calculations performed for Stratix.

1 Introduction

Image processing data pipelines require a large amount of data to be buffered. Memory is the place where integrated circuits (ICs) hold current data for the pipelines. Several types of memory could be used for this purpose, yet each has their pros and cons. Static RAMs (SRAMs) are fast and have reduced power consumption, but their low density - several transistors are required to store a single bit of data - increases their cost and limits their use.

On the other hand, Dynamic RAMs (DRAMs) have slower performance and higher power consumption, but they are denser and cheaper than SRAMs. These characteristics make them a better choice for the main memory in digital imaging systems. There are several types of DRAM. Previous types of DRAM include Fast Page Mode DRAM (FPM DRAM) and Extended Data Out DRAM (EDO DRAM) that are asynchronous. These are quite slow for current systems. Synchronous DRAM (SDRAM), Double-Data Rate (DDR) SDRAM, and RAMBus DRAM (RDRAM) are currently mainstream in the PC industry.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 279–288, 2004.

SDRAM is synchronous, meaning that the device directly depends on the clock speed driving the device. With SDRAMs, latency is significantly reduced when compared to previous DRAM technologies. Inputs and outputs are simplified in terms of signal interfacing. SDRAM is still used but is being quickly replaced by DDR SDRAM.

RDRAM narrows the memory bus to 16-bit, and uses a faster bus operation (up to 800MHz). Although the bandwidth is incremented respect to SDRAM, it has a big drawback: it is a proprietary technology.

DDR SDRAM is a natural evolution of SDRAM. Data transactions are done in both edges of the clock, thus doubling the raw bandwidth of the data path. Performance is improved as with RDRAM but costs are lower. Due to widespread adoption by the PC industry and improved long-term availability over SDRAM, it makes a good choice for imaging applications.

With the advent of modern process technologies (13μm and beyond), FPGAs are being considered for high-volume applications. FPGAs have evolved in terms of complexity and density and they are now not very far from ASIC performance and capacity. Therefore they can now be used in demanding applications that were not possible before. For instance, most of Xilinx and Altera FPGAs include enough features to implement a high-speed memory controller design.

FPGAs are now supporting the features needed to interface a DDR SDRAM. FPGA I/Os are now compatible with the SSTL-II electrical interface, I/Os include DDR registers, and phase locked loops (PLLs) are flexible enough to allow different clock frequencies and meet data capture timings. In some Altera families, hard-macro delay chains for data strobes have been added, a feature reserved to special ASIC cores [1]. The FPGA based intellectual propriety (IP) cores market offers memory controllers designed to make the most out of these features.

In conclusion, FPGAs have improved enough to implement DDR interfacing both for products and prototypes. Modern FPGA families, combined with IP cores, provide features for achieving a bandwidth comparable to ASIC designs. The goal and novelty of this paper is to compile the design techniques to implement high-speed memory controllers. In this paper we summarize the features to be taken into account when choosing an FPGA for high-speed DDR SDRAM interfacing, we depict a memory controller data path architecture that can be implemented with Altera and Xilinx FPGAs and we explain the design rules to meet timing requirements. We show timing measurements of a Virtex-II FPGA interfacing a Samsung K4H560838C DDR SDRAM device at 60MHz. We perform the read data capture timing analysis for a Stratix FPGA interfacing a Samsung M470L6524BT0 DDR SDRAM device at 80MHz.

2 DDR SDRAM Operation

The DDR SDRAM [2] operates from a differential clock (CLK and CLKn; the crossing of CLK going HIGH and CLKn going LOW will be referred to as the positive edge of CLK). Commands (address and control signals) are registered at every positive edge of CLK. A bidirectional data strobe (DQS) is transmitted externally, along with data, for use in data capture at the receiver. DQS is a strobe transmitted by the DDR SDRAM during reads and by the memory controller during writes. This strobe associated with data works like an independent clocking scheme and it is necessary to meet the demanding requirements in terms of timing. DQS is edge–aligned with data for reads and center–aligned with data for writes. Memory input data is registered on both edges of DQS, and output data is referenced to both edges of DQS, as well as to both edges of CK. Therefore, because the interface operates in both edges, the interface data width (n) is half the memory data size (2n). In figure 1 we have depicted its operation.

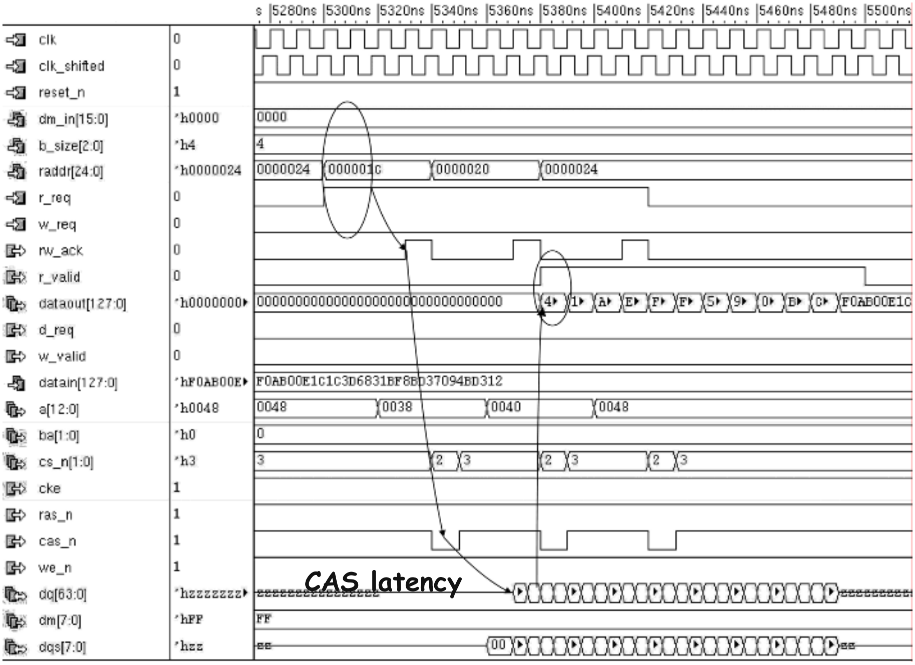


Fig. 1. The figure shows the simulation waveforms for a DDR SDRAM read operation. The clock frequency used in the simulation is 100MHz. The controller used for the simulation is the Altera DDR SDRAM Controller MegaCore IP [3].

Read and write accesses to the DDR SDRAM are burst oriented; accesses start at a selected location and continue sequentially for a programmed number of locations. Accesses begin with the registration of an ACTIVE command (RAS_n low), which is then followed by a READ (CAS_n low) or WRITE (CAS_n low and WE_n low) command. The address bits registered coincident with the ACTIVE command are used to select the bank and page (row) to be accessed. The address bits registered coincident with the READ or WRITE command are used to select the bank and the starting column location for the burst access. A READ command has been simulated using an Altera DDR SDRAM controller core [3] and it has been depicted in figure 1.

When a read command is issued data appears on the data bus 1.5 to 3 clock cycles later. This delay is known as the CAS latency and is due to the time required to read the internal DRAM core and register the data on the bus. The CAS latency depends on the speed of the SDRAM and the frequency of the memory clock. The DDR SDRAM provides for programmable read or write burst lengths of 2, 4 or 8 locations.

A PRECHARGE operation may be enabled to close the bank if a different row must be accessed. The AUTO-REFRESH command is issued periodically to ensure data retention.

3 FPGA Features for DDR SDRAM Interfacing

When choosing an FPGA for DDR SDRAM interface, one should check what relevant features are present. First of all, the presence of SSTL II signaling avoids the use of external buffers, and its associated overhead in terms of cost and delay. Secondly, the availability of I/Os with DDR registers saves flip-flops from the main logic and reduces the propagation delays. Thirdly, a 90^0 delay chain in the DQS path allows a direct DQS read data capture as opposed to an asynchronous read data capture. The impact of the data capture scheme in the architecture will be discussed in a later section. Finally, system bandwidth is conditioned by both the interface performance and the maximum data path width. Unfortunately some families have limitations related to this parameter. We have summarized in table 1 the features embedded in each family.

Table 1. Available features supporting DDR SDRAM for FPGA families. In the table we provide the expected performance claimed for each FPGA family (considering the fastest speed grade, and DQS data capture) by the MegaCore [3] and Northwest Logic [4] IP core documentation (available on-line) for Altera and Xilinx respectively.

FPGA Family	Vendor	SSTL II support	I/Os w/ DDR FFs	Delay Chain	Max. Datapath Width (bits)	Expected Performance
APEX II	Altera	✓	✓	✓	64	133 MHz
Virtex II	Xilinx	✓	✓		64	200 MHz
Cyclone	Altera	✓		✓	32	133 MHz
Stratix	Altera	✓	✓	✓	64	200 MHz
Spartan 3	Xilinx	✓	✓		64	133 MHz

Although FPGAs have been selectively adding these features to their fabric, still in modern FPGAs, only high-end FPGAs have all of them. Due to this fact, we have added a column summarizing the expected performance of the interface. The figures have been obtained from the IP core datasheets available for each device: MegaCore [3] and Northwest Logic [4] DDR SDRAM controller cores for Altera and Xilinx respectively. Table 1 clearly shows that the required performance of the memory interface conditions the FPGA family of choice. Other families not shown in the table are not suitable for high-speed operation due to the fact that they do not have dedicated hardware, at most I/Os compatible with SSTL II.

4 Memory Interface Data Path Architecture

We have depicted in figure 2 the architecture of the data path interface. In this figure we show from top to bottom the data path for the DQ (data), DQS (strobe) and clock for both the read and write directions. This architecture can be implemented in any FPGA family either using dedicated resources, such as IOs with DDR FFs, or general purpose logic with the consequent cost in terms of resources and performance.

In order for the memory interface to work, we should provide two clocks shifted 90 degrees between them, we will refer the memory nominal clock as DDR_clk, while its delayed version as DDR_clk_90. The DDR_clk_90 corresponds to the system clock driving the logic within the FPGA. The DDR_clk clock drives the memory device. We know that the DDR_clk and the DQS signals should arrive with very little skew to the device. Due to this fact the generation of both signals uses a similar scheme based on DDR flip-flops. Yet, on writes data should arrive earlier than the strobes. Due to this fact the DQ output is clocked with DDR_clk.

4.1 DQS Read Data Capture

During read operations, the DDR SDRAM device output DQ and DQS have simultaneous edges. To use DQS as a strobe to capture DQ, the DQS signal must be delayed 90º with respect to DQ within the FPGA. This delay could not be applied using a phase-locked loop (PLL) because the DQS signal does not have the properties of a clock. Instead, a delay chain should be put between the DQS and the read data clocking scheme. This scheme is depicted in figure 2.

Cyclone and Stratix families include special delay element chains on the DQS input path, which generates a 90º phase shift of DQS relative to DQ. The Stratix family also incorporates a DQS phase-shift reference circuit (a delay locked-loop) that continuously and automatically adjusts the phase shift to keep it at 90º, so this delay is independent of process, voltage, and temperature (PVT) variations. The circuit requires a reference clock equal in frequency to DDR SDRAM clock. The DQ, DM, and DQS trace lengths have to be tightly matched within each byte group so that the edges of the DQ and DQS nominally arrive at the FPGA aligned. Virtex-II and Spartan 3 families do not have a hard-macro performing a delay chain. Nevertheless, it can be built-up using LUTs. This scheme is used in the Northwest DDR SDRAM Controller and Altera DDR SDRAM Controller MegaCore IPs for Xilinx and Altera respectively.

4.2 Asynchronous Read Data Capture

Less modern Altera FPGAs do not have a delay chain. For Xilinx FPGAs, there might not be enough logic resources to implement the delay chain using logic cells. Under those circumstances, data cannot be captured in the FPGA using the DQS signal. Instead, an internal clock, a 2x version of the system clock, is generated to mimic the DRAM strobes [5]. This version is typically used for lower operating frequencies (under 133MHz).

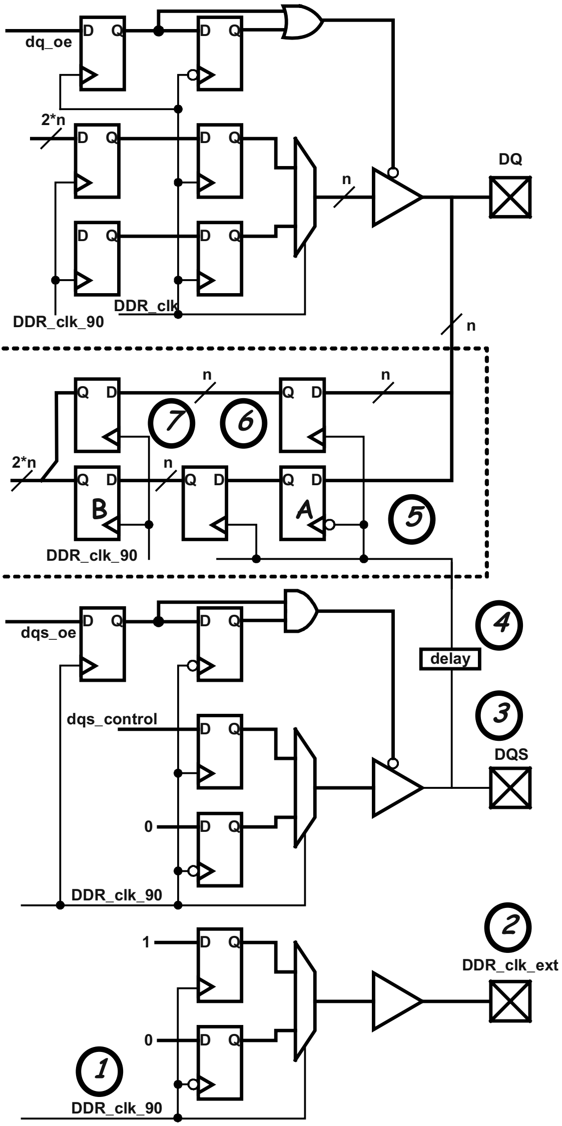


Fig. 2. Data Path for the DQS Read Data Capture. The dashed rectangle encloses the read data capture and resynchronization scheme.

5 Timing Analysis

Meeting timing is the most critical part when designing a high-speed interface. Potential variations in timing such as board layout, FPGA placement, and set up of the clocks in the system, must be taken into account. This section describes the timing analysis followed in the FPGA-implementation of a memory controller. The measurements shown in this section correspond to a DDR SDRAM interface implemented in a Virtex-II and running at 60 MHz. Our clock frequency was limited by the FPGA speed factor and by another clock domain using a multiple of the DDR frequency.

5.1 Write Data Timing

Two configurations can be used to generate the memory clock. The external clock could be provided by a dedicated PLL output, with a phase delay to help achieve resynchronization. In addition, this clock can also be generated using the DDR flip-flops following the configuration used for the DQS signal generation for memory writes as in figure 2.

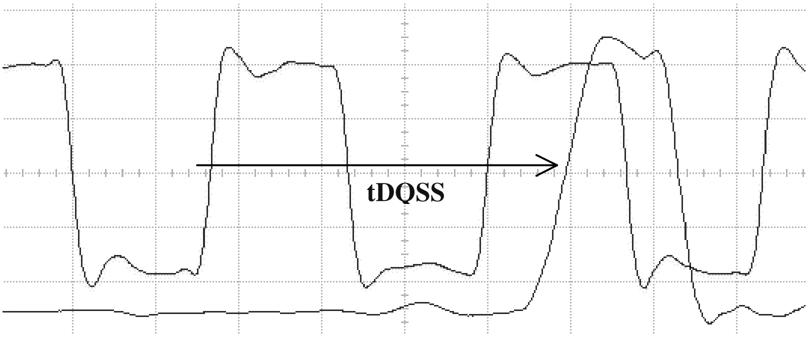


Fig. 3. Measured DQS[0] and CLK signals at the DDR SDRAM device pins. In this figure we show the write command to first DQS transition time (tDQSS). The measurement obtained with a Virtex II-based board gives a value for tDQSS of 1.23 tCLK.

The aforementioned delays may cause a mis-alignment between the clock and DQS at the FPGA pins. Fortunately, the JEDEC standard [2] allows sufficient skew between the clock and the DQS and given by the following expression:

$$tDQSS = (1.00 \pm 0.25)tCLK . \quad (1)$$

This condition should be measured as in figure 3. In the FPGA implementation of figure 2, data strobe signals are generated with the positive edge of the clock DDR_clk to meet tDQSS requirement.

5.2 Address and Command Timing

DDR SDRAM address and command inputs typically require symmetrical 1ns setup (tIS) and hold times (tIH) respect to the DDR SDRAM clock. Address and command outputs can be generated either on the positive or the negative clock edge (internal to the FPGA). The negative edge should normally be used to satisfy setup and hold times so the edge is center-aligned with address and commands as depicted in figure 4.

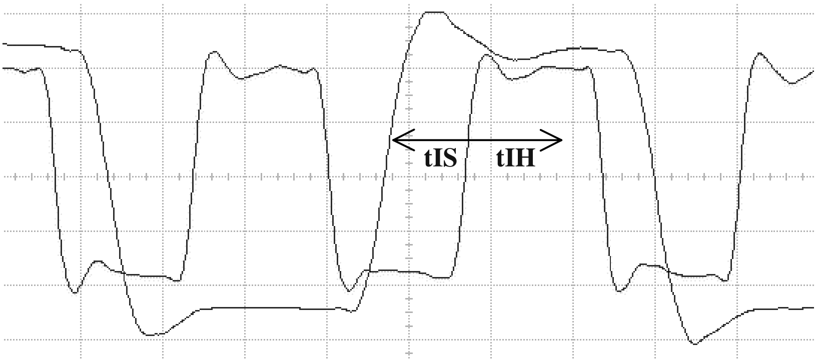


Fig. 4. Measured A[0] and CLK signals at the DDR SDRAM device pins. In this figure we show the address line setup and hold times for both the CLK rising and falling edges. The measurements were carried out with a Virtex II-based board. We obtain setup and hold times for both the falling (tIS=4.6ns,tIH=11.4) and rising edge (tIS=5.3ns,tIH=11.6) higher than the minimum specified by JEDEC (0.9ns).

5.3 Resynchronization of Captured Read Data

In the DQS read data capture, read data is captured into the DDR registers of the FPGA using DQS signals as a clock. To present data out synchronously at the local-side interface, data must be transferred from the DQS clock domain to the system clock (DDR_clk_90) domain in what can be called resynchronization.

Table 2. Delay paths for round trip delay calculation. Point numbers are associated to fig. 2.

Delay path description	Name	Points in fig. 2
FPGA DDR_clk_90 to Clock FPGA pin	tPD	1 to 2
Clock FPGA pin to Clock DRAM pin	tPD	(trace delay)
Clock DRAM pin to DQS DRAM pin on a read	tDQSCK	
DQS DRAM pin to DQS FPGA pin	tPD	(trace delay)
DQS phase shift (~90 degrees + jitter)		3 to 4
From previous to clock in put at register A	tPD	4 to 5
From previous to output in register A	tCO	5 to 6
From register A output to register B input	tPD	6 to 7

From now on we will refer to figure 2. To sample the data output from register A into register B, the phase relationship between DQS and DDR_clk_90 must be taken into account in order not to violate setup and hold conditions in register B. By calculating the round trip delay (RTD) we can obtain the window size before register B. RTD is the sum of maximum (longest) and minimum (shortest) delays related to the timing path partially depicted in figure 1 and fully described in table 2. In order to determine if data can be reliably resynchronized, minimum and maximum RTD values must allow data to be available for register B within its safe resynchronization window.

We have performed the RTD calculation for a design implemented in a Stratix FPGA with a target frequency of 80MHz. We have obtained the FPGA internal delays from Quartus II place and route results. Altera provides an Excel file that predicts the resynchronization margin [6]. It outputs a diagram showing the data capture window (Fig. 5).

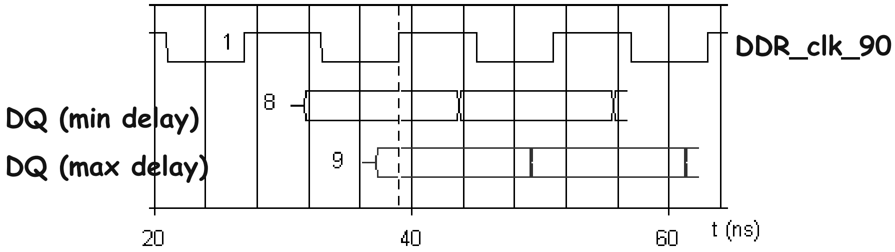


Fig. 5. Read data capture valid window. This figure has been obtained for a Stratix FPGA using the Altera round trip delay calculator [6].

6 Conclusions

In this paper we have exposed the techniques to design a high-speed DDR SDRAM memory controller for FPGAs. We have summarized the features that make certain FPGAs suitable for this application. We have described a DQS read data capture architecture and that can be implemented with commercial FPGAs from Xilinx and Altera. Finally, we have provided the fundamentals for performing the timing analysis and calculations for write and read operations in order to verify the successful operation of the interface. The timing section encloses measurements from a prototype based on Xilinx Virtex II and calculation for a design based on Altera Stratix.

Acknowledgments. Eduardo Picatoste acknowledges the support of Hewlett-Packard, Inkjet Commercial Division (ICD), R&D Large-Format Technology Lab, in the preparation of his engineering degree thesis. The opinions expressed by the authors are theirs alone and they do not represent the opinions of Hewlett-Packard.

References

1. Ryan, K.: DDR SDRAM Functionality and Controller Read Data Capture, Micron Design Line, Vol. 8, issue 3 (1999)
2. JEDEC: JEDEC Standard Double Data Rate (DDR) SDRAM Specification, JESD79. Available on-line (2002)

3. Altera Corp.: DDR SDRAM Controller MegaCore Function User Guide. Available on-line (2004)
4. Northwest Logic: DDR SDRAM Controller. Datasheet. Available on-line (2004)
5. Tran, J.: Synthesizable DDR SDRAM Controller. Xilinx Application Note XAPP200. Available on-line (2003)
6. Altera Corp.: Using the Stratix and Stratix GX DDR Round Trip Delay (RTD) Calculator. Available from Altera under request (2003)

Logic Modules with Shared SRAM Tables for Field-Programmable Gate Arrays

Fatih Kocan and Jason Meyer

Department of Computer Science and Engineering, Southern Methodist University
Dallas, TX 75275
kocan@engr.smu.edu, jason_meyer@raytheon.com

Abstract. In this paper, we propose a new area-efficient logic module architecture for SRAM-based FPGAs. This new architecture is motivated by the analysis results of some LUT-level benchmarks. The analysis results indicate that a large percentage of the LUTs in a LUT-level circuit are permutation (P) equivalent (not even including input negations or output negations, called NPN equivalences in the literature, or constant assignments). The proposed logic module utilizes lookup table sharing among two or more basic logic elements (BLEs) in a cluster, as opposed to one LUT per BLE. Preliminary results indicate that almost half of the LUTs are eliminated in all benchmarks. This great area reduction would reflect to the cost and prices of FPGAs and also would strengthen the FPGA usage in applications that have rigid area constraints such as an FPGA within a hearing aid.

1 Introduction

Reconfigurable computing has emerged as a new computing paradigm to bridge the general purpose and application-specific computing paradigms [8,12,24]. It is more flexible than application-specific computing, and much faster than general-purpose computing, as shown in several application domains including data encryption and cryptography [11, 13,20], and data, video, and image compression [7,14,15,21]. There are many other areas that benefit from reconfigurable computing due to its performance, flexibility, and so on. FPGAs are the building blocks of reconfigurable systems. There are various FPGA architectures with fine- and coarse-grained logic block structures [26,3]. Usually, an FPGA is a two dimensional array of configurable logic blocks (CLBs) that are interconnected by connection modules, wires, and switch modules. CLBs are logic blocks with one or more lookup tables (LUTs) where the truth tables of the implemented functions are stored.

In developing a larger family of FPGAs, the XC4000 family, two issues were encountered by the authors of [23]. First, as devices get larger, proportionally more interconnection is needed to take advantage of the greater capacity. Second, the number of gates that can be implemented increases quadratically with length, but the number of inputs only increases linearly with the length, so there is potential for insufficient inputs.

The authors of [16] proposed a new architecture combining FPGAs based on LUTs and CPLDs based on PALs/PLAs, which they called Hybrid FPGAs. This was based on the idea that some parts of circuits are suited for LUTs, while other parts are suited for the Product term-based structures of CPLDs.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 289–300, 2004.

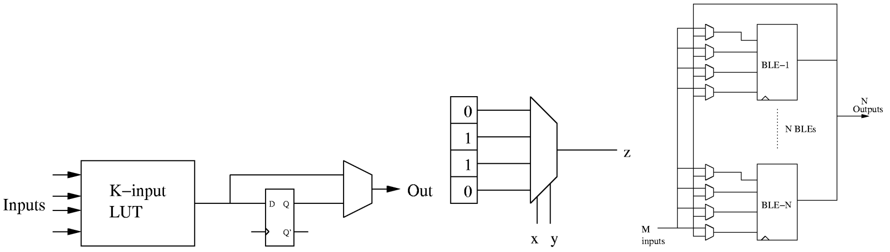
Many FPGAs use a clustered architecture where several LUTs are grouped together inside a configurable logic block. The LUT inputs can be chosen from cluster inputs, can come from other clusters, or can come from feedback of other LUTs in the cluster. Previously, these internal connections were assumed to be fully connected, where every LUT input could come from any cluster input or feedback connection. The authors of [19] proposed only sparsely populating these internal cluster connections, which succeeded in lowering the area required without affecting delay.

One important trade-off to study is how large to make clusters, since if the clusters are too large, the local interconnection would be larger than what was saved from global interconnection. The authors of [5] explored two questions: how many distinct inputs should be provided to each cluster, and how many LUTs should be included in each cluster. It was determined that for a cluster of size N, 2N + 2 inputs were sufficient. Secondly, any choice between 1 and 8 BLEs per CLB was acceptable, since they all required an area within 10% of each other.

In [1], the Vantis VF1 FPGA architecture was designed from scratch for performance and ease-of-use. This architecture included a new logic block with variable-logic granularity. It contained three levels of logic hierarchy: a Configurable-Building Block$^{\text{TM}}$, a Variable-Grain-Block$^{\text{TM}}$containing four CBBs, and a Super-Variable-Grain-Block$^{\text{TM}}$containing four VGBs.

Several new FPGA architectures have been proposed in recent years. If assumptions about CAD tools used in experimentation for these new architectures are incorrect, incorrect conclusions about these new architectures can be drawn. The authors of [27] studied the sensitivity of lookup-table size, switch block topology, cluster size, and memory size and showed that experiments are greatly affected by assumptions, tools, and techniques used in experiments.

The authors of [2] studied the effect of LUT size and cluster size on the speed and logic density of FPGAs. It was determined that for a cluster of size N with k-LUTs, the number of inputs should be k/2 * (N + 1). Also, it was determined that clusters with sizes between 4 and 10 with LUT sizes of 4 and 6 produced the best results.



(a) Basic logic element (BLE)

(b) 2-input LUT

(c) Cluster logic block of N BLEs

Fig. 1. Lookup table (LUT), Basic logic element (BLE), and Clustered Logic Block (CLB)

In this paper, we propose a new cluster configurable logic block to reduce the area of a traditional cluster CLB, and therefore, the FPGA. In turn, the cost and prices of FPGAs

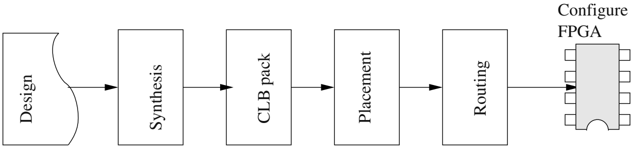


Fig. 2. FPGA/CAD flow

would drop substantially: It is well-known that cost of chips/dies grows to the fourth power of the die area, i.e. cost=(area)^4. We reduce the CLB areas by allowing an SRAM table in a CLB to be shared by 2 LUTs. Our preliminary analysis of the synthesized benchmarks indicates that a large percentage of many lookup tables are permutation (P) equivalent. This analysis result motivates us to design a new CLB that would allow mapping of multiple LUTs to one SRAM table. The remainder of the paper is organized as follows. Section 2 covers a generic SRAM-based FPGA architecture and its CAD tool. Section 3 presents the motivation behind this work. Section 4 introduces the architecture of our new CLB. In Section 5, the FPGA/CAD flow for the proposed architecture is presented. In Section 6, experimental results are included. Finally, in Section 7, we draw conclusions from the experimental results and give future directions.

2 Preliminaries for SRAM-Based FPGA Architectures

2.1 Cluster Logic Modules with SRAM Tables

A basic logic block consists of one basic logic element (BLE). Figure 1 depicts the components of a BLE. The BLE in Figure 1(a) consists of a K-input lookup table (LUT), a flip-flop, and a multiplexer. The details of a 2-input LUT are shown in Figure 1(b). A LUT has an array of SRAM cells and a selector controlled by the input literals of the stored truth table. In this particular example, we store the truth table of an xor function, (that is $z = x \oplus y$). Note that in a 2^k size SRAM table, we can realize any function of k inputs. A clustered logic block consists of an array of BLEs as shown in Figure 1(c). The cluster has M inputs and N outputs and allows sharing of M inputs among the BLEs as well as feeding the outputs of BLEs back to the inputs of other BLEs.

2.2 Computer-Aided Design (CAD) for FPGAs

There are tools that enable automated implementation of designs/circuits/architectures on FPGAs. The design can be specified with a hardware description language such as Verilog and VHDL or a tool that allows schematic capture. The overall CAD flow is illustrated in Figure 2. In order to map a design onto FPGAs, first the design is technology mapped using synthesis tools such as SIS [10] and RASP [9] tools. Second, the synthesized design is placed and routed using tools for this task such as VPR [4]. Finally, the FPGA is configured with a bit stream to implement the functionality.

3 Motivation

Prior to the introduction of SRAM-based logic modules, researchers investigated universal logic blocks that would support a great majority of logic functions with minimum area. The proposed universal logic blocks have the capability of implementing functions that are negated at the primary inputs, permuted at the inputs and negated at the primary outputs, referred to as NPN equivalence [6,22]. The definition of P equivalence follows.

Definition 1. (Input permutation) *Two functions f_1, f_2 are said to P equivalent iff there exists a permutation π, such that $f_1(x_{\pi(1)}, ..., x_{\pi(n)}) = f_2(x_1, ..., x_n)$. $P(f)$ represents the set of all functions which are P equivalent to f.*

SRAM-based logic blocks are universal since they can implement any function by simply loading the truth table of the function into the SRAM table. The new goal is to compact the SRAMs found in logic blocks. The authors of [17] employ a NOT and OR function folding method to compact SRAMs found in logic blocks. Their folding method works as follow, the example taken from the original paper. Let $f(a, b, c)$ and $g(a, b, c)$ be two logic functions whose truth tables are 00011110 and 10001001, respectively. The truth table of function f is divided into two parts which are 0001 and 1110. The entire truth table of function f is constructed from one part as $f = a' \cdot f(0, b, c) + a \cdot \overline{f(0, b, c)}$. Because of this relation, we can keep only half of the truth table and derive the other half from the first part by simply taking NOT of it. For function g, there is no such NOT relation. However, there is an OR relation with 0001 part of f. That is, the 1001 part of g is derived from the bit-wise logical ORing of 1000 of g and 0001 of f. As a result,

$$g(a, b, c) = a' \cdot g(0, b, c) + a \cdot \overline{\{g(0, b, c) + f(0, b, c)\}}$$

From this relation, we notice that to realize functions f and g we only need to store halves of the truth tables of f and g. It is shown that their OR and NOT folding methods reduce the memory requirement of full adders, comparators, and equality checkers. Note that their method eliminates some part of SRAMs at the expense of addition of some extra circuity as shown in their paper. Our method of reducing SRAMs in logic blocks is different from theirs and is inspired from the previous universal logic module research. In our approach, we identify *P* equivalent LUTs and map two P-equivalent LUTs into the same SRAM table in a cluster.

4 Cluster CLB with Shared SRAM Tables

The proposed logic module allows sharing of SRAM tables within clustered CLBs. The sharing concept is illustrated in Figure 3. In this figure, one SRAM table is shared by two basic logic elements. The SRAM table implements two functions at the same time: f and g, where $f(y_{\pi(0)}, ..., y_{\pi(3)}) = g(x_0, ..., x_3)$. With this sharing, instead of having one SRAM table per BLE, we only need one SRAM table per two BLEs. The idea of sharing can be extended to allow the sharing of one SRAM table among K BLEs.

The transistor counts of old and new CLB blocks are tabulated below. We assume that $Muxes$ are implemented with nMOS pass-transistor logic, each SRAM cell requires 6 transistors, and the flip-flop is an edge-triggered D flip-flop [25]. With these implementations our new CLB requires 96 less transistors than the original CLB.

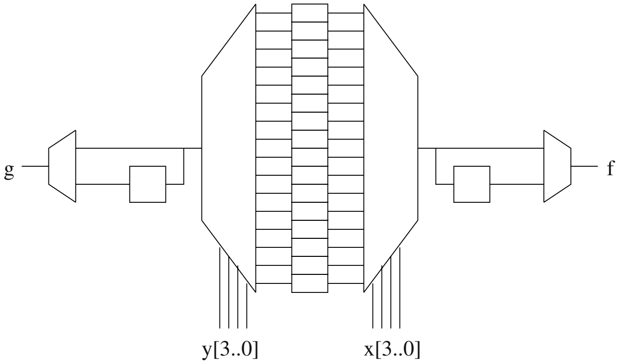
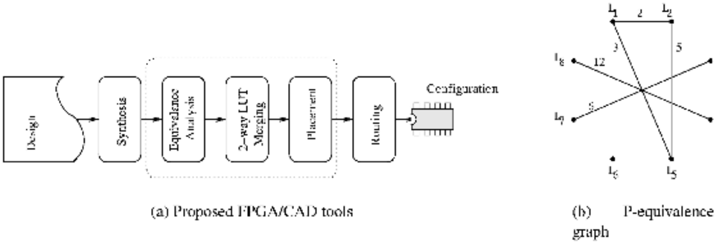


Fig. 3. 2 BLEs sharing one SRAM table



(a) Proposed FPGA/CAD tools

(b) P-equivalence graph

Fig. 4. Proposed FPGA/CAD tools

	$Mux_{16\times1}$	$Mux_{2\times1}$	D	$SRAM_{16}$	$CLB_{original}$	CLB_{new}	$Savings$
#Transistors	16	2	22	96	272	176	96

5 CAD Tool for New FPGA Architecture

We explain our approach and tools assuming that the target FPGA architecture has 2 BLEs that share the SRAM table in a CLB. Each cluster has 2×4 inputs and 2 outputs. Our new tools and their positions in the general FPGA/CAD flow are shown in Figure 4(a). In a synthesized circuit, we identify P-equivalent classes of LUTs in our Equivalence Analysis tool, decide which LUTs to pack together in 2-way LUT Merging, and perform placement of packed CLBs in our Placement tool. Although the LUT Merging and Placement tasks are shown as discrete steps, in our implementation, they are carried out simultaneously.

Equivalency Analysis Tool: After synthesis, an equivalency analysis needs to be performed to determine which LUTs are P-equivalent and could potentially be merged together. Our tool reads in a LUT-level circuit to gather all the necessary information about the LUTs, (these are the literal names and truth tables of the functions). It then creates a graph $G =< V, E >$ where vertex $v_i \in V$ corresponds to LUT L_i in the circuit. If L_i is P-equivalent L_j, then there is an edge between L_i and L_j, and the edge is assigned some non-zero value based on which permutation is required for L_i and L_j. Figure 4(b) illustrates such a graph. For n input LUTs, there are $n!$ possible permutations to check at most; however, since we only use LUTs with four inputs, there are only 24 possible permutations to check, so a brute force approach is used to determine if two LUTs are P-equivalent.

2-Way LUT Merging and Placement: After creating a graph of equivalent LUTs, the next step is to determine which 2 LUTs to actually merge together. For a group of m LUTs that are P-equivalent, there are $\frac{m!}{(2^{m/2} \times (m/2)!)}$ (if m is even) or $\frac{(m+1)!}{2^{(m+1)/2} \times ((m+1)/2)!}$ (if m is odd) different ways to pair the LUTs together.

This obviously grows very fast, so a brute force approach would probably not be the best way to determine which equivalent LUTs to pair together. Since each CLB has two 4-input LUTs and eight inputs, there are no input limitations on which equivalent LUTs could be merged together. Because of this, the choice of which LUTs to merge would have no effect on the total number of CLBs required. The choice of which LUTs to merge, though, would have an effect on routing, such as channel width and delay; however, placement also has a large effect on routing.

We devise a simulated annealing algorithm to optimize both which LUTs to merge in each CLB and the placement of these CLBs. The Simulated Annealing algorithm is an optimization approach based on the way metals cool and freeze in a minimum energy structure[18]. The first step is to create an objective function that determines how good a potential solution is. For our study, the overall goal is to minimize the routing costs. VPR is used to find the best routing for a given circuit, yielding the required channel width and the delay. Because the routing takes quite a bit of time (several minutes), the actual routing is too time consuming to be included as part of the simulated annealing process. Instead, our algorithm attempts to minimize the length of all the connections between each of the LUTs, given which LUTs are merged in each CLB and the placement of each CLB. Our objective function is simply the length of all the wires, both between the LUTs, and between the LUTs and the IO pads. This program generated a file readable by VPR for routing. We formulate the objective function (C) in Eq. 3. Let L_n be the n^{th} LUT, O_n be the output of L_n and I_n be the input set of L_n. Also, let S_m be the m^{th} I/O CLB or CLB. Eq. 1 computes the distance between two CLBs from their physical coordinates. Eq. 2 measures the distance between I/O cell of O_n and L_n, if O_n is not an output of the design, then $outdist(O_n, L_n)$ returns 0.

$$dist(CLB_{x_1,y_1}, CLB_{x_2,y_2}) = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

$$outdist(L_n) = \left\{ \begin{matrix} dist(O_n, L_n) & O_n\ goes\ I/O\ CLB \\ 0 & O_n\ goes\ to\ another\ LUT \end{matrix} \right\} \tag{2}$$

$$C = min \sum_{n=1}^{|L|} \sum_{m=1}^{|I_n|} dist(S_m, L_n) + \sum_{n=1}^{|L|} outdist(L_n) \tag{3}$$

Once the objective function has been chosen, the simulated annealing algorithm is ready to proceed. This algorithm is described in Algorithm 1. The first step is to randomly create a potential solution. For our study, the algorithm would create a potential solution by randomly picking which equivalent LUTs would be merged together in each CLB and would also randomly pick the placement for these CLBs on the FPGA. The cost of this solution is then computed, storing off the result as the best solution seen so far. The following steps can be repeated as often as necessary. Our algorithm stopped after running for a given number of iterations so that it would stop after a reasonable amount of time, usually several minutes, but it could have also stopped after the best solution seemed to converge.

For each iteration, the current solution is slightly mutated, yielding a slightly different solution. For our study, the algorithm modified the current solution by either swapping the location of two IO pads, swapping the location of two CLBs, or swapping a pair of LUTs that were merged. The algorithm determined which of these three mutation methods to use probabilistically, with the probabilities of each being chosen set ahead of time. 40% of the time, the placement of two CLBs was swapped. 45% of the time, the choice of merging two equivalent LUTs was swapped. 7% of the time, the placement of two inputs was swapped, and 8% of the time, the placement of two outputs was swapped. These values were determined experimentally and were found to give good results. The algorithm then computes the cost of this mutated solution and tracks if this new solution is the best solution seen so far. If this new solution is better than the previous current solution, i.e. $\varDelta C < 0$ where C stands for cost, the swap is accepted, and the current solution is set to this new solution, ending the current iteration.

Algorithm 1 LUT merging and placement

```
$Current \leftarrow CurrentInitialSolution()$
$Best \leftarrow Current$
while StoppingCriterion == FALSE do
    while Swap not accepted do
        $Temp \leftarrow Swap(Current)$
        if $Temp < Best$ then
            $Best \leftarrow Temp$
        end if
        if $\varDelta C < 0$ then
            Accept swap
            $Current \leftarrow Temp$
        else
            $P \leftarrow Random(0,1)$
            if $e^{-\varDelta C \times K/T} > P$ then
                Accept swap
                $Current \leftarrow Temp$
            end if
        end if
    end while
end while
$Return\ Best$
```

If this new solution is not as good as the current solution, i.e. $\varDelta C > 0$, the current solution could still potentially be set to this new candidate. This is where the benefit of the simulated annealing approach comes in. Since a worse candidate solution can be

Table 1. Benchmarks descriptions with LUT permutation equivalence classes and their average sizes

Circuits	#Inputs	#Outputs	#LUTs	2-inp	2-inp	3-inp	3-inp	4-inp	4-inp	ave.	ave.	ave.
				#LUTs	#Cls	#LUTs	#Cls	#LUTs	#Cls	2-LUT	3-LUT	4-LUT
alu4	14	8	1522	121	6	446	19	955	43	20.17	23.47	22.21
apex2	39	3	1878	117	4	589	10	1172	29	29.25	58.90	40.41
apex4	9	19	1262	23	4	538	9	700	13	5.75	59.78	53.85
ex1010	10	10	4598	66	6	502	19	829	61	11.00	26.42	13.59
ex5p	8	63	1064	129	7	584	26	1037	60	18.43	22.46	17.28
misex3	14	14	1397	53	6	892	11	2745	47	8.83	81.09	58.40
pdc	16	40	4575	190	1	1944	5	2464	10	190.00	388.80	246.40
seq	41	35	1750	84	5	979	20	3512	51	16.80	48.95	68.86
spla	16	46	3690	45	6	227	9	792	21	7.50	25.22	37.71

accepted, the algorithm would not necessarily get stuck in local minima, which is what happens when using hill-climbing optimization approaches. For our implementation, a swap was accepted if it was close enough, within a percentage randomly chosen at each iteration, to the best solution seen so far, as described below:

Select P as a random number in the range (0,1), with K some constant. If $e^{-\Delta C \times K/T} > P$, then new solution is close enough, and the swap is accepted.

Most simulated annealing approaches include the concept of temperature. In the beginning, the temperature is set very high; meaning the probability of a worse candidate being accepted is large. As the algorithm proceeds, the temperature systematically lowers; meaning the probability of a worse candidate being accepted lessens. A given number of iterations are usually run for each temperature. For simplicity, our implementation just used the best cost seen so far as the temperature. Since the best cost lowers as the algorithm proceeds, this has the same effect as manually lowering the temperature at a predetermined rate. In the beginning, the iterations proceed extremely fast, but as the algorithm proceeds, the amount of time required for each iteration grows at a non-linear rate.

6 Experimental Results

In this section, first, we provide the P-equivalence analysis results of the LUT-level benchmarks described in Table 1 to backup our motivation for designing new logic blocks that allow sharing of SRAM tables. Second, for the same benchmarks, we assessed the performance of our new logic block in terms of area, delay, etc., and also compared it with the performance of the equivalent logic block that does not allow sharing. The benchmarks chosen were the sample circuits available with the VPR package. They also seemed to be commonly used in other FPGA studies in the literature. Out of the 20 available circuits from the VPR package, we have so far only used the 9 combinational circuits and ignored the 11 sequential circuits. This was done only for simplicity in writing our tools; the same results should also hold for sequential circuits since we did nothing to change the behavior of the flip-flops in the CLBs.

Table 2. 2 BLE/CLB results

	Channel			Logic delay	Net delay	Critical path	Original
Circuit	width	#CLBs	#CLB/row	$(\times10^{-9}\ sec)$	$(\times10^{-8}\ sec)$	$(\times10^{-8}\ sec)$	#CLBs
alu4	17	761	28	4.05	7.10	7.51	761
apex2	20	939	31	5.14	7.86	8.37	939
apex4	21	631	26	4.05	7.26	7.66	631
misex3	19	2299	48	4.05	14.7	15.1	2299
seq	21	532	24	4.05	6.72	7.12	532
spla	19	699	27	4.05	7.34	7.74	699
ex1010	27	2288	48	5.14	15.9	16.4	2288
pdc	20	875	30	3.50	7.70	8.05	875
ex5p	28	1845	43	4.05	14.4	14.8	1845

6.1 P-equivalence Analysis of LUTs in Benchmarks

Our Equivalence Analysis tool performed P-equivalence analysis of the benchmarks as shown in Table 1. For each benchmark, the average size of each P-equivalence class was quite large. Table 1 tabulates the number of 2-input, 3-input, and 4-input LUTs in each benchmark. For example, alu4 has 121 2-input, 446 3-input, and 955 4-input LUTs. For each input size, the number of P-equivalent classes (#Cls) and the average number of LUTs (ave. LUT) in a class are also tabulated. For example, alu4 has 6 permutation equivalent classes of 2-input LUTs and the average size of each class of this type LUTs is 20.17. It means there are 20.17 2-input LUTs per class on average.

6.2 Performance of New Logic Block

We performed two sets of experiments and collected statistics about the number of required CLBs, the channel width, logic delay, net delay, and critical path of each benchmark. In the first set, we mapped the circuits onto an FPGA with 2 BLEs per CLB. Tv-pack packed the CLBs, and VPR performed the placement and routing steps. In the second set, the circuits were again mapped onto an FPGA with 2 BLEs per CLB, but this time our tool picked two P-equivalent LUTs, included them in the same CLB, and also performed placement. Also, for the latter set, VPR routed the placed benchmarks. We then compared the area and routing requirements of the proposed and traditional CLBs. The results for channel width, logic delay, net delay, and critical path were all obtained from VPR.

The results of the first and second experiments are tabulated in Tables 2 and 3, respectively. In some benchmarks, there were slight increases in the channel width. The reason for the increase is because only equivalent LUTs could be placed in the same cluster. If synthesis was done to maximize the number of Equivalent LUTs, or perhaps if NPN equivalence was used instead of just P equivalence, more LUTs would be equivalent, so it would converge on the same channel width as the original method. In addition, there was also a slight increase in the critical path for some of the benchmarks. However, we believe that future improvements, such as fine tuning the simulation annealing algorithm to also include the critical path when calculating the objective function, would eliminate this increase in critical path delay. Finally, we measured the area saving of our proposed logic block for each benchmark in terms of transistors. Table 3 also tabulates the area savings by comparing only the utilized CLBs. For example, alu4 requires 761 original CLBs versus 782 new CLBs. With the new CLBs, a saving of 69360 transistors was achieved. Note, however, that this only takes into consideration the transitors for the actual BLEs. It does not take into consideration the size of the multiplexers for the internal connections of the CLBs, which is dependent on the number of BLEs per CLB, as well as the number of internal connections allowed.

Table 3. Shared: 2 BLE/CLB results

	Channel			Logic delay	Net delay	Critical path	Shared	Transistor
Circuit	width	#CLBs	#CLB/row	$(\times 10^{-9}\ sec)$	$(\times 10^{-8}\ sec)$	$(\times 10^{-8}\ sec)$	#CLBs	Savings
alu4	19	782	28	4.60	6.99	7.45	782	69360
apex2	23	952	31	5.14	7.97	8.49	952	87856
apex4	22	638	26	4.05	8.19	8.60	638	59344
misex3	21	2303	48	5.14	14.4	14.9	2303	220000
seq	22	543	24	4.05	8.14	8.55	543	49136
spla	22	728	27	4.05	7.56	7.96	728	62000
ex1010	30	2311	49	5.14	15.4	15.9	2311	215600
pdc	23	901	31	4.60	9.56	10.0	901	79424
ex5p	32	1864	44	5.14	15.0	15.5	1864	173776

Table 4. FPGA area comparison

	Original	Shared	Extra	Transistor	Extra Wires	
Circuit	FPGA	FPGA	#CLB	Saving	units in # CLBs	WL_{new}/WL_{orig}
alu4	28×28	28×28	0	75264	3248	0.899
apex2	31×31	31×31	0	92256	5952	0.925
apex4	26×26	26×26	0	64896	1404	0.842
misex3	48×48	48×48	0	221184	9408	0.889
seq	24×24	24×24	0	55296	1200	0.842
spla	27×27	27×27	0	69984	4536	0.931
ex1010	48×48	49×49	97	204112	19992	0.930
pdc	30×30	31×31	61	75664	8432	0.986
ex5p	43×43	44×44	87	162192	20768	0.962

Table 4 tabulates the area savings in terms of number of transistors by comparing the target $n \times n$ FPGAs. For example, the original target FPGA architecture of ex1010 has a total of 48×48 CLBs, but only 2288 of them are utilized. For the same benchmark, the new target FPGA architecture has a total of 49×49 new CLBs, but only 2311 of them are utilized. Although ex1010 requires 97 extra new CLBs, still the new FPGA has 204112 less transistors than the corresponding original FPGA. Moreover, we computed the wire overhead of our approach. If we assume that the dimensions of the new and old CLBs are equal, the number of extra wires used in the new FPGA is given under the Extra Wires column. However, it is clear that due to area reduction of our new CLB, the dimensions of the new CLB would be lower than those of the original CLB. We used Eq. 5 to estimate the ratio of new wirelength over old wirelength, which is 0.804. Also, we give the ratio of the overall wirelengths for the new and the old architectures (WL_{new}/WL_{orig}). The total wirelength in terms of the number of CLBs crossed can be computed by the following formula:

$$Wirelength = 2 \times (\#rows + 1) \times (\#cols) \times ChannelWidth \tag{4}$$

For the wirelength with the shared CLBs, each CLB is only 0.804 as large as the original CLBs, so the new wirelength is scaled down by a factor of 0.804. As can be seen in the last column of the table, these ratios are less than 1 in all benchmarks.

This guaranties that there will be reductions not only in CLB area but also in the total wirelength. Again, this does not take into account the multiplexers for the internal CLB connections.

$$\frac{WL_{new}}{WL_{orig}} = \sqrt{\frac{176}{272}} = 0.804 \tag{5}$$

7 Conclusions and Future Work

In this paper, we proposed a new clustered logic block for SRAM-based FPGAs that allow sharing of SRAM tables among basic logic elements (BLEs). Our analysis results indicate that there is a great potential for saving from logic block area when the proposed logic blocks and tools are used. With the proposed architecture, around half of the SRAM tables can be eliminated from the existing FPGA architectures when we merged only 2 LUTs. In case of merging more than two LUTs, more than half of the SRAMs tables would be eliminated. As a result, significant area reduction will reflect to the costs of FPGAs and will make the proposed architecture an essential choice for applications that demand very low area. In the light of these results, we will continue this research to better understand pros and cons of SRAM table sharing in the clustered logic blocks.

As future work, we will physically design the new FPGA architecture and assess its performance at the layout level. Also, we will investigate new FPGA architectures which can efficiently utilize new concept of LUT sharing. We plan to study how sharing SRAM tables is affected by the number of inputs to each cluster as well as the number of CLBs in each cluster. Another important area to consider is the tradeoff in terms of power consumption with the new shared architecture.

References

1. O. Agrawal, H. Chang, B. Sharpe-Geisler, N. Schmitz, B. Nguyen, J. Wong, G. Tran, F. Fontana, and B. Harding. An innovative, segmented high performance fpga family with variable-grain-architecture and wide-gating functions. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 17–26, Monterey, California, 1999.
2. E. Ahmed and J. Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 3–12, Monterey, California, 2000.
3. Altera. *Programmable Logic Data Book*. 1996.
4. V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In W. Luk, P. Y. Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Springer-Verlag, Berlin, 1997.
5. V. Betz and J. Rose. How much logic should go in an fpga logic block? *IEEE Design and Test of Computers*, 15(1):10–15, 1998.
6. C. chang Lin, M. Marek-Sadowska, and D. Gatlin. Universal logic gate for fpga design. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 164–168. IEEE Computer Society Press, 1994.
7. Y. Y. Chung and N. M. Bergmann. Video compression on fpga-based custom computers. In *International Conference on Image Processing*, volume 1, pages 361–364, 26-29 October 1997.

8. K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, 2002.
9. J. Cong, J. Peck, and V. Ding. Rasp: a general logic synthesis system for sram-based fpgas. In *Proc. of ACM/SIGDA Int'l Symp. on FPGAs*, pages 137–143, 1996.
10. E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.
11. A. J. Elbirt and C. Paar. An fpga implementation and performance evaluation of the serpent block chiper. In *Proc. of ACM/SIGDA Int'l Symp. on FPGAs*, pages 33–40, 2000.
12. R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of Design, Automation and Test in Europe*, pages 642–649, 2001.
13. J. R. Hauser and J. Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 12–21, 1997.
14. J. Heron, D. Trainor, and R. Woods. Image compression algorithms using re-configurable logic. In *Conference Record of the Thirty-First Asilomar Conference on Signals,Systems and Computers*, volume 1, pages 399–403, 2-5 November 1997.
15. W.-J. Huang, N. Saxena, and E. J. McCluskey. A reliable lz data compressor on reconfigurable coprocessors. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 249–258, April 2000.
16. A. Kaviani and S. Brown. Hybrid fpga architecture. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 3–9, Monterey, California, 1996.
17. S. Kimura, T. Horiyama, M. Nakanishi, and H. Kajihara. Folding of logic functions and its application to look up table compaction. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 694–697. ACM Press, 2002.
18. S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
19. G. Lemieux and D. Lewis. Using sparse crossbars within lut clusters. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 59–68, Monterey, California, 2001.
20. K. H. Leung, K. W. Ma, W. K. Wong, and P. H. W. Leong. Fpga implementation of a microcoded elliptic curve cryptographic processor. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 68–76, 2000.
21. J. Ritter and P. Molitor. A partitioned wavelet-based approach for image compression using fpga's. In *Proc. of IEEE Custom Integrated Circuits Conf.*, pages 547–550, 21-24 May 2000.
22. S. Thakur and D. F. Wong. On designing ulm-based fpga logic modules. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 3–9. ACM Press, 1995.
23. S. Trimberger, K. Duong, and B. Conn. Architecture issues and solutions for a high-capacity fpga. In *Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*, pages 3–9, Monterey, California, 1997.
24. J. Villasenor and W. H. Mangione-Smith. Configurable computing. *Scientific American*.
25. N. H. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison Wesley, 1995.
26. XILINX. *FPGA Data Book*. 1996.
27. A. Yan, R. Cheng, and S. J. Wilton. On the sensitivity of fpga architectural conclusions to experimental assumptions, tools, and techniques. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 147–156, Monterey, California, 2002.

A Modular System for FPGA-Based TCP Flow Processing in High-Speed Networks*

David V. Schuehler and John W. Lockwood

Applied Research Laboratory, Washington University
One Brookings Drive, Campus Box 1045
St. Louis, MO 63130-4899 USA
{dvs1, lockwood}@arl.wustl.edu
http://www.arl.wustl.edu/projects/fpx/reconfig.htm

Abstract. Field Programmable Gate Arrays (FPGAs) can be used in Intrusion Prevention Systems (IPS) to inspect application data contained within network flows. An IPS operating on high-speed network traffic can be used to stop the propagation of Internet worms and to protect networks from Denial of Services (DoS) attacks. When used in the backbone of a core network, the device will be exposed to millions of active flows simultaneously. In order to protect the data in each connection, network devices will need to track the state of every flow. This must be done at multi-gigabit line rates without introducing significant delays.
This paper describes a high performance TCP processing system called TCP-Processor which supports flow processing in high-speed networks utilizing multiple devices. This circuit provides stateful flow tracking, TCP stream reassembly, context storage, and flow manipulation services for applications which process TCP data streams. A simple client interface eases the complexities associated with processing TCP data streams. In addition, a set of encoding and decoding circuits has been developed which efficiently transports this interface between multiple FPGA devices. The circuit has been implemented in FPGA hardware and tested using live Internet traffic.

1 Introduction

Including reconfigurable networking technology within the core of the Internet offers enhanced levels of service to users of the network. New types of data processing services can be applied to either all traffic traversing the network, or to just a few selected flows.

This paper presents a modular circuit design of a content processing system implemented in FPGA hardware. A circuit has been built that reassembles TCP/IP data packets into their respective byte streams at multi-gigabit line rates. The implementation contains a large per-flow state store which maintains

* This work was supported by a grant from Global Velocity. The authors of this paper have received equity from the the license of technology to that company. John Lockwood has served as a consultant and co-founder to Global Velocity.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 301–310, 2004.

64 bytes of state information per active flow and supports 8 million bidirectional TCP flows concurrently.

The technology described in this paper supports the coupling of other FPGA-based data processing circuits in order to develop larger, more complex processing systems. This technology enables a new generation of network services to operate within the core of the Internet.

The remainder of this paper is divided into the following sections. Section 2 provides motivation for this work. Section 3 describes related work on high-performance processing systems. Section 4 describes the design of the system. Section 5 describes current results. Section 6 outlines future work and section 7 provides concluding statements.

2 Motivation

Over 85% of all traffic on the Internet today uses the TCP/IP protocol. TCP is a stream-oriented protocol providing guaranteed delivery and ordered byte flow services. Processing of TCP data flows at a location in the middle of the network is extremely difficult. Along the way, packets can be dropped, duplicated and re-ordered. Packet sequences observed within the interior of the network can be different from packets received and processed at the connection endpoints. The complexities associated with tracking the state of end systems and reconstructing byte sequences based on observed traffic are significant [2].

Many types of network services require access to the TCP stream data traversing high-speed networks. These services may include those which detect and prevent the spread of Internet worms/viruses, those that detect and remove spam, those that perform content-based routing operations, and those that secure data. The TCP processing circuit described in this paper enables these complex network services to operate at gigabit speed by providing an environment for processing TCP stream data in hardware.

A vast number of end systems communicate over the Internet. This traffic is concentrated to flow over a relatively small number of routers which forward traffic through the core of the Internet. Currently, Internet backbones operate over communication links ranging in speed from OC-3 (155 Mbps) to OC-768 (40 Gbps) rates. Table 1 illustrates a breakdown of common communication links, their corresponding data rates, and the rate at which packets of different sizes can be transmitted over those links. It is important to note that with faster links processing smaller packets, circuits must be able to process millions of TCP packets per second. Other existing TCP processing circuits are unable to operate at high bandwidth rates and manage millions of active flows. Instead, these other network monitors typically operate on end systems or in local area networks where the overall bandwidth and/or the total number of flows to be processed is low.

The TCP processing circuit described in this paper handles the complexities associated with flow classification and TCP stream reassembly. It exposes network traffic flow data through a simple client interface. This enables other

Table 1. Optical links and associated data rates

Link type	Data rate	40 byte pkts/sec	64 byte pkts/sec	500 byte pkts/sec	1500 byte pkts/sec
OC-3	155 Mbps	.48 M	.3 M	38 K	12 K
OC-12	622 Mbps	1.9 M	1.2 M	.16 M	52 K
GigE	1.0 Gbps	3.1 M	2.0 M	.25 M	83 K
OC-48	2.5 Gbps	7.8 M	4.8 M	.63 M	.21 M
OC-192/10 GigE	10 Gbps	31 M	20 M	2.5 M	.83 M
OC-768	40 Gbps	130 M	78 M	10 M	3.3 M

high-performance data processing sub-systems to operate on network content without having to perform complex protocol processing operations.

3 Related Work

The majority of existing packet capturing and monitoring systems are software-based and have severe performance limitations that prevent them from effectively monitoring high speed networks. Software-based solutions like Bro [11], Snort [12], and WebSTAT [15] perform analysis of TCP flows, but are limited to processing data at rates less than 100Mbps.

FPGA-based network processing systems can process network traffic at much higher data rates. A hardware circuit developed at Georgia Tech called TCP-Stream Reassembly and State Tracking can analyze a single TCP flow at 3.2Gbps [10]. The circuit was tested using a FPGA device which tracks the state of a TCP connection and performs limited buffer reassembly. Because the circuit operates only on a single flow, additional copies of the circuit need to be instantiated in order to process multiple flows simultaneously. Using this scheme, a maximum of 30 flows can be processed with a Xilinx Virtex 2000E FPGA device.

Another FPGA-based approach to TCP processing was undertaken at the University of Oslo [7]. This design provides TCP connection state processing and TCP stream reassembly functions for both client and server directed traffic on a single TCP connection. A 1024-byte reassembly buffer is maintained for both client-side and server-side traffic. Packets lying outside of the reassembly buffer space on receipt are dropped. Portions of the circuit design have been implemented and are able to process data at 3.06Gbps. A separate instance of this circuit is required to process each individual TCP connection. A Xilinx Virtex 1000E FPGA can support a maximum of 8 TCP flows. Thus for hardware, existing systems are severely limited in the number of flows that they can process. Additionally, they provide no simple mechanisms to support interfacing with other types of data processing sub-systems.

Hardware sub-systems have been developed which match patterns in data streams using deterministic finite automata (DFA) and nondeterministic finite automata (NFA) circuits. Sidhu and Prasanna implemented a regular expression matching engine in FPGA logic that constructs NFAs [14]. Franklin *et al.*

extended this work by creating FPGA-based regular expressions corresponding to the Snort rule database [6]. Moscola *et al.* developed a DFA-based approach to regular expression processing in hardware [9]. Although the time and space requirements associated with constructing DFAs can be exponentially long in the worst case, the authors show that such scenarios rarely occur and that their DFA implementation on average contains fewer states than a NFA implementation. Other work has led to the development of content matching systems implemented with Bloom filters that can scan for very large numbers of strings [5]. Circuits have been developed which use multiple hashes and probabilistic matching in order to scan for 35,475 unique signatures on a Virtex 2000E [1].

All of these types of FPGA-based data processing circuits are prime candidates for integration with the modular TCP processing system outlined in this paper.

4 Design

The TCP flow processing architecture described in [13] has been implemented in FPGA logic. It is capable of monitoring 8 million bidirectional TCP flows on an OC-48 (2.5 Gbps) network link. This circuit provides a simple client interface enabling other FPGA circuits to easily process TCP data streams. Network data packets are annotated with additional control signals which provide information about which data bytes correspond to the IP header, the TCP header, and the TCP payload section. There are also signals that indicate which TCP data bytes are part of a consistent stream of data and which bytes should be ignored because they are retransmissions. Signals which indicate the beginning and end of flow are included along with a unique flow identifier so that the client can independently manage per-flow context.

The architecture for the TCP-Processor consists of six distinct components. These include an *input buffer*, a *TCP processing engine*, a *state store manager*, a *routing module*, an *egress module*, and a *statistics module*. The layout of the components along with data flow and component interactions can be seen in Figure 1. There are two additional components, an *encode* module and a *decode* module, which encode and decode data for transport between multiple FPGA devices. The main flow of data traverses the circuit following the path indicated by the bold/white arrows. The *state store manager* stores and retrieves per-flow context information from off-chip memory. The dotted line through the middle of the circuit indicates that the ingress and egress portions of the TCP processing can be separated from each other and placed on different devices.

Data enters the engine as IP data packets from the Layered Protocol Wrappers [3,4] via a 32-bit wide data bus. Packets are initially processed by the *input buffer* component which buffers packets if there are any downstream processing delays. These delays can be caused by client applications which can induce backpressure into the system by sending a flow control signal.

Data from the *input buffer* flows into the *TCP processing engine*. Packets are classified and associated with the appropriate flow context information retrieved

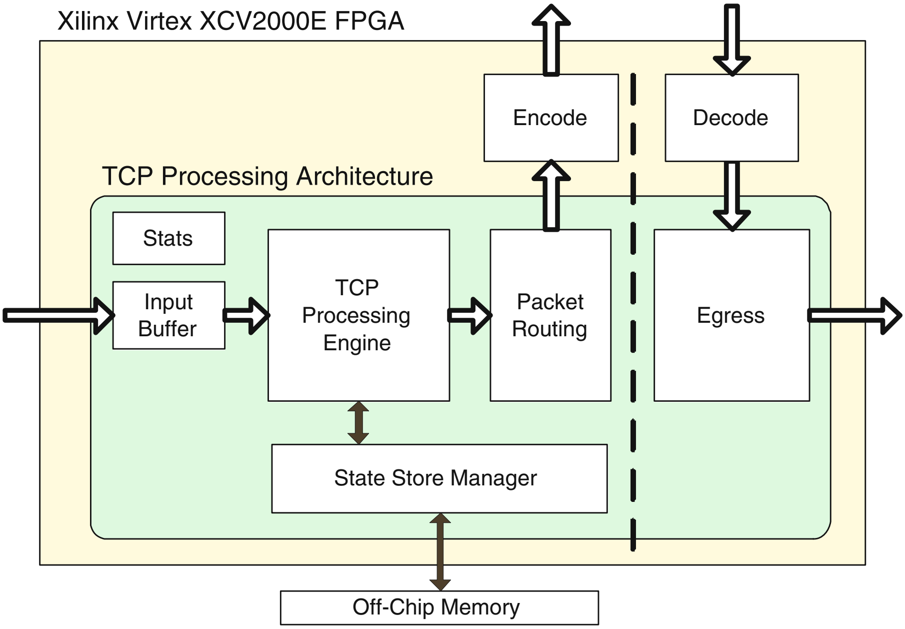


Fig. 1. Layout of TCP-Processor

via the *state store manager*. The TCP checksum is validated and other TCP-specific processing occurs at this time. New flow context information is written back to the *state store manager* and the packet is passed on to the *packet routing module*.

The *packet routing module* provides the client interface. Figure 2 shows the timing diagram of the transmission of a single TCP data packet to the monitoring application. The DATA and FLOWSTATE busses contain an indication of the data that would be present at each clock cycle during normal processing. The components of the IP and TCP headers are shown with the control signals. Signals are provided which indicate the start of frame (SOF), start of IP header (SOIP), and the start of IP payload (SOP). The TCP data enable signal (TDE) indicates that there is TCP stream data on the DATA bus and the BYTES vector identifies which of the four bytes on the DATA bus contain in-order TCP stream data. In this example, the NEWFLOW signal indicates that the data contained within this packet represents the first few bytes of a new data stream and that the monitoring application should initialize its processing state prior to processing this data. Additional information can be clocked through the data bus before and/or after the packet data. This allows for lower-level protocols, such as VCIs, Ethernet MAC addresses, shims, or packet routing information to pass through the hardware along with the packet. On the `DATA` bus of Figure 2, the `AAA` and `BBB` data values represent lower layer protocol and control information prior to the start of the IP packet. CCC and DDD represent trailer fields that follow the IP packet. The AAA, BBB, CCC, and DDD content is ignored by the monitoring application but is passed through to the client interface of the TCP-Processor for use in outbound packet processing.

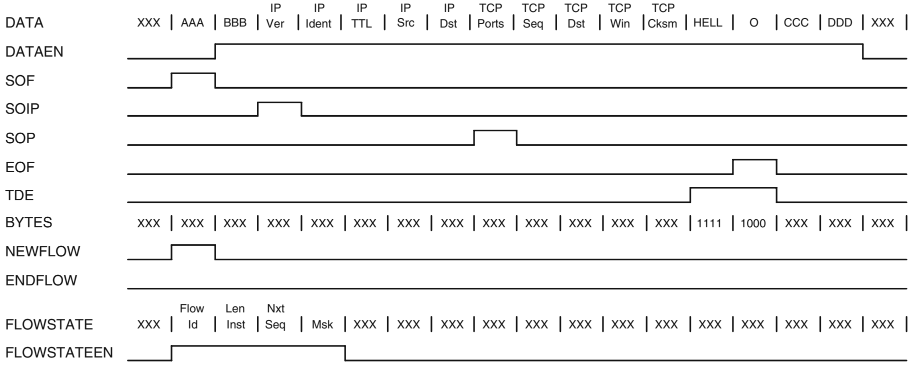


Fig. 2. Timing Diagram showing Client Interface

The *state store manager* manages interactions with a large external memory module to store per-flow context information. By default, 64 bytes of context information is stored for each flow. When the TCP-Processor is configured for bidirectional monitoring, 32 bytes of storage are used for each direction of traffic flow. Because the context information needs to be retrieved and updated when processing each packet, the circuit that implements the *state store manager* is highly optimized.

The *statistics module* collects and maintains event counts from the other components in the system. 28 independent counters collect information that includes the number of packets processed by each module, the number of new flows, the number of terminated flows, the number of active flows, counts of the TCP data bytes processed, counts of the total bytes processed, counts of the out-of-sequence packets and the number of retransmitted packets. This data is accumulated in separate 16, 24 and 32 bit counters, depending on the expected frequency of each event. On a predetermined periodic interval, all of the collected statistics are formatted into a UDP packet and transmitted to an external application where the data can be written to disk or plotted in real-time.

This TCP processing architecture includes a mechanism for efficiently transporting signals from one FPGA device to another, which enables the development of complex intrusion prevention systems implemented on multiple FPGAs. Signals on one device are encoded and transported to a second device where the information is decoded and the original waveform is reproduced. The encoded format uses an 8-bit header containing a 4-bit type and a 4-bit header length. This format is both self describing and extensible and therefore enables additional information to be added to the encoded data as it is passed from device to device. This encoded header information is prepended to the network data packet and sent across the inter-device data path. Older processing circuits can easily skip past new control headers by examining the common header fields and advancing to the next control header. Figure 3 shows the encoding and decoding process.

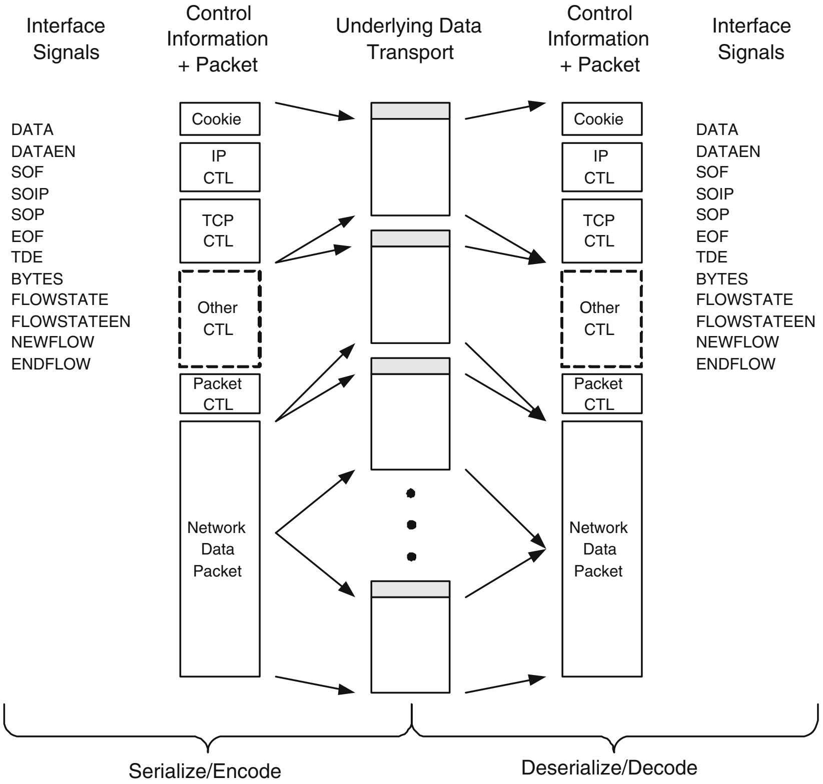


Fig. 3. Encoding signals for TCP/IP flows processed by multiple FPGAs

5 Results

The TCP-Processor circuit has been implemented in FPGA logic. When targeting a Xilinx Virtex XVC2000E-8 FPGA, the TCP-Processor has a post place-and-route frequency of 85.565MHz utilizing 59% (95/160) of the block RAMs and 35% (7279/19200) of the slices. The device is capable of processing 2.9 million 64-byte packets per second. A diagram of the TCP-Processor circuit layout on a Xilinx Virtex 2000E FPGA is shown in Figure 4. The layout is loosely divided into regions which correspond to the various processing functions of the circuit.

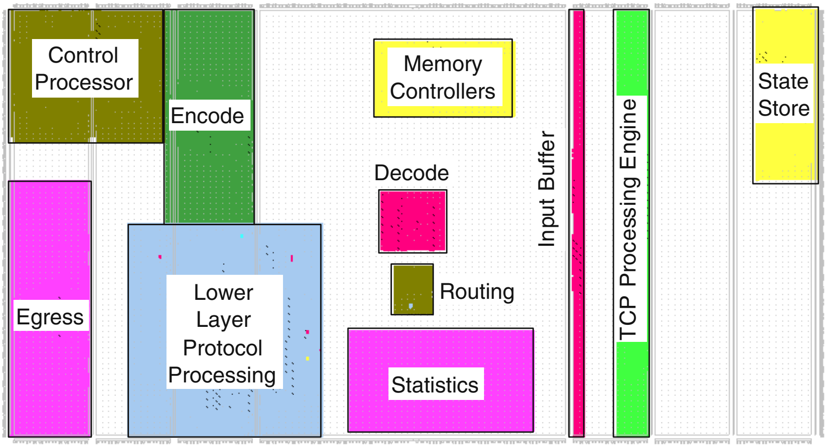


Fig. 4. TCP-Processor circuit layout on Xilinx XCV2000E

The TCP-Processor has been tested on the Field-programmable Port Extender (FPX) platform [8]. The FPX platform contains a Xilinx Virtex 2000E-8 FPGA, two 2.5Gbps interfaces, two 2MB external ZBT SRAM devices and two 512MB external PC100 SDRAM devices. The two network interfaces are positioned such that devices can be stacked on top of one another. Figure 5 shows FPX devices in a stacked configuration. A Gigabit Ethernet line card used to transfer data out of the FPX platform is visible in the foreground.

6 Future Work

By incorporating newer FPGAs, such as the Xilinx Virtex-II Pro, traffic can be processed at OC-192 (10Gbps) data rates using a circuit like the one described in this paper. A few additional mechanisms can be employed to increase the performance of the TCP-Processor for use in OC-768 (40Gbps) networks. In a faster circuit, memory latency could prevent the circuit from fully processing a steady stream of minimum length packets. By instantiating two copies of the *TCP processing engine* and using an *input buffer* to route traffic between the two engines, the memory latency issue can be overcome by threading.

A new extensible networking platform is currently under development at the Washington University Applied Research Laboratory. This platform will incorporate the Xilinx Virtex-II Pro FPGA, Double Data Rate (DDR) memory, a Ternary Content Addressable Memory (TCAM) and an Intel network processor

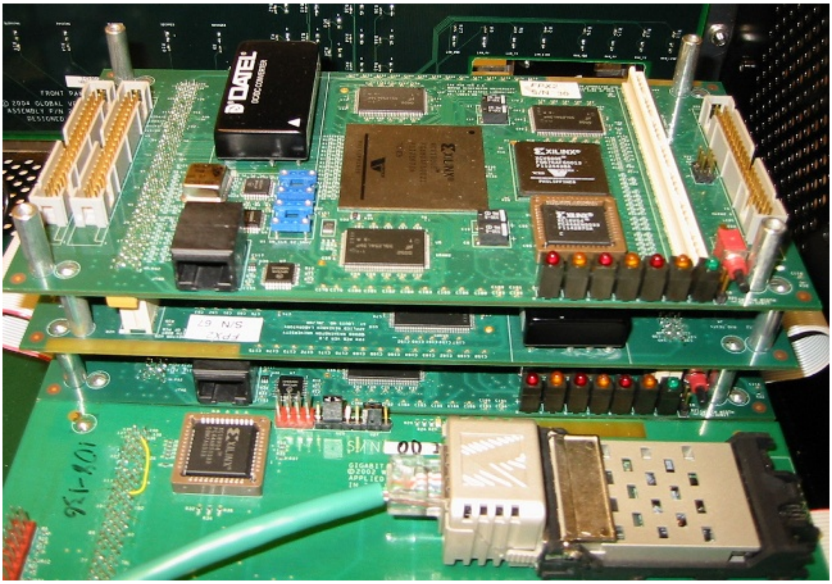


Fig. 5. Stacked FPX devices

to support traffic processing at 10 gigabits per second. We plan to support the TCP-Processor on this new research platform.

7 Conclusion

This paper described the design and implementation of a high-performance TCP flow monitoring system called TCP-Processor for use in an extensible environment. The TCP-Processor was implemented and tested on a Xilinx XCV2000E FPGA utilizing the FPX platform. The circuit operates at 85MHz and is capable of monitoring 8 million bidirectional TCP flows at OC-48 (2.5 Gbps) data rates. This design could be enhanced to monitor higher speed networks by employing parallel processing circuits and using current FPGA and memory devices.

The TCP-Processor provides a simple client interface for monitoring TCP flows which annotates existing network data packets with additional signals. The design of the TCP-Processor is both modular and flexible and can be easily adapted to other extensible networking environments which employ FPGA devices.

References

1. M. Attig, S. Dharmapurikar, and J. Lockwood. Implementation results of bloom filters for string matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2004.

2. K. Bhargavan, S. Chandra, P. J. McCann, and C. A. Gunter. What packets may come: automata for network monitoring. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–219. ACM Press, 2001.
3. F. Braun, J. Lockwood, and M. Waldvogel. Reconfigurable router modules using network protocol wrappers. In *Proceedings of Field-Programmable Logic and Applications*, pages 254–263, Belfast, Northern Ireland, Aug. 2001.
4. F. Braun, J. W. Lockwood, and M. Waldvogel. Layered protocol wrappers for Internet packet processing in reconfigurable hardware. In *Proceedings of Symposium on High Performance Interconnects (HotI'01)*, pages 93–98, Stanford, CA, USA, Aug. 2001.
5. S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of Symposium on High Performance Interconnects (HotI'03)*, pages 25–29, Stanford, CA, USA, Aug. 2003.
6. R. Franklin, D. Carver, and B. L. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, Apr. 2002.
7. S. Li, J. Toressen, and O. Soraasen. Exploiting stateful inspection of network security in reconfigurable hardware. In *Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, Sept. 2003.
8. J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, Feb. 2001.
9. J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, Apr. 2003.
10. M. Necker, D. Contis, and D. Schimmel. TCP-Stream Reassembly and State Tracking in Hardware. FCCM 2002 Poster, Apr 2002.
11. V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
12. M. Roesch. SNORT - Lightweight Intrusion Detection for Networks. In *LISA '99: USENIX 13th Systems Administration Conference*, November 1999.
13. D. V. Schuehler, J. Moscola, and J. Lockwood. Architecture for a hardware based, tcp/ip content scanning system. In *Proceedings of Symposium on High Performance Interconnects (HotI'03)*, pages 89–94, Stanford, CA, USA, Aug. 2003.
14. R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, CA, USA, Apr. 2001.
15. G. Vigna, W. Robertson, V. Kher, and R. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs

Zachary K. Baker and Viktor K. Prasanna

University of Southern California, Los Angeles, CA, USA
zbaker@halcyon.usc.edu, prasanna@ganges.usc.edu

Abstract. This paper presents a tool for automatic synthesis of highly efficient intrusion detection systems using a high-level, graph-based partitioning methodology, and tree-based lookahead architectures. Intrusion detection for network security is a compute-intensive application demanding high system performance. This tool automates the creation of efficient FPGA architectures using system-level optimizations, a relatively unexplored field in this area. The pre-design tool allows for more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance. The tool is available online for public use.

1 Introduction

Pattern matching for network security and intrusion detection demands exceptionally high performance. This performance is dependent on the ability to match against a large set of patterns, and thus the ability to automatically optimize and synthesize large designs is vital to a functional network security solution. Much work has been done in the field of string matching for network security [1,2,3,4,5]. However, the study of the *automatic design* of efficient, flexible, and powerful system architectures is still in its infancy.

Snort, the open-source IDS [6], and Hogwash [7] have thousands of content-based rules. A system based on these rulesets requires a hardware design optimized for thousands of rules, many of which require string matching against the entire data segment of a packet. To support heavy network loads, high performance algorithms are required to prevent the IDS from becoming the network bottleneck. One option is to move the matching away from the processor and on to an FPGA, wherein a designer can take advantage of the reconfigurability of the device to produce customized architectures for each set of rules.

By carefully and intelligently processing an entire ruleset (Figure 1), our tool can *partition* a ruleset into multiple pipelines in order to optimize the area and time characteristics of the system. By applying automated graph theory and trie techniques to the problem, the tool effectively optimizes large ruleset, and then generates a fully synthesizeable architecture description in VHDL ready for

[1] Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by an equipment grant from the HP Corporation.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 311–321, 2004.

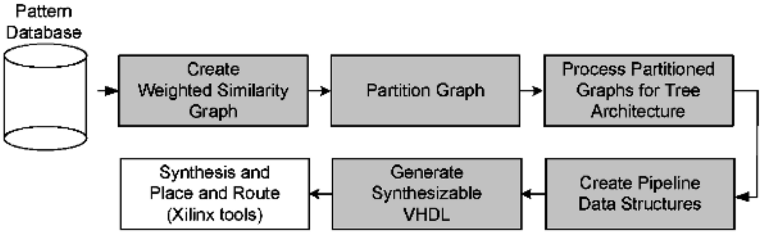

Pattern Database
Create Weighted Similarity Graph
Partition Graph
Process Partitioned Graphs for Tree Architecture
Create Pipeline Data Structures
Generate Synthesizable VHDL
Synthesis and Place and Route (Xilinx tools)

Fig. 1. Automated optimization and synthesis of partitioned system

place and route and deployment in less than 10 seconds for a set of 351 patterns, and less than 30 for 1000 patterns.

2 Related Work in Automated IDS Generation

Snort [6] and Hogwash [7] are current popular options for implementing intrusion detection in software. They are open-source, free tools that promiscuously tap the network and observe all packets. After TCP stream reassembly, the packets are sorted according to various characteristics and, if necessary, are string-matched against rule patterns. However, the rules are searched in software on a general-purpose microprocessor. This means that the IDS is easily overwhelmed by periods of high packet rates. The only option given by the developers to improve performance is to remove rules from the database or allow certain classes of packets to pass through without checking. Some hacker tools even take advantage of this weakness of Snort and attack the IDS itself by sending worst-case packets to the network, causing the IDS to work as slowly as possible. If the IDS allows packets to pass uninspected during overflow, an opportunity for the hacker is created. Clearly, this is not an effective solution for maintaining a robust IDS.

Automated IDS designs have been explored in [1], using automated generation of Non-deterministic Finite Automata. The tool accepts rule strings and then creates pipelined distribution networks to individual state machines by converting template-generated Java to netlists using JHDL. This approach is powerful but performance is reduced by the amount of routing required and the logic complexity required to implement finite automata state machines. The generator can attempt to reduce logic burden by combining common prefixes to form matching trees. This is part of the pre-processing approach we take in this paper.

Another automated hardware approach, in [5], uses more sophisticated algorithmic techniques to develop multi-gigabyte pattern matching tools with full TCP/IP network support. The system demultiplexes a TCP/IP stream into several substreams and spreads the load over several parallel matching units using Deterministic Finite Automata pattern matchers. In their architecture, a web interface allows new patterns to be added, and then the new design is generated and a full place-and-route and reconfiguration is executed, requiring 7-8 minutes.

As their tools have been commercialized in [8], only some of their architectural components are freely available to the community.

The NFA concept is updated with predecoded inputs in [9]. This paper addresses the poor frequency performance as the number of patterns increases, a weakness of earlier work. This paper solves most of these problems by adding predecoded wide parallel inputs to a standard NFA implementations. The result is excellent area and throughput performance.

In [2,3], a hardwired design is developed that provides high area efficiency and high time performance by using replicated hardwired 32-bit comparators in a pipeline structure. The matching technique proposed is to use four 32-bit hardwired comparators, each with the same pattern offset successively by 8 bits, allowing the running time to be reduced by 4x for an equivalent increase in hardware. These designs have adopted some strategies for reducing redundancy through pre-design optimization. Expanding their earlier work, [2] reduces area by finding identical alignments between otherwise unattached patterns. Their preprocessing takes advantage of the shared alignments created when pattern instances are shifted by 1, 2, and 3 bytes to allow for the 32-bit per cycle architecture. The work in [3] shares the pre-decoded shift-and-compare approach with our work, but they utilize SRL16 shift registers where we utilize single-cycle delay flip-flops. Their work also utilizes a partitioning strategy based on incrementally adding elements to partitions to minimize the addition of new characters to a given partition.

3 Our Approach

This research focuses on the automatic optimization and generation of high-performance string-matching systems that can support network traffic rates while providing support for large pattern databases. The tool generates two basic architectures, a pre-decoded shift-and-compare architecture, and a variation using a tree-based area optimization. In this architecture, a character enters the system and is "pre-decoded" into its character equivalent. This simply means that the incoming character is presented to a large array of AND gates with appropriately inverted inputs such that the gate output asserts for a particular character. The outputs of the AND gates are routed through a shift-register structure to provide time delays. The pattern matchers are implemented as another array of AND gates and the appropriate decoded character is selected from each time-delayed shift-register stage. The tree variation is implemented as a group of inputs that are pre-matched in a "prefix lookahead" structure and then fed to the final matcher stage. The main challenge in the tree structure is creating the trees; this is discussed in Section 4.

The notion of predecoding has been explored in [10] in the context of finite automata, the use of large, pipeline brute-force comparators for high speed was initiated by [11] and furthered by [12]. The use of trees for building efficient regular expression state machines was initially developed by [1]. We explored the partitioning of patterns in the pre-decoded domain in [13]. We utilize these foundational works and build automatic optimization tools on top.

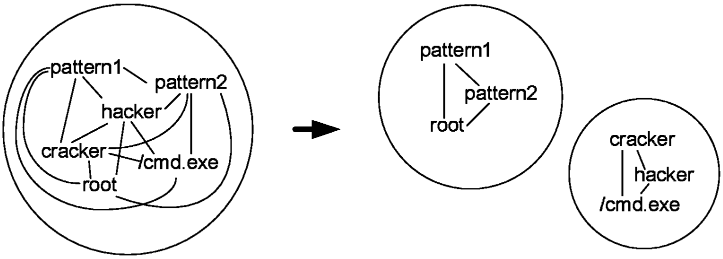


Fig. 2. Partitioned graph; by reducing the cut between the partitions we decrease the number of pipeline registers

With our domain specific analysis and generation tool, extensive automation carefully partitions a rule databases into multiple independent pipelines. This allows a system to more effectively utilize its hardware resources. The key to our performance gains is the idea that characters shared across patterns do not need to be redundantly compared. Redundancy is an important idea throughout string matching; the Knuth-Morris-Pratt algorithm, for instance, uses precomputed redundancy information to prevent needlessly repeating comparisons. We utilize a more dramatic approach; by pushing all character-level comparisons to the beginning of the comparator pipelines , we reduce the character match operation to the inspection of a single bit.

Increasing the number of bits processed at a single comparator unit increases the delay of those gates. The pre-decoding approach moves in the opposite direction, to single-bit, or *unary*, comparisons. We decode an incoming character into a “one-hot” bit vector, in which a character maps to a single bit.

Because intrusion detection requires a mix of case sensitive and insensitive alphabetic characters, numbers, punctuation, and hexadecimal-specified bytes, there is an interesting level of complexity. However, each string only contains a few dozen characters, and those characters tend to repeat across strings.

Using the min-cut heuristic, the patterns are partitioned n-ways such that the number of repeated characters within a partition is maximized, while the number of characters repeated between partitions is minimized. The system is then generated, composed of n pipelines, each with a minimum of bit lines. The value of n is determined from results; we have found $n=$ 2-4 most effective for rulesets of less than 400 patterns. Conversely, for the 603 and 1000 pattern rulesets, the highest time performance is achieved with eight partitions. However, as the area increases moderately as the number of partitions increases, the area-time tradeoff must be considered. The tools, on average, can partition a ruleset to roughly 30 bits, or about the same amount of data routing in the 32 bit designs of [2,3]. The matching units are least 8x smaller (32 down to 4 bits), and we have removed the control logic of KMP-style designs such as [14].

Our unary design utilizes a simple pipeline architecture for placing the appropriate bit lines in time. Because of the small number of total bit lines required (generally around 30) adding delay registers adds little area to the system design.

First, the tool partitions the patterns into several groups (Figure 2). Each group of patterns is handled by an independent pipeline to minimize the length of interconnects. We create a graph representation of the patterns to minimize the number of characters that have to be piped through each pipeline.

The graph creation strategy is as follows. We start with a collection of patterns, represented as nodes of a graph. Each pattern is composed of letters. Every node with a given letter is connected by an edge to every other node with that letter. We formalize this operation as follows:

$$S_k = \{a : a \in C \mid a \text{ appears in } k\} \tag{1}$$

$$V_R = \{p : p \in T\} \tag{2}$$

$$E_R = \{(k,l) : k,l \in T,\ k \neq l \text{ and } S_k \cap S_l \neq \emptyset\} \tag{3}$$

Graph creation before partitioning; a vertex V is added to graph R for each pattern p in the ruleset T and an edge E is added between any vertex-patterns that have a common character in the character class C

This produces a densely connected graph, almost 40,000 edges in a graph containing 361 vertices. Our objective is to partition the big graph into two or more smaller groups such that the number of edges between nodes within the group is maximized, and the number of edges between nodes in different groups is minimized. Each pipeline supplies data for a single group. By maximizing the edges internal to a group and minimizing edges outside the group which must be duplicated, we reduce the width of the pipeline registers and improve the usage of any given character within the pipeline. We utilize the METIS graph partitioning library [15].

One clear problem is that of large rulesets (>500 patterns). In these situations it is essentially impossible for a small number of partition to not each have the entire alphabet and common punctuation set. This reduces the effectiveness of the partitioning; however, if we add a weighting function the use of partitioning is advantageous into much larger rulesets. The weighting functions is as follows:

$$W_E = \sum_{i=1}^{min(|k|,|l|)} [(min(|k|,|l|) - i) \text{ if } (k(i) == l(i)) \text{ else } 0] \tag{4}$$

The weight W_E of the edge between k and l is equal to the number of characters $k(i)$ and $l(i)$ in the pattern, with the first character comparison weighted as the length of the shortest pattern. The added weight function causes patterns sharing character locality to be more likely to be grouped together.

The addition of the weighting function in Equation 4 allows the partitioning algorithms to more strongly put patterns with similar initial patterns of characters to be grouped together. The weighting function is weak enough to not force highly incompatible patterns together, but is strong enough to keep similar prefixes together. This becomes important in the tree approach, described next.

Table 1. Illustration of effectiveness of tree strategy for reducing redundant comparisons.

	Number of Prefixes	
No. of Patterns	First Level	Second Level
204	83	126
361	204	297
602	270	421
1000	288	523

4 Tree-Based Prefix Sharing

We have developed a tree-based strategy to find pattern prefixes that are shared among matching units in a partition. By sharing the matching information across the patterns, the system can reduce redundant comparisons. This strategy allows for increased area efficiency, as hardware reuse is high. However, due to the increased routing complexity and high fanout of the blocks, it can increase the clock period. This approach is similar to the *trie* strategy utilized in [1], in which a collection of patterns is composed into a single regular expression. Their DFA implementation could not achieve high frequencies, though, limiting its usefulness. Our approach, utilizing a unary-encoded shift-and-compare architecture and allowing only prefix sharing and limited fanout, provides much higher performance. Beyond the strategic difference of the shift-and-compare architecture, our tree approach differs from the *trie* approach because in that it is customized for the 4-bit blocks of characters. This produces a lower depth tree as there is only a new branch for each block of four, making for a smaller architectural description generation problem. Moreover, the four character prefixes map to four decoded bits, fitting perfectly within a single Xilinx Virtex 4-bit lookup table.

Figure 3 illustrates the tree-based architecture. Each pattern (of length greater than 8) is composed of a first-level prefix and a second-level prefix. Each prefix is matched independently of the remainder of the pattern. After a single-clock delay, the two prefixes and the remainder of the pattern are combined together to produce the final matching information for the pattern. This is effective in reducing the area of the design because large sections of the rulesets share prefixes. The most common prefix is /scripts, where the first and second-level prefixes are used together. The 4-character prefix was determined to fit easily into the Virtex-style 4-bit lookup table, but it turns out that number is highly relevant to intrusion detection as well. Patterns with directory names such as /cgi-bin and /cgi-win can share the same first-level prefix, and then each have a few dozen patterns that share the -bin or -win second-level prefix.

In Table 1, we show the various numbers of first and second-level prefixes for the various rulesets we utilized in our tests. Second-level prefixes are only counted as different within the same first-level prefix. For this table, we created our rulesets using the first n rules in the Nikto ruleset [7]. There is no intentional pre-processing before the tool flow. The table shows that, on average, between 2 and 3x redundancy can be eliminated through the use of the tree architecture.

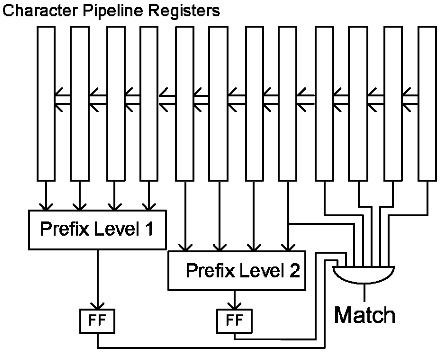


Fig. 3. Illustration of Tree-based hardware reuse strategy. Results from two appropriately delayed prefix blocks are delayed through registers and then combined with remaining suffix. The key to the efficiency of this architecture is that the prefix matches are reused, as well as the character pipeline stages

However, some of this efficiency is reduced due to the routing and higher fanout required because of the shared prefix matching units.

5 Performance Results for Tool-Generated Designs

This section will present results based on partitioning-only unary and tree architectures automatically by our tool. The results are based on ruleset of 204, 361, 602 and 1000 patterns, subsets of the Nikto ruleset of the Hogwash database [7].

We utilized the tool to generate system code for various numbers of partitions. Table 2 contains the system characteristics for partitioning-only unary designs, and Table 3 contains our results for the tree-based architecture. As our designs are much more efficient than other shift-and-compare architectures, the most important comparisons to make are between “1 Partition” (no partitioning) and the multiple partition cases. Clearly, there is an optimal number of partitions for each ruleset; this tends toward 2-3 below 400 patterns and toward 8 partitions for the 1000 pattern ruleset. The clock speed gained through partitioning can be as much as 20%, although this is at the cost of increased area. The tree approach produces greater increases in clock frequency, at a lower area cost. The 602 pattern ruleset shows the most dramatic improvements when using the tree approach, reducing area by almost 50% in some cases; the general improvement is roughly 30%. Curiously, the unpartitioned experiments actually show an increase in area due to the tree architecture, possible due to the increased fanout when large numbers of patterns are sharing the same pipeline.

Table 4 contains comparisons of our system-level design versus individual comparator-level designs from other researchers. We only compare against designs that are architecturally similar to a shift-and-compare discrete matcher, that is, where each pattern at some point asserts an individual signal after comparing against a sliding window of network data. We acknowledge that it is

Table 2. Partitioning-only Unary Architecture: Clock period (ns) and area (slices) for various numbers of partitions and patterns sets

		Number of Patterns in Ruleset			
	No. Partitions	204	361	602	1000
Clock Period	1	4.18	5.18	5.33	5.41
	2	4.45	4.50	5.60	5.17
	3	3.86	4.80	4.55	5.6
	4	3.99	4.24	5.06	5.22
	8	4.17	5.19	4.60	4.93
Area	1	800	1198	2466	4028
	2	957	1394	3117	4693
	3	1043	1604	3607	5001
	4	1107	1692	4264	5285
	8	2007	1891	5673	6123
Total chars in ruleset:		4518	8263	12325	19584
Characters per slice (min):		5.64	6.89	4.99	4.86

impossible to make fair comparisons without reimplementing all other designs. We have defined performance as throughput/area, rewarding small, fast designs. The synthesis tool for our designs is Synopsis Synplicity Pro 7.2 and the place and route tool is Xilinx ISE 5.2.03i. The target device is the Virtex II Pro XC2VP100 with -7 speed grade. We have done performance verification on the Xilinx ML-300 platform. This board contains a Virtex II Pro XC2VP7, a small device on the Virtex II spectrum. We have subsets of the database (as determined to fit on the device) and they execute correctly at the speeds documented in Table 2.

In Table 2 and 3, we see that the maximum system clock is between 200 and 250MHz for all designs. The system area increases as the number of partitions increases, but the clock frequency reaches a maximum at 3 and 4 partitions for sets under 400 rules and at 8 partitions for larger rulesets. Our clock speed, for an entire system, is in line with the fastest single-comparator designs of other research groups. On average, the tree architecture is smaller and faster than the partitioning-only architecture. In all cases the partitioned architectures (both tree and no-tree) are faster than the non-partitioned systems.

The smallest of designs in the published literature providing individual match signals is in [10], in which a state machine implements a Non-deterministic Finite Automata in hardware. That design occupies roughly 0.4 slice per character. Our tree design occupies roughly one slice per 5.5-7.1 characters, making it significantly more effective. While this approach is somewhat limited by only accepting 8 bits per cycle, the area efficiency allows smaller sets of patterns to be replicated on the device. This can increase throughput by allowing for parallel streams of individual 8-bit channels. For a single, high-throughput channel, the stream is duplicated, offset appropriately, and fed through duplicated matchers, allowing multiple bytes to be accepted in each cycle. The tool is capable of generating 4 and 8-byte systems as well (results are included in Table 4, descriptions of these architectures can be found in [13]).

Table 3. Tree Architecture: Clock period (ns) and area (slices) for various numbers of partitions and patterns sets

		Number of Patterns in Ruleset			
	No. Partitions	204	361	602	1000
Clock Period	1	4.89	5.25	5.43	5.35
	2	4.18	4.27	4.8	4.22
	3	3.99	4.15	4.32	5.08
	4	4.1	4.1	4.54	4.69
	8	4.3	4.43	4.628	4.9
Area	1	773	1165	2726	4654
	2	729	1212	2946	3170
	3	931	1410	2210	5010
	4	1062	1345	2316	5460
	8	1222	1587	2874	6172
Total chars in ruleset:		4518	8263	12325	19584
Characters per slice (min):		6.19	7.09	5.577	6.17

Table 4. Pattern size, average unit size for a 16 character pattern (in logic cells; one slice is two logic cells), and performance (in Mb/s/cell). Throughput is assumed to be constant over variations in pattern size

Design	Frequency	Throughput	Unit Size	Performance
USC Unary	258 MHz	2.07 Gb/s	7.3	283
USC Unary (1 byte)	223 MHz	1.79 Gb/s	5.7	315
USC Unary (4 byte)	190 MHz	6.1 Gb/s	22.3	271
USC Unary (8 byte)	160 MHz	10.3 Gb/s	32.0	322
USC Unary (Tree)	250 Mhz	2.00 Gb/s	6.6	303
Los Alamos[4]	275 MHz	2.2 Gb/s	243	9.1
UCLA RDL [2]	100 MHz	3.2 Gb/s	11.4	280
GATech (NFA) [9]	218 MHz	7.0 Gb/s	50	140
U/Crete (FCCM) [3]	303 MHz	9.7 Gb/s	57	170

After partitioning, each pattern within a given partition is written out, and a VHDL file is generated for each partition, as well as a system wrapper and testbench. The size of the VHDL files for the 361 ruleset total roughly 300kB in 9,000 lines, but synthesize to a minimum of 1200 slices. While the automation tools handle the system-level optimizations, the FPGA synthesis tools handle the low-level optimizations. During synthesis, the logic that is not required is pruned – if a character is only utilized in the shallow end of a pattern, it will not be carried to the deep end of the pipeline. If a character is only used by one pattern in the ruleset, and in a sense wastes resources by inclusion in the pipeline, pruning can at least minimize the effect on the rest of the design.

In the 361 pattern, 8263 character system, the design automation system can generate the character graph, partition, and create the final synthesizeable, optimized VHDL in less than 10 seconds on a desktop-class Pentium III 800MHz with 256 MB RAM. The 1000 pattern, 19584 character ruleset requires about 30 seconds. All of the code code except the partitioning tool is written in Perl, a runtime language. While Perl provides powerful text processing capabilities useful for processing the rulesets, it is not known as a high performance language. A production version of this tool would not be written in a scripting language. Regardless of the implementation, the automatic design tools occupy only a small fraction of the total hardware development time, as the place and route of the design to FPGA takes much longer, roughly ten minutes to complete for the 361 pattern, 8263 character design.

6 Conclusion

This paper has discussed a tool for system-level optimization using graph-based partitioning and tree-based matching of large intrusion detection pattern databases. By optimizing at a system level and considering an entire set of patterns instead of individual string matching units, our tools allow more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance. Through preprocessing, our tool automatically generates highly area-efficient designs with competitive clock frequencies.

We release the collection of tools used in this paper to the community at `http://halcyon.usc.edu/~zbaker/idstools`

References

1. Hutchings, B.L., Franklin, R., Carver, D.: Assisting Network Intrusion Detection with Reconfigurable Hardware. In: Proceedings of FCCM '02. (2002)
2. Cho, Y., Mangione-Smith, W.H.: Deep Packet Filter with Dedicated Logic and Read Only Memories. In: The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04). (2004)
3. Sourdis, I., Pnevmatikatos, D.: A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In: The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04). (2004)
4. Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards Gigabit Rate Network Intrusion Detection. In: Proceedings of FPL '02. (2002)
5. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a Content-Scanning Module for an Internet Firewall. In: Proceedings of FCCM '03. (2003)
6. Sourcefire: Snort: The Open Source Network Intrusion Detection System. `http://www.snort.org` (2003)
7. Hogwash Intrusion Detection System: (2004) `http://hogwash.sourceforge.net/`.
8. Global Velocity: (2004) `http://www.globalvelocity.info/`.
9. Clark, C.R., Schimmel, D.E.: Scalable Parallel Pattern Matching on High Speed Networks. In: The Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04). (2003)
10. Clark, C.R., Schimmel, D.E.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In: Proceedings of FPL '03. (2003)
11. Cho, Y.H., Navab, S., Mangione-Smith, W.H.: Specialized Hardware for Deep Network Packet Filtering. In: Proceedings FPL '02. (2002)

12. Sourdis, I., Pnevmatikatos, D.: Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In: Proceedings of FPL '03. (2003)
13. Baker, Z.K., Prasanna, V.K.: A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. Accepted for publication at FCCM '04 (2004)
14. Baker, Z.K., Prasanna, V.K.: Time and Area Efficient Pattern Matching on FPGAs. In: Proceedings of FPGA '04. (2004)
15. Karypis, G., Aggarwal, R., Schloegel, K., Kumar, V., Shekhar, S.: METIS Family of Multilevel Partitioning Algorithms (2004)

BIST Based Interconnect Fault Location for FPGAs

Nicola Campregher[1], Peter Y.K. Cheung[1], and Milan Vasilko[2]

[1] Department of EEE, Imperial College, London, UK.
[2] School of Design, Engineering and Computing, Bournemouth University, UK.

Abstract. This paper presents a novel approach to interconnect fault location for FPGAs during power-on sequence. The method is based on a concept known as *fault grading* which utilizes defect knowledge during manufacturing test to classify faulty devices into different defect groups. A Built-In Self-Test (BIST) method that can efficiently identify the exact location of the interconnect fault is introduced. This procedure forms the first step of a new interconnect defect tolerant scheme that offers the possibility of using larger and more cost effective devices that contain interconnect defects without compromising on performance or configurability.

1 Introduction

The area occupied by wiring channel and interconnect configuration circuits in an FPGA is significant, occupying 50 to 90 percent of the chip area [1]. With current trends aiming to reduce the area occupied by wire segments in the routing channels, wire width and separation have been reduced. This has in turn led to higher occurrences of wiring defects, such as breaks and shorts, and manufacturing yield decrease [2]. As an alternative to increase once again wire widths and separation, we propose a method to categorize devices exhibiting similar functional defects, in order to provide a solution to tolerate such physical defects and increase manufacturing yield.

Our work aims to take advantage of the deep knowledge manufacturers have of the defects occurrences in their devices [11], while trying not to affect the user's load and device performance.

This paper will introduce a new method to categorize faulty devices, as well as providing test procedures to locate device defects whenever needed. The Built-In-Self-Test (BIST) requires a relatively small amount of configurations to efficiently locate and identify a specific type of defect, determined by the defect grade of the device. Our BIST architecture is easily scalable and can detect multiple functional faults.

This paper will introduce a new approach to defect tolerance. In Section 2 a brief summary of the relevant work carried in this field is given. We go on to provide essential background information in Section 3, and introduce the *fault grading* concept in Section 4. Section 5 will introduce the testing procedure. Section 6 provides some details of the implementation, and finally, in Section 7 we give a brief summary and describe future developments of the work.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 322–332, 2004.

2 Previous Work

There are two main strategies to interconnect testing in FPGAs. One is the application of BIST approach to interconnects [3,7,8,9]. The BIST technique proposed by *Stroud et al.* [3] is a comparison based approach. Two WUTs (Wires Under Test) receive identical test patterns and their outputs are compared by ORAs (Output Response Analyzers). This approach however fails to detect multiple faults that have identical faulty behavior in the two WUTs groups. This BIST technique is also aimed at dynamic faults, and can be used during device operation. It provides complete fault coverage, however it requires an external reconfiguration controller.

A similar concept has been proposed by *Niamat et al* [7]. Two sets of 4 wires are applied with test vectors and their output compared. The ORA in this case does not produce a pass/fail indication but a bit sequence. The output response is then stored in a LUT and used at later stages to locate the position of the fault.

A different implementation based on parity check was proposed by *Sun et al* [10]. In this approach the outputs of the WUTs, connected in snake-like chains, are checked for parity against the parity from the original test vectors at the ORA to produce a pass/fail result. This approach however, due to the way parity checking is done, has only 50% error detection efficiency and is very time consuming. The authors in [8] presented a BIST scheme for cluster-based FPGA architectures. Based on the concept of test transparency, they define configurations which enable test access to high density logic clusters embedded within each FPGA tile.

In general, the BIST approaches reported so far require many configurations and are unsuitable if fast testing is needed. The other strategy is a more classical approach not using BIST. Various researchers have proposed different models [4,5,6], based on different assumptions. All these methods, albeit very fast and compact, are limited in functionality by the number of I/O pins needed to access the chip at various stages of the testing process. They are thus unsuitable to applications requiring resource inexpensive testing.

3 Background

3.1 SRAM FPGA Architecture and Physical Layout

The target FPGA architecture is an island-style FPGA. As stated in previous sections, our work is targeting problems that may arise as devices grow in size. We are therefore targeting the latest top-end devices [12,13]. These have clear architectural characteristics, such as hierarchical routing and multiple LUTs in their configurable logic blocks. Because of their advanced development, the switch matrices inside these elements are very complex. Unfortunately, not much is known about the their structure and connectivity. We will therefore assume a simple switch matrix block, where any incoming wire can connect to only one other wire in all 3 directions. This simplified model, while sufficient to demonstrate basic principles of our method, can easily be extended to cope with complex switch matrices.

Furthermore, we make the assumption that the defect occurs into only one type of interconnect resource and that all others are functionally faultless. This in turns leads to the assumption that all wires of the same segment type lie on the same physical layer. Furthermore, wires of the same type in the horizontal and vertical channels do not lie in adjacent layers, thus eliminating the issue of cross-connections between them. All these assumptions can easily be relaxed without affecting the basic method described.

3.2 Fault Models

Our structural fault model only targets FPGA's interconnect resources. The interconnect structures of a typical FPGA include wires and programmable switch matrices.

Typical faults affecting wirings are short and open faults. Switch Matrices faults are limited to switches being *stuck-off* or *stuck-on*. *Stuck-off* faults are treated in the same way as wire opens. *Stuck-on* faults, however, are more difficult to detect and diagnose. *Stuck-on* faults mean that two wires are permanently connected via the switch matrix. In the presence of stuck-on faults the signal is propagated through an unwanted resource and hence the fault location procedure has to account for all possibilities. This means that all possible combinations of switch matrix configuration have to explored. Figure 1 shows all the possible fault occurrences.

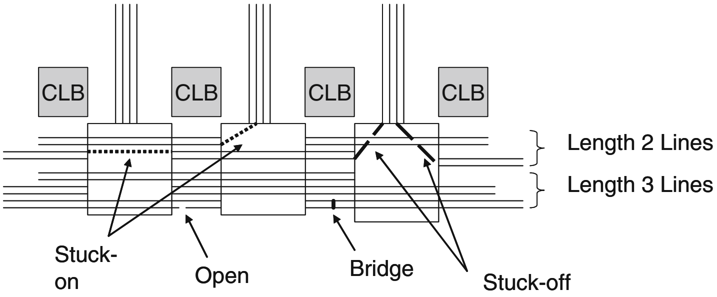


Fig. 1. Fault Models

4 Fault Grading

Devices undergo a multitude of manufacturing tests at various stages of the manufacturing process. Some degree of test is performed at the wafer level, where defective devices are marked and discarded after cutting. Parametric tests are performed on packaged devices; failing devices are once again discarded, whereas devices that pass parametric tests are "binned" into different speed categories depending on how they performed during tests. Failed devices are mainly discarded even though the total amount of resources affected by a physical defect is minimal. Some of these devices can be used, albeit not in full capacity. Manufacturers have already looked at ways to reuse faulty devices. One such solution is offered by Xilinx with their Easypath devices[14]. Easypath devices are tested only for the resources used for a specified design. This means that devices do not have to pass all manufacturing tests, but only a limited number of them. Customers are then offered with devices mapped exclusively for their design, but at a reduced cost. They however lose the possibility to reconfigure the devices for future upgrades.

Instead of using the Easypath approach, we propose that devices can be categorized with respect to the functional fault they exhibit. Functional faults are independent of physical faults, such as open vias or masking defects found during manufacturing tests [11]. A specific functional fault only affects part of the device, and if it can be avoided, the rest of the chip can be made to work with only slightly altered characteristics. Our fault grading scheme aims to provide fault categories for defective devices that have failed similar functional tests.

The concept of fault grading is very similar to that of speed grading: devices will always exhibit different characteristics, and are therefore categorized according to specific parameters. Devices are marked and designs compiled according to those specific parameters. It is therefore possible to generate new categories, and using this information defective devices can be used to implement the majority of designs.

The fault grades contain information about the fault the device exhibits. The amount of information the fault grades contain is a trade-off between what is needed in order to avoid the fault and generalization of the defect. One extreme is a fault grade that contains the exact location and type of fault. The other extreme is a simple tag identifying a faulty device. A good compromise is a fault grade that indicates what type of resource is affected. This leads to a limited number of fault grades, that contain enough information to generate test configurations to locate the fault in the device during power-on.

As an example, consider a Xilinx Virtex II PRO device and its general routing matrix [12]. This device offers 4 different types of lines: direct connections, double lines, hex lines, long lines. Four fault grades could be used to categorize fault on the wire resources. Two grades could be used for switch matrices faults, one to identify stuck-on faults and one for stuck-off faults. These grades are chosen with a defect tolerance scheme in mind, and how to avoid certain defects with the lowest overhead possible. Assuming that all other resources are unaffected, we can efficiently test all the interconnects of the same type that could possibly be faulty, during the power-on sequence, in order to provide an alternative design to avoid the faulty resource.

5 Testing Strategy

We propose a BIST methodology to detect and diagnose a functional fault on a known interconnect resource type. Our strategy consists of a point to point analysis, where a wire is forced to a logical value at one end and observed at the other. If the observed output is not equal to the input, a fault is present. A BIST environment consists of the *Test Vector Generator*(TVG), the *Wires Under Test*(WUT), and the *Output Response Analyzer*(ORA). TVGs select the pattern of 0's and 1's to be applied on the WUTs, while the ORAs compare the the WUTs response against a predefined sequence and issue a pass/fail signal accordingly.

Considering the nature of modern FPGAs, where routing channels are considerably large, it is feasible to group the Wires Under Test together and perform an analysis at the ORA of all grouped wires.

TVGs and ORAs can be implemented using CLBs. As most modern devices are made of large CLBs (comprising of multiple LUTs) we can implement a TVG *and* a ORA using a single CLB. The TVG/ORA combinations are arranged in a chain that spans the entire width or height of the device.When a fault is detected, a 'fail' signal is passed on through to the chain end. The propagation within the chain is synchronized with a clock, so that an ORA in the N^{th} position in the chain will only be allowed to access the chain at the N^{th} clock cycle. When a 'fail' signal is detected at the end of chain, the position of the chain in the array is found by the BIST controller using simple decoding logic. A diagram of such a system is shown in Figure 2.

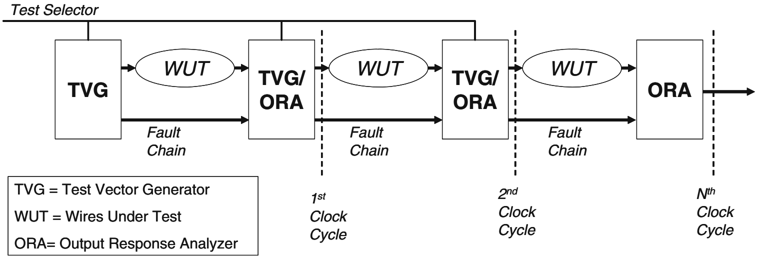


Fig. 2. Testing Strategy

5.1 WUTs Grouping

Taking into account that 4-input LUTs are the main building block in most modern FPGA, the simplest ORA implementation is by using one such element. The TVGs are implemented using multiple LUTs, one for each wire in the set of WUTs. This allows complete independence of test vectors between wires in a set of WUTs. As a compromise between TVGs and ORAs implementations, it was decided to group the WUT in groups of 4. This arrangement would require a single 4-input LUT for the ORA, whereas 4 4-input LUTs would be required for the TVGs. Such quantities are not uncommon in readily available devices [12]. The implementation can be altered to best fit any device with different architectural characteristics. The resulting arrangement is shown in Figure 3. The dotted lines in Figure 3 represent wires from the adjacent set of wires, which have to be driven with opposite signals as the adjacent WUT to account for bridging faults across assigned sets. Those wires are not considered at the ORA but might nonetheless have an effect on the WUT in case of bridging faults.

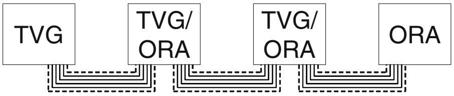


Fig. 3. Grouped WUTs between ORAs and TVGs

5.2 TVG and ORA Operation

TVGs generate bit sequences to account for all possible faults that could develop in the interconnect resource. They are implemented as simple look-up tables, where the output is selected from the Test Selector input. The Test Selector input is generated from the BIST controller, and is a global signal routed to all TVGs. For wiring faults, four basic test vectors can detect any defective occurrence. These, defined as the four *basic* vectors, are:

- 0000 Tests for stuck at 1 faults.
- 1111 Tests for stuck at 0 faults.
- 1010 No two adjacent wires are applied the same value. Tests for bridging faults.
- 0101 Alternating 1's and 0's, in reverse order from the previous test vector. Tests for bridging faults.

The *basic* test vectors can identify the set of WUTs that contains a fault. To correctly identify the exact faulty wire within a given set of WUTs, extra test vectors can be used. This second set of vectors is dependent upon the result of four *basic* test vectors and is decided by the BIST controller. The function of the second set of vectors is purely to improve the fault resolution.

The ORA function is to generate a pass/fail signal according to some predefined parameters. The ORA is designed to fit in a single 4-input LUT and under our scheme, it will issue a 'pass' signal only if the four WUT have logical values corresponding to the 4 basic test vectors. Under all other circumstances it will issue a 'fail' signal.

5.3 BIST Controller Operation

The BIST controller operation during test is shown in Figure 4. While the test vectors are being propagated through the chains, the BIST controller checks the end of all chains for any 'fail' signal being issued. If such a signal is found, the current counter value (representing how far along the chain the vectors have been propagated) and the chain end identifier represent the coordinate of the ORA that has detected the fault. The output from the BIST controller is a string of four bits regarding which of the four basic test vectors has found a fault. If, for instance, the string of results from the BIST controller is 1000, test vector 1 has caused a fault. This means that the fault present in the system is a *stuck-at-1* fault, as test vector 1 could not have caused or detected any other unexpected behavior.

```
var ChainEnds: array of binary(0 to N-1) :=(all=0);      //Chain Ends
var result: array of binary (0 to 3) := '0000'          //Test results
var counter, x_coord, y_coord: integer

begin
 for ( j in 0 to 3)
      case j is
           when (0) - apply 0000                         //Test vectors
           when (1) - apply 1111
           when (2) - apply 1010
           when (3) - apply 0101
      end case
      counter := 0
      for (x in 0 to M-1)
          for (i in 0 to N-1)
               if ChainsEnds(i) = 1 then                  //Fault found
                    result(j) = 1                         //Fault recorded
                    x_coord := counter
                    y_coord := i
                 end if
          end for
          counter:=counter + 1
      end for
 end for
end
```

Fig. 4. BIST operation during test

From the inspection of the test results of the *basic* test vectors the BIST controller can determine what type of fault is present in the system and apply other test vectors to identify exactly which wire in the group of WUTs is faulty. Note than any fault or combination of faults confined within the set of WUT would cause at least two tests to fail. The only possible fault not confined within the set of WUT is a bridge onto adjacent set of wires. This causes only one of the bridging test vectors to fail. From the combination of failed tests the BIST controller can reduce the fault resolution to 2 or 3 wires or pairs of wires in the set of WUTs, as shown in Table 1. The second set of test vectors is designed purely to increase the fault resolution by selection of any one of the already selected wires. During propagation of the extra test vectors, the pass/fail signal from the ORAs are used as selection between wires to identify the faulty one.

If, for example, the combined test results are 1010, the fault is limited between Wire 2 or Wire 4 being stuck at 1. The next test vector, 1110, is then propagated.

Table 1. BIST selection

Test Vectors (Wire 1 - Wire 4)					
(1) - 0000	*(2) - 1111*	*(3) - 1010*	*(4) - 0101*	*Fault*	*Next Vector*
0	0	0	0	No Fault	*N/A*
1	0	1	0	Wire 2 or Wire 4 s-a-1	1110
1	0	0	1	Wire 1 or Wire 3 s-a-1	0111
0	1	1	0	Wire 1 or Wire 3 s-a-0	1000
0	1	0	1	Wire 2 or Wire 4 s-a-0	0001
0	0	1	1	Bridge	All previous 4
0	0	0	1	Bridge onto next set	All previous 4
0	0	1	0	Bridge onto next set	All previous 4
All others				Multiple faults	Composite

As by this point we have eliminated the possibility of any other fault, the ORA inputs can only be 1110, if Wire 2 is s-a-1, or 1111, if Wire 4 is s-a-1. The first option will result in a 'fail' signal from the ORA, whereas the second option will result in a 'pass' signal. From the ORA response we can therefore increase our fault resolution to identify precisely the faulty wire.

6 Implementation

We are proposing a BIST strategy to be used with prior knowledge of the faulty resource. Our BIST strategy is a point to point one, where test vectors are applied at one end to a set of WUTs and observed at the other. TVGs and ORAs are arranged in rows, so that pass/fail results are propagated and read from only one location for each row. The BIST controller decodes the outputs from the end of the ORA chains to provide fault location. The WUTs are grouped in sets in order to offer the highest degree of parallelism considering the architectural and strategic constraints. The total number of configurations needed to complete testing is dependent upon the total number of wires of the same type present in the device. The configurations are grouped into phases, where configurations belonging to the same phase aim to test different interconnects appertaining to the same channel.

6.1 Number of Configurations

To fully test a routing channel all lines originating or terminating from a single CLB have to be tested. If the architecture has L lines of a specific type originate from any CLB in a channel, then the test of all lines in a channel will need $\lceil L/4 \rceil$ number of configurations. Modern FPGAs rarely have more than 20 lines of any type generating from any CLB in one channel [12], hence 5 configurations are sufficient to test all lines in channel. These make up a test phase.

6.2 Wire Testing Phases

Considering an $M \times N$ array , with N CLBs in each horizontal channel and N CLBs in each vertical channel, $N + 1$ vertical routing channels and $M + 1$ horizontal routing channels exist [1]. Testing of each horizontal or vertical routing channel requires all the CLBs in a row or column, respectively. In the vertical and horizontal direction, testing of all channels requires at least 2 phases, where during the first N or M channels respectively are tested, and in the second the channel left over is tested. The second phase of the vertical and horizontal channels testing can be combined, as shown graphically in Figure 5. Three phases are required to test all the lines of the same type in all channels. If $\lceil L/4 \rceil$ are required for each phase, a total of $3 \times \lceil L/4 \rceil$ are needed for testing the whole device.

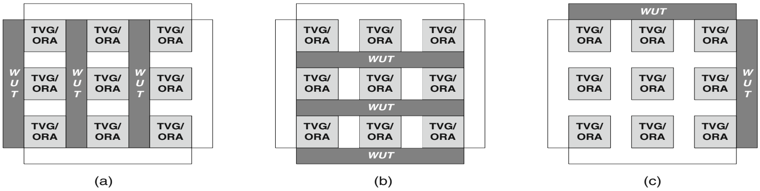


Fig. 5. Three configuration phases

6.3 Switch Matrix Testing Phases

To test for switch matrix faults the WUTs signals are routed trough switch matrices. *Stuck-off* faults are dealt dealt with just like open faults. In the event of *stuck-on* faults, bridges are created within the switch matrix configurations for detection.

The switch matrix configurations needed to test for all *stuck-on* and *stuck-off* faults ar shown in Figure 6. The diagram shows the routing inside the switch matrix in order to cause bridging faults under all possible matrix combinations. At the same time, the routing shown also explores all the possible connections within the matrix itself. The testing scheme remains unchanged: if an ORA

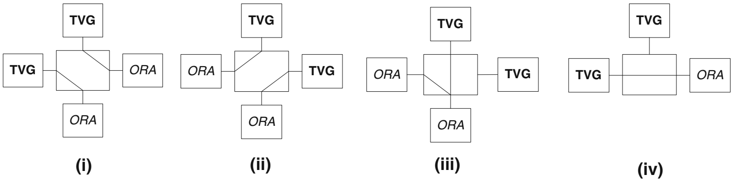


Fig. 6. Switch Matrix fault diagnosis phases

detects an unexpected behavior, a ’fail’ signal is propagated through the end of the chain. The only tweak to the original scheme is that two TVGs connected to the same switch matrix produce opposite signals. In the case of *stuck-on* switches, this causes a behavior identical to that of a bridging fault.

The diagnosis of *stuck-off* faults is straight forward, as ’fail’ signal from any ORA can only be caused by one faulty connection in all routing configurations. For *stuck-on* faults, however, a different analysis has to be performed. A permanent connection between two terminal will cause all the ORAs connected to the faulty switch matrix to detect a fault. But any permanent connection will only be detected during a specific number of routing configurations. From the analysis of the result of the 4 test phases, the BIST controller can determine the exact faulty connection. The faults and failures caused are summarized in Table 2.

Table 2. Stuck-on faults resolution

Faulty Connection	***Routing configuration detected by***
North-East	ii,iii,iv
North-West	i,iii,iv
South-East	i,iii
South-West	ii
North-South	i,ii
East-West	i,ii,iii

6.4 Case Study: Xilinx Virtex II Pro

This FPGA device allows TVG and ORAs to be implemented in a single CLB, thanks to the high number of 4-input LUTs present. For simplicity purposes we consider the case where a double line in the general routing matrix is faulty for a XC2V20 device. The Virtex II Pro has 20 double lines originating from a CLB in both vertical and horizontal channels. This leads to a total of 5 configurations per test phase. Therefore a complete test would require 15 configurations. to fully test all double lines available in the FPGA. Assuming a worst-case scenario of JTAG download, each configurations requires $249ms$, so the total time required for reconfigurations is $3.74s$. This time can be considerably reduced if a SelectMap interface is used for download. In this case, total download time would be just over $0.3s$. The actual test time, in terms of clock cycles is in both cases much smaller than the configuration download time and thus it would not affect total testing time by a great amount.

7 Conclusions and Future Work

We have presented a new framework for FPGA testing for defect tolerance. The concept of device fault grading has been introduced, together with simple, effective testing procedures. Under our scheme it is possible to load testing configurations to the FPGA with the specific aim of locating a fault whose nature is already known. The testing is done completely on-chip.

This work provides manufacturers and users with a different approach to defect tolerance. The development of this framework is based around the assumption that defective devices will show similar functional faults spread around the chip area. It is possible to categorize these defects with respect to their functional faults. In the design process we can account for the fault to be found anywhere around the chip and limit the usage of a faulty resource to a minimum. The exact location of the fault can be found by loading the proposed test configurations during the power-on sequence

The next step in our work will be to integrate the fault grading and fault diagnostic into a complete defect tolerance framework, offering an alternative design to the most common defect tolerance problems.

References

1. S.D.Brown, R.J.Francis, J.Rose, and Z.G.Vranesic, *Field Programmable Gate Arrays.* Norwell, MA:Kluwer, 1992.
2. F. Hanchek, and S. Dutt, "Methodologies for tolerating cell and interconnect faults in FPGAs," *Computers, IEEE Transactions on* , Vol. 47(1), pp. 15-33, Jan. 1998.
3. C. Stroud, S.Wijesuriya, C.Hamilton, and M.Abramovici, "Built in self test of FPGA interconnect," *Proc.Int. Test Conf.*, pp. 404-411, 1998.
4. M.Renovell, J.Figueras, and Y.Zorian, "Test of RAM-based FPGA: Methodology and application to the interconnect structure," in *Proc. 15th IEEE Very Large Scale Integration (VLSI) Test Symp.*, 1997, pp. 204-209.
5. M.Renovell, J.M.Portal, J.Figueras, and Y.Zorian, "Testing the interconnect of RAM-based FPGAs," *IEEE Des. Test Comput.*, 1998, pp. 45-50.
6. H.Michinishi, T.Yokohira, T.Okamoto, T.Inoue, and H.Fujiwara, "Test methodology for interconnect structures of LUT-based FPGAs," *Proc. 5th Asian Test Symp.*, pp. 68-74, 1996.
7. M.Y.Niamat, R.Nambiar, and M.M. Jamall, "A BIST Scheme for testing the interconnect of SRAM based FPGAs," *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, Vol. 2 pp. 41-44, 2002.
8. I.G. Harris and R. Tessier, "Testing and diagnosis of interconnect faults in cluster-based FPGA architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.21, pp.1337-1343, 2002.
9. J.Liu and S. Simmons, "BIST diagnosis of interconnects fault locations in FPGA's," *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*, Vol. 1, pp.207-210, May 4-7, 2003.
10. X. Sun, S.Xu, J.Xum and P.Trouborst, "Design and implementation of a parity-based BIST scheme for FPGA global interconnects", *CCECE*, 2001.
11. Xilinx Inc., "The Reliability Report," Sep. 2003.
12. Xilinx Inc., "Virtex II Pro Platform FPGA Handbook," Oct. 2002.
13. Alter Corp., "Stratix II Device Handbook," Feb.2004.
14. Xilinx Inc., "Virtex II Pro EasyPath Solutions," 2003.

FPGAs BIST Evaluation*

A. Parreira, J.P. Teixeira, and M.B. Santos

IST / INESC-ID, R. Alves Redol, 9, 1000-029 Lisboa, Portugal
marcelino.santos@inesc-id.pt

Abstract. This paper addresses the problem of Test Effectiveness (TE) evaluation of digital circuits implemented in FPGAs. A Hardware Fault Simulation (HFS) technique, particularly useful for evaluating the effectiveness of built-in self-test (BIST) is detailed. This HFS, efficiently, injects and un-injects faults using small partial reconfiguration files and ascertain (or not) the BIST to be used in the FPGA circuits. Different fault models are compared regarding their efficiency and complexity. The methodology is fully automated for Xilinx Spartan and Virtex FPGAs. Results, using a Digilab2 board, ISCAS'85 and 89 benchmarks, show that our methodology can be accurate and orders of magnitude faster than software fault simulation even with more demanding fault models.

1 Introduction

Stand-alone or embedded FPGA cores in Systems on a Chip (SoC) need to be tested. Dependable systems require that, during product lifetime, FPGA circuits may be tested, in particular for the target-programmed functionality. Such system specification makes the use of Built-In Self-Test (BIST) technology very attractive. In fact, lifetime testing using BIST may allow a low cost solution, and by using at-speed testing, it enables the detection of *dynamic* faults, which are relevant in DSM (Deep Sub-Micron) semiconductor technologies [1]. BIST technology routinely uses low cost PR (Pseudo-Random) TPG (Test Pattern Generators), like LFSRs, and signature analyzers, like MISRs. Random pattern resistant faults may require refined approaches, e.g., as weighted PR, or re-seeding techniques [2]. The modules under self-test may be combinational, sequential or reconfigured in test mode. Nevertheless, *test quality* needs to be ascertained, during the design phase. Four quality metrics can be defined: *test effectiveness*, *test overhead*, *test length* (the number of test vectors in the test session) and *test power*. The necessary condition is *test effectiveness*, i.e., the ability of a given test pattern to detect likely defects, induced during IC manufacturing or during lifetime operation and aging. Defects coverage is usually evaluated through the FC (Fault Coverage) metrics, using a fault model, such as the single Line Stuck-At (LSA) fault model. Typically, BIST solutions lead to low test overhead (additional hardware resources / speed degradation), large test length (compensated by at-speed vector application) and moderately high test power (Power consumption required for the BIST session).

* This work has been partially funded by FCT (Portugal), POCTI/ESE41788/2001.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 333–343, 2004.

The major issue addressed in this paper is: *how can we, in a cost-effective way, evaluate the test effectiveness in a FPGA circuit?* This question can be divided in two sub questions: *what to test* (fault model related) and *how to evaluate test effectiveness* (methodology related).

What to test?

The functionality of circuits implemented in FPGAs depends not only on the absence of physical defects in the device but also on the presence of the right configuration. As a consequence, there are two possible approaches for testing FPGA circuits:

1. Test the hardware as exhaustively as possible and afterwards reconfigure and/or read back the device configuration in order to make sure that it is correct [3][4];
2. Test the circuit structurally with its configuration;

The first approach has the advantage of being implementation independent but has serious disadvantages in test overhead, length and power, since the configuration information needs to be duplicated and exhaustive hardware test requires from the system the capability of reconfiguring the FPGA. However, those overheads can be significantly reduced if the reconfiguration hardware is shared by several FPGAs. In that case the remaining cost is reduced to the duplication of the configuration information. This solution is not BIST at chip level, since the FPGA cannot test its own functionality, but can be viewed as system level BIST.

The second approach is the chip level BIST. However, for FPGA circuits, self-test effectiveness, besides the usual measure of the (physical) defects coverage must also include an evaluation of the correctness of the configuration. Thus, for test effectiveness evaluation, structure and configuration faults must be taken into account.

How to evaluate test effectiveness?

For complex devices, fault simulation (FS) may be a very costly process, especially for sequential circuits. In fact, circuit complexity, test length and fault list size may lead to a large computational effort. Although many efficient algorithms have been proposed for SFS (Software Fault Simulation) (see, e.g., [5] and [6]), for complex circuits it is still a very time-consuming task and can significantly lengthen the time-to-market. Moreover, observability in embedded BIST is not for each vector, but only for each signature captured after all test vectors are applied. This fact may compromise fault dropping, routinely used to severely restrict SFS costs.

FS can be implemented in software or hardware [1]. The ease of developing software tools for FS (taking advantage of the flexibility of software programming) made SFS widely used. However, the recent advent of very complex FPGAs components created an opportunity for HFS (Hardware Fault Simulation), which may be an attractive solution for at least a subset of practical situations. As an example, BIST effectiveness evaluation may require a heavy computational effort in fault simulation since long test sessions are needed to evaluate systems composed of several modules that have to be tested simultaneously in order to evaluate aliasing and generate fault dictionaries for diagnosis purpose. Another FS task not efficiently performed by software tools is *multiple fault simulation*, mainly because of the enormous number of possible fault combinations. However, multiple fault simulation may become mandatory, namely in the certification process of safety-critical applications [7]. HFS may cope with these issues. In particular, *if a FPGA design with BIST is programmed in the target FPGA, could we use it for fault injection and test effectiveness evaluation?*

In this paper, the classical single and multiple LSA fault models are compared with the recently proposed [8] CSA (combination stack at) fault model using a highly efficient HFS methodology and tool for BIST effectiveness evaluation. The HFS is based on efficient partial reconfiguration with very small bit files used for fault injection. These bit files are obtained from the full configuration bit file, by means of its direct manipulation, without requiring any additional software tools.

In section 2, previous work is reported. In section 3, fault models are discussed. Section 4 details LUTs extraction and HFS using partial reconfiguration. Section 5 describes experimental results. Finally, section 6 presents the conclusions of this work.

2 Previous Work

Several research works have addressed the test of FPGAs in the recent past. [3] and [4] are examples of how FPGA hardware can be reconfigured and tested, without extra dedicated hardware in the FPGA, taking advantage of the reconfiguration capability. The main focus of research in this area targets on-line test and test time minimization while assuring exhaustive hardware defects coverage. The configuration loaded is not tested and external hardware is required to reconfigure the FPGA under test. In this work it is assumed that the FPGA should be able run BIST without any external dedicated equipment.

Test effectiveness must be evaluated via fault simulation. Different HFS approaches using FPGAs have been proposed in the literature, mainly targeting ASIC prototyping. Dynamic fault injection, using dedicated extra hardware, and allowing the injection of different faults without reconfiguring the FPGA, was proposed in [9-12]. The additional hardware proposed for implementing dynamic fault injection uses a Shift Register (SR) whose length corresponds to the size of the fault list. Each fault is injected when the corresponding position in the SR has logic "1", while all other positions have logic "0". Upon initialization, only the first position of the SR is set to "1". Then, the "1" is shifted along the SR, activating one fault at a time. This technique was further optimized in [13]. However, a major limitation of this technique is the fact that the added hardware increases with the number of faults to inject, which limits the size of the circuits that can be simulated. In [10], it is shown that parallelism is possible by injecting independent faults at the same time. This parallelism is limited to faults in different propagation cones; however, the reported speedup is only 1.36 times the pure serial FS. In [14], a new approach that included a backward propagation network to allow critical path tracing [15] is proposed. This information allows multiple faults to be simulated for each test vector; nevertheless, it also requires heavy extra hardware. Only combinational circuits have been analyzed. A serial FS technique that requires only partial reconfiguration during logic emulation was proposed in [16], showing that no extra logic need be added for fault injection purposes. The authors show that HFS can be two orders of magnitude faster than SFS, for designs over 100,000 gates. More recently, other hardware fault injection approaches were proposed in [17-19] using the JBITS [20] interface for partial FPGA reconfiguration. The JBITS software can achieve effective injection of faults on Look-Up-Tables (LUTs) [17,18] or erroneous register values [19] requiring the Java SDK 1.2.2 [21] platform and the XHWIF [22] hardware interface.

3 Fault Models

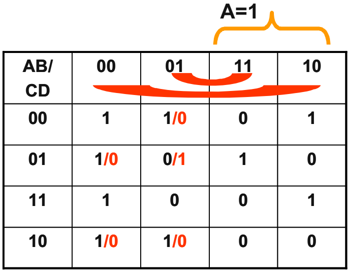
3.1 Structural Faults

In the HFS works mentioned in previous section, FPGAs are used for ASIC prototyping purposes and their possible faults are not the main target of the test.

In order to test a circuit implemented in a FPGA, ensuring that its functionality is correct with a certain degree of confidence (measured by test efficiency), not only structural faults must be modeled, but also configuration must be tested.

Most efficient HFS methods are based in LUT fault injection since it is possible to inject/un-inject LUT faults very efficiently and LUTs are used to implement a significant part of the circuit's logic (for prototyping purposes a significant sample of the fault coverage can be evaluated).

LSA faults are the type of faults commonly modeled in LUTs. The reconfiguration vector that corresponds to the LUT input *A* stuck at value *v* is obtained by copying the values y_{vBCD} to $y_{\neg vBCD}$ for each *BCD* combination. For instance, the vector for the fault input *A* LSA-1 is obtained, as illustrated in Figure 1, by copying y_{1000} to y_{0000}, y_{1001} to y_{0001}, y_{1110} to y_{0110}, …, y_{1111} to $y_{0111.}$ Fault collapsing can be made easily by identifying the faults that require the same faulty LUT contend.



AB/ CD	00	01	11	10
00	1	1/0	0	1
01	1/0	0/1	1	0
11	1	0	0	1
10	1/0	1/0	0	0

Fig. 1. Computing the 16 bits for LUT injection of fault "input *A* LSA-1".

This LUT LSA fault model significantly evaluates if the structure of the circuit is tested since the logic implemented in LUT gives an over 70% sample of the CLBs structural faults [23].

Routing configuration bits are also partially tested in this structural LUT test since LSA are modeled in all LUT ports (a significant sample of the CLB used logic) and routing faults may be detected as incorrect values at LUT's inputs. However, it is possible that the wrong connected node exhibits a correct logic value each time it is observed. The probability of this undesired aliasing is reduced when the number of observations is increased.

3.2 Configuration Faults

In the example of Figure 1 there are five LUT input combinations that can produce an erroneous output in the faulty LUT (corresponding to five changes in the LUT contents). If the test sequence includes one of them, and the corresponding output is observed, this fault is considered detected by the test.

In order to test each LUT position content and reduce the aliasing probability of routing faults the *Combination Stuck At* (CSA) fault model [8] must be used. The number of faults, for this model, is identical to the possible combinations of LUT active inputs. Each fault is injected by inverting only the LUT bit position that corresponds to one input combination. The total coverage of CSA faults corresponds to the exhaustive LUT functionality test for the programmed configuration. Thus, CSA is a *functionality* driven fault model instead of a *line* driven fault model. It models configuration faults and also structure faults since LSA fault model can be viewed as multiple CSA fault modeling.

Note that CSA test requires half the vectors needed for testing each LUT position with both logic values (exhaustive LUT testing). Table 1 compares the number of CSA faults with the number of LSA faults for different LUT types. The column “LSA colap.” presents an approximate number of collapsed LSA faults based on the experiments that are reported in section 5. This Table shows that, when the CSA fault model is used, it leads to an increase in the fault list size, especially for LUTs with 4 active inputs. As these are the most used LUTs, CSA fault lists are around 50% bigger then LSA fault lists. This increase reflects linearly in the fault simulation time, since in HFS faults are injected consecutively. Nevertheless, since partial reconfiguration based HFS is very fast, this increase in the fault list size is an affordable price to pay for the increase of accuracy granted by the CSA model, as it will be demonstrated in section 5.

Table 1. Number of LSA faults and CSA faults.

LUT type	LSA faults	LSA colap.	CSA faults
LUT0	2	1	1
LUT1	4	2	2
LUT2	6	4	4
LUT3	8	5	8
LUT4	10	8	16

4 LUTs Extraction and Fault Injection

4.1 LUT Extraction

Xilinx Virtex [24] and Spartan FPGA components were used in this work, due to the fact that partial reconfiguration of these components is possible and documented [25]. The proposed methodology is an extension of the work reported in [8]. As shown in [25], the binary file for configuration of these FPGAs can be divided in 6 major parts:

1. Header with project and target FPGA id.
2. Enable program mode.
3. CLB configuration.
4. BRAM configuration.
5. Error detection code (crc).
6. Enable normal operation mode.

Each configuration file (total or partial) consists on sequences of commands and frames. Each command contains two 32 bit word: the first word is an operation code. The second word is an operand. Frames are the smallest amount of data that can be read or written with a single command; frame size depends on the specific FPGA number of rows: 18 times the number of rows, rounded up to a multiple of 32. CLBs (Configurable Logic Blocks) are the building blocks for implementing custom logic. Each CLB contains two slices. Each slice contains two 4 input LUTs, 2 flip-flops and some carry logic. In order to extract the LUT's configuration, the software tool developed analyses the part of the frames that describes the CLBs configuration, as depicted in Figure 2.

Each LUT can be used to implement a function of 0, 1, 2, 3 or 4 inputs. The number of inputs relevant for each used LUT must be identified, for the target functionality, in order to include the corresponding faults in the fault list.

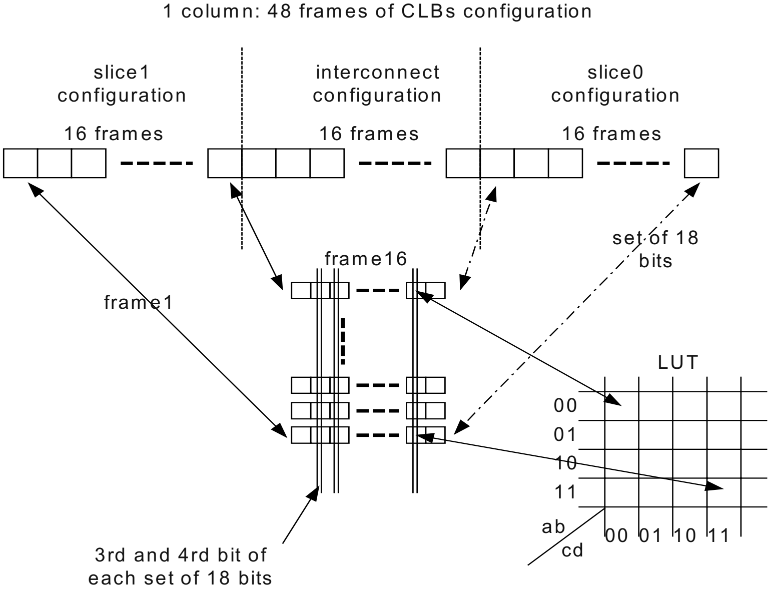


Fig. 2. Positions in the frames of the LUT configuration bits.

Since each LUT has 4 inputs, the LUT contents consist of a 16-bit vector, one vector for each combination of the inputs. The LUT configuration 16-bit vector can be denoted y_{0000}, y_{0001}, y_{0010}, …, y_{1111}, where each bit position corresponds to the LUT output value for the respective combination of the inputs i_3, i_2, i_1 and i_0. If one input has a fixed value, then an 8 bit vector is obtained. For instance, if we have always i_3=0, then the relevant 8 bit vector is y_{0000}, y_{0001}, y_{0010}, y_{0011}, y_{0100}, y_{0101}, y_{0110}, y_{0111}.

After retrieving the information of each LUT from the bit file, the relevance of each input i_x (x=0,1, 2 and 3) is evaluated, by comparing the 8-bit vectors corresponding to i_x=0 and i_x=1. If these two vectors are identical, then the input i_x is not active. This is how we extract the LUT types, e.g., LUT2, LUT3 or LUT4, according to their number of relevant inputs.

4.2 Fault Injection and Simulation

In order to inject a fault, the LUT content must be changed. This change depends on the fault to inject and on the LUT fault free information. LUTs with identical fault free configurations require fault injections with the same faulty LUT contend. In order to increase efficiency, the faulty LUT contend is pre-computed for each possible fault in the fault-free LUT configuration.. The definition of fault model is made in a file that associates one or several LUTs faulty contents to each possible combination of the 16 bits that correspond to the fault free LUT configuration. The simulation tool, after loading this file, sequentially injects the faults using bit files that reconfigure only the minimum number of frames required to un-inject the previous fault and inject the next one. The binary file for partial reconfiguration requires the faulty frames and a restricted number of commands, which we don't find clear in [25], and we have identified as:

1. Header - 32 bits “1”.
2. Synchronization command.
3. CRC reset command.
4. FLR – spec. of the frame size.
5. COR – start-up enable command.
6. MASK – en. writing the CTL reg.
7. ASSERT HIGH
8. FAR – add. of initial CLB frame.
9. write – en. configuration writing.
10. FDRI – nbr of conf. words to send.
11. CLBs configuration words
12. CRC
13. LFRM – type / add. of last frame
14. FDRI – number of configuration words to send for the last frame.
15. Config. words for the last frame.
16. START-UP – start-up enable.
17. CTL – Memory read enable and config. pins use specification.
18. CRC
19. Termination bits.

This partial reconfiguration is repeated during the FS process. In order to start each FS, the developed tool sends a start-BIST commands and waits for BIST-end to read the signature in the multiple input shift register (MISR). At present, this interaction between the software that controls the fault simulation process and the FPGA is accomplished using the JTAG 200 Kb parallel III port. Xilinx USER1 instruction is used as the start-BIST command. USER2 instruction is the read signature command. The validation of the tool was carried out in a *DIGILAB 200 E* board with a Xilinx Spartan 200 E FPGA. However, the developed software tool supports every board with JTAG interface with Virtex or Spartan FPGAs, including the E type.

The developed tool delivers a fault simulation report. This report includes, for each fault, the target LUT type and location, equivalent faults, detection result and MISR signature. The LUT inputs and output are also identified, using information from the LUT extraction.

In order to enable fault dropping and increase performance, a twin implementation of the circuit under test is configured. Faults are injected in one of the circuits and each output response is compared instead of BIST signatures. Besides fault dropping, this approach enables FC tracing.

5 Results

In Table 2, the collapsed number of faults is given, for the CSA, single and multiple LSA fault models, for ISCAS'85 and 89 benchmark circuits [26] C7552, S5378 (with full scan). Multiple LSA faults include all combinations of 2, 3 and 4 LSA faults in the LUT inputs. In this table HFS times are also reported for the simulation of **one million** PR test vectors with each fault list. Fault collapsing is more relevant, as it can be seen, for multiple faults model. HFS was carried out at 25MHz.

A commercial SFS tool required 4261 seconds in order to simulate **65535** vectors in the C7552 with the 4298 prime LSA faults on a Sun ultra10/440 workstation with 1 GB RAM.

The fault coverage results, presented for different seeds, in Figures 3 and 4 show clearly that, when including configuration LUT bits, single or multiple LSA fault models are too optimistic. Thus, the CSA fault model must be used in order to evaluate test effectiveness of FPGA implemented circuits – it evaluates structure and configuration. Additionally, the development of new methodologies to improve CSA coverage is required since after a significant number or random vectors the CSA FC does not approach 100%: the S5378 was simulated with 16 million vectors (in 560 seconds) and FC=88,89%.

Table 2. Fault lists and simulation times.

	CSA		Single LSA (collapsed)		Multiple LSA (collapsed)	
	# faults	Time [s]	# faults	Time [s]	# faults	Time [s]
C7552	6065	221	4243	121	26720	297
S5378	5205	194	3843	91	22560	204

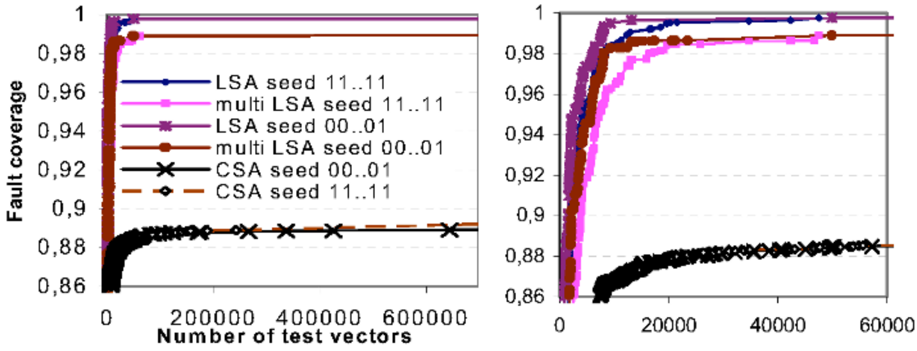


Fig. 3. S5378 HFS FC results with 1 million test vectors (zoomed on the right).

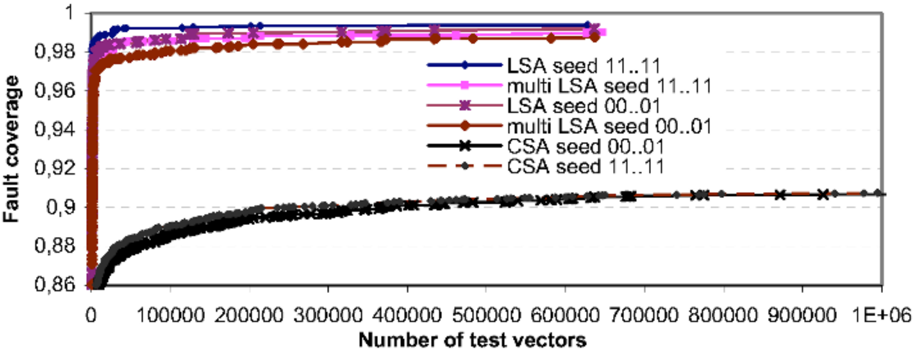


Fig. 4. C7552 FC HFS results with 1 million test vectors.

Figures 3 and 4 show also that the selection of seeds for BIST purposes can be efficiently accomplish using HFS.

6 Conclusions

Single or multiple LSA fault model in LUT terminals leads to optimistic results when the configuration of LUTs must also be evaluated. The CSA fault model, including each possible bit flip in each used LUT position is a much more demanding model and can be used to evaluate the test effectiveness of FPGA implemented circuits: evaluating the test of their structure and configuration.

BIST test effectiveness evaluation of FPGA cores using software fault simulation tools is a costly process. Hardware fault simulation can, very efficiently evaluate test quality even with more demanding models such as the CSA (the S5378 benchmark is simulated with 5205 faults and 16 million vectors in 560 seconds).

The LUT extraction and partial reconfiguration processes were detailed in this work.

Additional research in BIST methodologies is required in order to increase CSA fault coverages.

Acknowledgement. This work has been partially funded by FCT (Portugal), under the project POCTI/41788/ESE/2001.

References

[1] M.L. Bushnel, V.D. Agrawal, “Essentials of Electronic Testing for Digital Memory and Mixed-Signal VLSI Circuits”, Kluwer Academic Pubs., 2000.

[2] Ch. E. Stroud, “A Designer’s Guide to Built-In Self Test”, Kluwer Academic Pubs., ISBN 1-4020-7050-0, 2002.

[3] John Emmert, Stanley Baumgart, Pankaj Kataria, Andrew Taylor, Charles Stroud, Miron Abramovici, “On-Line Fault Tolerance for FPGA Interconnect with Roving STARs”, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'01), Oct. 2001.

[4] Manuel G. Gericota, Gustavo R. Alves, Miguel L. Silva, José M. Ferreira, "On-line Defragmentation for Run-Time Partially Reconfigurable FPGAs", Proc. of the 12th Int. Conf. on Field Programmable Logic and Applications (FPL'2002), pp. 302-311, Sept. 2002.

[5] T.M. Niermann, W. T. Cheng, J. H. Patel, "PROFS: A fast, memory-efficient sequential circuit fault simulator", IEEE Trans. Computer-Aided Design, pp.198-207, 1992.

[6] E.M. Rudnick, J.H. Patel, "Overcoming the Serial Logic Simulation Bottleneck in Parallel Fault Simulation", Proc. of the IEEE International Test Conference (ITC), pp. 495-501, 1997.

[7] F.M. Gonçalves, M.B. Santos, I.C. Teixeira and J.P. Teixeira, "Design and Test of Certifiable ASICs for Safety-critical Gas Burners Control", Proc. of the 7th. IEEE Int. On-Line Testing Workshop (IOLTW), pp. 197-201, July, 2001.

[8] A. Parreira, J. P. Teixeira,, M.B. Santos, "A Novel Approach to FPGA-based Hardware Fault Modeling and Simulation", Proc. of the Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS), pp. 17-24, April, 2003.

[9] R.W.Wieler, Z. Zhang, R. D. McLeod, "Simulating static and dynamic faults in BIST structures with a FPGA based emulator", Proc. of IEEE Int. Workshop of Field-Programmable Logic and Application, pp. 240-250, 1994.

[10] K. Cheng, S. Huang, W. Dai, "Fault Emulation: A New Methodology for Fault Grading", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 10, pp1487-1495, October 1999.

[11] Shih-Arn Hwang, Jin-Hua Hong and Cheng-Wen Wu, "Sequential Circuit Fault Simulation Using Logic Emulation", IEEE Transations on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 8, pp. 724-736, August 1998.

[12] P.Civera, L.Macchiarulo, M.Rebaudengo, M.Reorda, M.Violante, "An FPGA-based approach for speeding-up Fault Injection campaigns on safety-critical circuits", IEEE Journal of Electronic Testing Theory and Applications, vol. 18, no.3, pp. 261-271, June 2002.

[13] M.B. Santos, J. Braga, I. M. Teixeira, J. P. Teixeira, "Dynamic Fault Injection Optimization for FPGA-Based Hardware Fault Simulation", Proc. of the Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS), pp. 370-373, April, 2002.

[14] M. Abramovici, P. Menon, "Fault Simulation on Reconfigurable Hardware", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 182-190, 1997.

[15] M. Abramovici, P. R. Menon, D. T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation", IEEE Design Automation Conference, pp. 468 - 474 , 1984.

[16] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, O. Lepape, "Serial fault simulation", Proc. Design Auomation Conference, pp. 801-806, 1996.

[17] L.Antoni, R. Leveugle, B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp.405-413, October 2000.

[18] L.Antoni, R. Leveugle, B. Fehér, "Using Run-Time Reconfiguration for Fault Injection Applications", IEEE Instrumentation and Measurement Technology Conference, vol. 3, pp.1773-1777, May 2001.

[19] L.Antoni, R. Leveugle, B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 245-253, 2002.

[20] S. Guccione, D. Levi, P.Sundararajan, "Jbits: A Java-based Interface for Reconfigurable Computing", Proc. of the 2nd Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), pp. 27, 1999.

[21] E. Lechner, S. Guccione, “The Java Environment for Reconfigurable Computing”, Proc. of the 7th International Workshop on Field-Programmable Logic and Applications (FPL), Lecture Notes in Computer Science 1304, pp.284-293, September 1997.

[22] P.Sundararajan, S.Guccione, D.Levi, “XHWIF: A portable hardware interface for reconfigurable computing”, Proc. of Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, SPIE 4525, pp.97-102, August 2001.

[23] Abílio Parreira, J. P. Teixeira and Marcelino Santos, “Built-In Self-Test Preparation in FPGAs”, accepted for pub. in the Proc. of the Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS), April 2004.

[24] Xilinx Inc., “Virtex-E 1.8V Field Programmable Gate Arrays”, Xilinx DS022, 2001.

[25] Xilinx Inc., “Virtex Series Configuration Architecture User Guide”, Application Note: Virtex Series, XAPP151 (v1.5), September 27, 2000.

[26] F. Brglez, D. Bryan, K. Kominski, “Combinational Profiles of Sequential Benchmark Circuits”, Proc. Int. Symp. on Circuits and Systems (ISCAS), pp. 1229-34, 1989.

Solving SAT with a Context-Switching Virtual Clause Pipeline and an FPGA Embedded Processor

C.J. Tavares, C. Bungardean, G.M. Matos, and J.T. de Sousa

INESC-ID/IST-Technical University of Lisbon/Coreworks, Lda
R. Alves Redol, 9, 1000-029 Lisboa, Portugal[1]
jose.desousa@inesc-id.pt

Abstract. *This paper proposes an architecture that combines a context-switching virtual configware/software SAT solver with an embedded processor to promote a tighter coupling between configware and software. The virtual circuit is an arbitrarily large clause pipeline, partitioned into sections of a number of stages (hardware pages), which can fit in the configware. The hardware performs logical implications, grades and select decision variables. The software monitors the data and takes care of the high-level algorithmic flow. Experimental results show speed-ups that reach up to two orders of magnitude in one case. Future improvements for addressing scalability and performance issues are also discussed.*

1 Introduction

Definitions and motivation. The satisfiability (SAT) problem — given a Boolean formula $F(x_1,x_2, ..., x_n)$, find an assignment of binary values to (a subset of the) variables, so that F is set to 1, or prove that no such assignment exists — is a central, NP-complete computer science problem [1], with many applications in a variety of fields. Typically F is expressed as a product-of-sums, which is also called conjunctive normal form (CNF). The terminology is reviewed via an example: in the formula $F=(x_1+x_2)(\neg x_1+ x_2)(x_1+\neg x_2)$, we have two variables, x_1 and x_2, and three clauses, each with two literals; the literals in the third clause are x_1 and $\neg x_2$, where x_1 is a non-inverted literal and $\neg x_2$ is an inverted literal. The assignment (x_1=1, x_2=1) is a satisfying assignment, as it sets $F=1$. Hence F is satisfiable. The formula $G=(x_1+x_2)(\neg x_1+ x_2)(x_1+\neg x_2)(\neg x_1+ \neg x_2)$ is unsatisfiable. The number of variables in a formula is denoted n and the number of clauses m. A k-clause is a clause that has k literals. A k-SAT problem is one where clauses have at most k literals.

Previous work. In recent years, solving SAT using reconfigurable hardware (configware) has become a major challenge for Reconfigurable Computing (RC) experts. It is well known that, to become a generally accepted computing paradigm, RC has to prove itself able to tackle important computer science problems such as

[1] This work has been partially funded by FCT (Portugal), Project POSI/CHS/34562/2000.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 344–353, 2004.

SAT, competing with advanced software SAT solvers such as GRASP [4], CHAFF [5] and BERKMIN [6].

Several research groups have recently explored different approaches to implement SAT on configurable hardware [8-17], as an alternative to software SAT solvers. The satisfiers implement variations of the classical full search Davis -Putnam (DP) SAT, algorithm [18]. More recently, incomplete, local search SAT algorithms like WSAT or GSAT have also been contemplated with configware implementations [14,19]. An interesting survey comparing the various approaches that have been proposed in the literature is given in [20].

The most important problems addressed by the various proposals are the following: (1) the method used to select the next decision variable and its value to be tried [8,16,17]; (2) the compilation time spent in preparing the FPGA -based circuit to be emulated [14,16,17]; (3) the ability to solve problems of an arbitrary large size [12,16,17]; software-hardware partitioning [13,14,16,17].

Main contributions and organization of the paper. This paper presents a hard evidence analysis of our approach to configware-software SAT solving. The main contributions are the following:

1. Proposes the use of an embedded processor (Microblaze from Xilinx [21]) to create a tighter coupling between configware and software, eliminating expensive communications between the two.
2. Proposes the use of a decision variable pipelined comp arator tree, to select the next decision variables.
3. Publishes the first experimental results obtained with the actual (non simulated) configware/software SAT solver system proposed in [16] and refined in [22], which is also used to indirectly derive results for the architecture proposed in this paper.

The remainder of this paper is organized as follows. Section 2 presents an overview of the predecessor of the SAT solver system being proposed. Section 3 presents the new system. Section 4 presents experimental results and their analyses. Finally, Section 5 outlines our conclusions and describes our current and future work.

2 Overview of the Predecessor System

Our current system evolved from a previous architecture already published in [16,22,23], which is summarized in Figure 1. The SAT solver runs partly in software and partly in configware. The software runs on a host computer and communicates with the configware via the PCI bus. The configware resides in a board containing an FPGA, a control CPLD and two single port RAMs, M1 and M2. After the configware architecture is loaded in the FPGA, it may be used to process any SAT problem instance, totally avoiding hardware instance-specific computation. The architecture can be outlined as a virtual pipeline of clause data processors. Each clause data processor represents a clause of the SAT instance. The virtual circuit is partitioned in several hardware pages. Each page implements a pipeline section of D stages and M clauses per stage. The board memories store the programming of the hardware pages (context), which specify the clauses in each page and their data, and the latest state of the SAT variables. Each memory position has N variables, and each variable is represented by a K-bit word (K is even), having 2 fields, P_0 and P_1, of $K/2$ bits. L memory addresses are used to store a total of LN variables. Therefore, the configware architecture is perfectly characterized by the parameters *(D,K,L,M,N)*. The processing consists of streaming the variables through the pipeline, back and forth between M1 and M2.

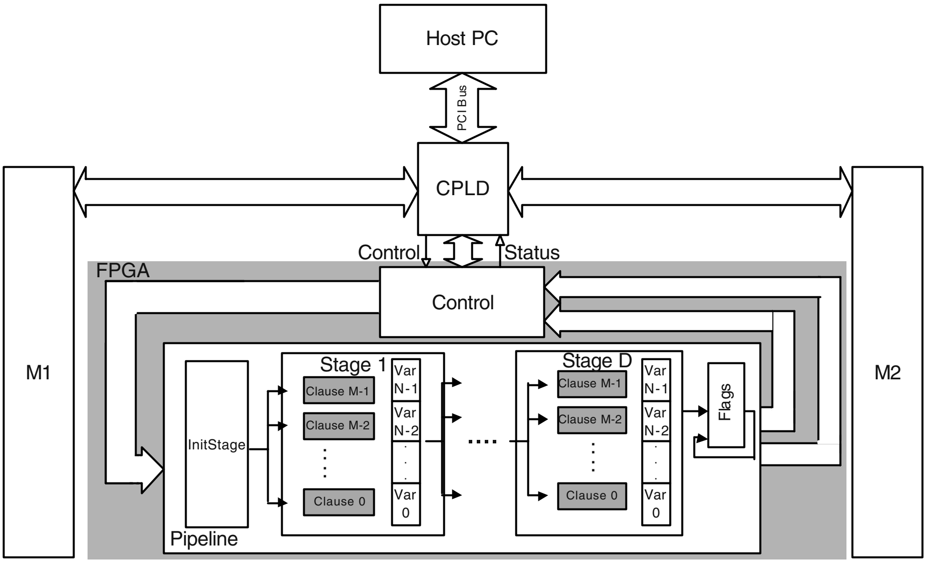


Fig. 1. High-level view of the previous system.

The algorithm starts when the software reads the SAT problem instance from a file in the CNF format, and maps it to the configware structure as a 3-SAT problem. If the original problem is k-SAT (k>3), it is converted into a 3-SAT problem. The whole formula compilation process runs in polynomial time, in a matter of seconds, much faster than any FPGA compilation flow. If the resulting circuit model is larger than the available configware capacity, it is partitioned in P virtual hardware pages able to fit in the configware. Thus the number of stages of the virtual clause pipeline is PD.

After the hardware pages are generated, the processing of variables in the clause pipeline can start. While moving the variables through the clause pipeline back and forth between M1 and M2, the values of their fields P_0 and P_1 are updated, until they reflect the number of clauses that would be satisfied if the variable had the value '0' or '1', respectively. This is because each field P_b is incremented whenever the variable is processed by one of its unresolved clauses that is satisfied for value b. The incrementing saturates when P_b reaches the value

$$P_b\,|_{MAX\_SCORE} = 2^{K/2} - 2$$

The maximum possible value, $P_b|_{\mathrm{ASSIGNED}}$, is reserved to represent an implied or assigned variable:

$$P_b\,|_{ASSIGNED} = 2^{K/2} - 1$$

Our SAT solver implements a variation of the DP algorithm, and can be thought of as having three engines: the deduction, diagnosis and decision engines. The deduction engine assigns variables in unit clauses (computes logical implications) and grades unassigned variables according to the heuristic described above. When one or more clauses are falsified, the deduction engine reports the existence of conflicting assignments (conflicts). The diagnosis engine checks if a solution has been found, or if the last decision variable assignment has resulted in a conflict. If a conflict happened, the decision engine returns to the last decision variable (chronological backtracking), and sets it to its untried value. If no conflict is found the decision engine chooses the variable with the best heuristic score to be the next decision variable. If after a conflict there is no backtrack point then the formula is unsatisfiable. A flowchart summarizing the operation of the system is shown in Figure 2, where the filled areas represent tasks implemented in configware and the unfilled areas represent tasks implemented in software.

The operation of the virtual hardware scheme is as follows. Suppose the variables rest initially in memory M1. The first virtual hardware page is programmed, and all variables are streamed from M1, processed through the virtual hardware page, and stored in M2. If no conflict is detected, the next hardware page is loaded, and the processing of variables continues now from M2 back to M1. This process goes on for all virtual hardware pages sequentially, so that all sections of the virtual clause pipeline get to process all variables. Running all hardware pages on all variables is denoted a *pass*. During a pass new implications are generated and new clauses are satisfied - these are called *clause events*. For as long as new clause events are generated in a pass, another one is started, until no more events arise (this situation is denoted *stasis*) or a conflict is found. For the variables not yet assigned, P_0 and P_1 are recomputed during each pass so that their values are up to date when stasis is reached. After stasis, the *configware* informs the software on the latest location of the variables, either memory block M1 or M2. Then the software runs the diagnosis and decision engines.

3 The New System

After implementing and evaluating the system described in the previous section, we were not surprised to find out that its performance was far from what the simulation results in [23] had predicted — this is usually the case with a first prototype. Hence, we proceeded to analyse the causes of the discrepancies between simulated and actual results, and two major causes have been identified:

1. The communication between software and hardware was slow due to the latency of the PCI bus.
2. The software processing time was high, since the decision engine required all variables to be read, to find the one with the highest heuristic score.

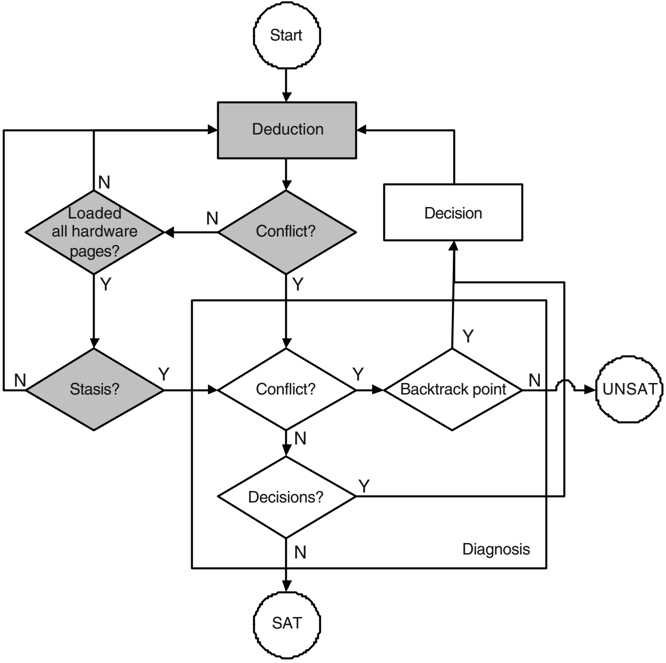


Fig. 2. Configware/Software SAT Solver Algorithm

To address these problems we came up with the improved architecture depicted in Figure 3, whose main new features are:

1. An embedded processor, MicroBlaze (MB) from Xilinx [21], was introduced to create a tight coupling between software and configware.
2. Comparator stages were introduced in the pipeline to select the variable with the best heuristic score, relieving the software of this burden.

MB uses the On-chip Peripheral Bus (OPB, inherited from IBM's CoreConnect infrastructure) to communicate with the clause pipeline, and to access the memories via the control CPLD. This way, MB and the clause pipeline share the same memory, and *there is no need to move the variables elsewhere*. In the predecessor system, where the host PC was running the software, all variables were transferred via DMA to the PC's central memory to be processed there, and then transferred back to the board memory. This had to be done for every decision variable, which, due to the high latency of the PCI bus, was still less expensive than accessing the variables one by one from the board memory.

To select the next decision variable a tree of variable comparators has been introduced. To preserve the frequency of operation, each level of the tree is placed in a pipeline stage. The number of levels (height of the tree) is *$log_2(2N)$*, which creates a constraint for the minimum pipeline depth *D*. However, since *N* is not a large number, the tree is quite short anyway, and this new constraint is irrelevant.

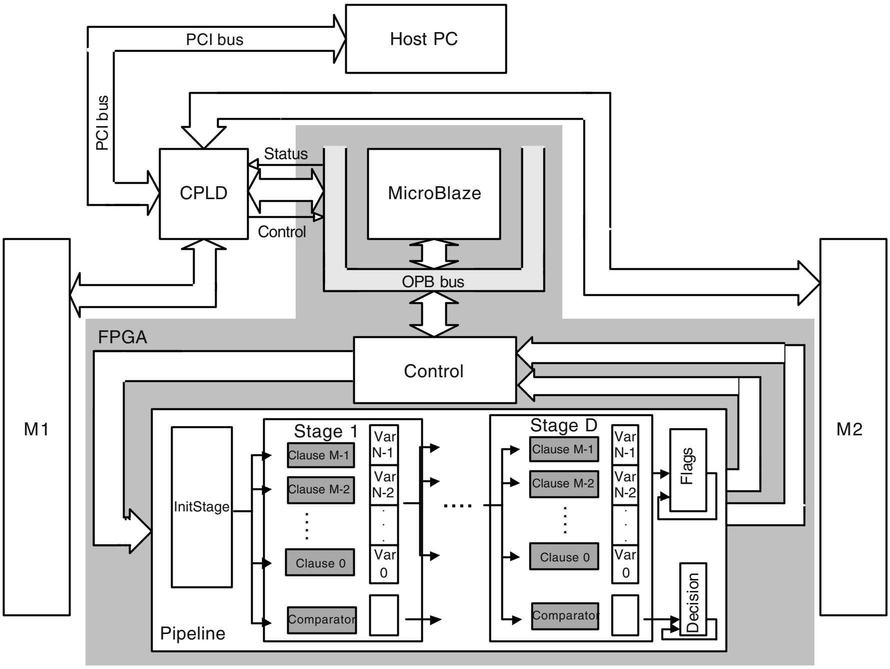


Fig. 3. High-level view of the proposed system.

The compilation of the SAT problem instance is still performed on the host PC for convenience and speed, since it is only done once as a pre-process step. Work to incorporate the decision variable comparator tree and the Xilinx's MicroBlaze soft processor is currently under way.

4 Results

All experimental results have been obtained using the system described in Section 2, whose prototype has been finished recently. The results for the proposed architecture have been derived by carefully measuring the DMA communication time and the elapsed time of the decision variable selection software routine, and subtracting these two terms from the total elapsed time obtained with the predecessor system of Section 2. The results obtained in this way are valid for an FPGA 30% larger, which is the hardware overhead estimated for the added variable comparator tree and the MicroBlaze embedded processor. This is no problem since FPGA devices much larger than the ones used in our experiments are commercially available.

Experimental setup. The software runs on a *Linux Suse 8.0* host PC with a Pentium 2 processor, at 300.699 MHz and 192 Mbytes of memory. The configware architecture is implemented in a Celoxica's RC1000 board [2] with *PCI* interface, featuring a XCV2000E device with 4 SRAM banks of 2 Mbytes and 32 bits wide. The memory blocks M1 and M2 are implemented using 2 SRAM banks each, so the variables are accessed as 64-bit words. The clause pipeline programming data (hardware pages or contexts) are stored as 128-bit words in the 4 SRAM banks simultaneously. The configware architecture is characterized by the parameters *D*=17, *K*=8, *L=1024*, *M*=4, *N*=7, as described in Section 2 and optimized according to [22]. Thus the system implemented can process SAT formulae of complexity up to 7168 variables 165036 clauses. The hardware occupies 96% of the FPGA area, so it has a complexity of 1.92M system gates and works at 40 MHz.

	A0 and A1			GRASP		
Example	Variables	Clauses	Decisions	Variables	Clauses	Decisions
aim-50-1_6-no-2	50	80	10141	50	80	13390
aim-50-1_6-no-3	50	80	37363	50	80	100471
aim-50-1_6-no-4	50	80	2883	50	80	2332
aim-50-2_0-yes1-3	50	100	2022	50	100	2170
aim-50-2_0-yes1-4	50	100	135	50	100	6164
aim-100-1_6-yes1-1	100	160	1287235	100	160	14384
aim-100-1_6-yes1-2	100	160	2119121	100	160	3916671
dubois20	60	160	25165823	60	160	12582911
ssa432_3	561	1405	3911	435	1027	3115
hole6	63	196	5883	42	133	3245
hole7	96	324	49405	56	204	21420
hole8	126	459	674595	72	297	378343
hole9	150	595	7520791	90	415	4912514

Table 1. Benchmark SAT instances used.

Experimental results. Our results have been obtained using a subset of the well-known benchmark set from DIMACS [7]. The results are compared to those obtained with GRASP, a well-known and publicly available SAT solver. Its options have been set to implement the same DP search algorithm we use in our system. Our *k*-SAT to 3-SAT decomposition technique augments the size of the formula, which may alter the number of decisions comparatively to using the original formula; GRASP is always run on the original formula. Table 1 shows the number of variables, clauses and decisions when running our algorithms, denoted A0 and A1, and GRASP. Note that a larger formula does not necessarily mean more decisions, since a different direction of the search may change the number of decisions dramatically. In Table 2, execution time results are presented. TGRASP is the total time taken by GRASP for each instance. TA0 is the total time taken by our predecessor system, and TA1 is the time taken by the system proposed in this paper. SUA1 is the overall speed-up, and SUA1PD is the speed-up per decision.

These results show that the predecessor system (algorithm A0) can not obtain any speed-ups compared to GRASP (see columns TGRASP and TA0), while the proposed system (algorithm A1) can in fact obtain an acceleration against GRASP (see columns TGRASP, TA1 and SUA1). For the *aim-50-2_0-yes1-4* example the overall speed-up against GRASP is almost 250, which is a 2 orders of magnitude acceleration. However, comparing the execution times without taking in consideration the number of decisions shown in Table 1 is imprecise. In fact, the *aim-50-2_0-yes1-4* benchmark has a significantly lower number of decisions (135) when using the 3-SAT formula (A0 and A1) than when using the original formula (6164 decisions with GRASP). Therefore, a more fair comparison is to use the execution time per decision rather than the total elapsed time. These results are shown in column SUA1PD, which shows more modest speed-ups reaching one order of magnitude. On the other hand, many more examples show speed-ups greater than one, when using the SUA1PD metric.

EXAMPLE	TGRASP	TA0	TA1	SUA1	SUA1PD
aim-50-1_6-no-2	1,830	3,461	0,340	5,382	4,076
aim-50-1_6-no-3	10,510	14,165	0,874	12,025	4,472
aim-50-1_6-no-4	0,250	1,264	0,248	1,008	1,246
aim-50-2_0-yes1-3	0,350	0,847	0,030	11,667	10,871
aim-50-2_0-yes1-4	1,000	0,220	0,004	249,988	5,475
aim-100-1_6-yes1-1	2,600	787,969	85,081	0,031	2,735
aim-100-1_6-yes1-2	614,310	1312,940	164,268	3,740	2,023
dubois20.cnf	1040,400	6751,870	561,789	1,852	3,704
ssa432_3.cnf	1,300	39,810	10,817	0,120	0,151
hole6.cnf	0,260	2,325	0,372	0,699	1,267
hole7.cnf	4,340	24,830	6,003	0,723	1,668
hole8.cnf	56,450	408,494	136,194	0,414	0,739
hole9.cnf	825,120	6225,630	1932,090	0,427	0,654

Table 2. Execution time results for GRASP, A0 and A1.

Comparing Tables 1 and 2 we can observe that the speed-ups drop with the size of the instance, reflected in the size of the virtual clause pipeline. The explanation for this is the still immature virtual hardware scheme that has been implemented. In our current approach for every new variable assignment all clauses are evaluated, no matter if the new variables assigned are present in the evaluated clauses or not. This is inefficient and makes the assignment evaluation time *O(mn)*. Ideally the number of evaluated hardware pages should depend only on their having clauses to update with new assignments. Also, all variables are updated in the process, when there is only need to update variables in clauses that have been updated themselves. We have current plans to optimize this aspect, which will considerably boost the performance and prevent degradation when the problem scales up.

5 Conclusion

In this paper we have proposed a novel architecture of a SAT solver that combines a configurable hardware device with a small size embedded processor. The configware device implements a section of a virtual clause pipeline circuit (hardware page). The large virtual circuit embodies the SAT instance to be solved, and is operated by context-switching, where each context is a hardware page and its data.

The configware computes logical implications, grades decision variables using a heuristic score, and selects the next decision variable based on this figure. The software manages the search process (decision tree).

Experimental results have been obtained using a host PC to implement the software, and an FPGA to implement the configware. The performance of the proposed architecture has been derived by subtracting the PCI communication time and the elapsed time of the decision variable selection routine from the total elapsed. Work to incorporate the MicroBlaze embedded processor and the proposed comparator tree to select the next decision variable is under way. Our results show that speed-ups up to 2 orders of magnitude can be obtained with the proposed system.

Future work. We now have an architecture flexible enough to implement sophisticated algorithmic improvements, such as non-chronological backtrack and clause addition, like in modern software SAT solvers.

References

1. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, pp. 19-151, 1997.
2. http://www.celoxica.com: RC1000 configware platform.
3. M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP Completeness", W.H. Freeman and Company, San Francisco, 1979.
4. J.M. Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability", IEEE Trans. Computers, vol. 48, n. 5, pp. 506-521, 1999.
5. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver", in Proc. 38th Design Automation Conference, 2001, pp. 530-535.
6. E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust SAT-solver", in Proc. Design, Automation and Test in Europe Conference, 2002, pp. 142-149.
7. http://www.intellektik.informatik.tudarmstadt.de/SATLIB/benchm.html: DIMACS challenge benchmarks.
8. T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving Satisfiability Problems Using Reconfigurable Computing", IEEE Trans. on VLSI Systems, vol. 9, no. 1, pp. 109-116, 2001.

9. P. Zhong, M. Martonosi, P. Ashar, and S. Malik, “Using Configurable Computing to Accelerate Boolean Satisfiability”, IEEE Trans. CAD of Integrated Circuits and Systems, vol.18, n. 6, pp. 861-868, 1999.
10. M. Platzner, “Reconfigurable accelerators for combinatorial problems”, IEEE Computer, Apr. 2000, pp. 58-60.
11. M. Abramovici and D. Saab, “Satisfiability on Reconfigurable Hardware”, in Proc. 7th Int. Workshop on Field-Programmable Logic and Applications, 1997, pp. 448-456.
12. M. Abramovici and J. T. de Sousa, “A SAT solver using reconfigurable hardware and virtual logic”, Journal of Automated Reasoning, vol. 24, n. 1-2, pp. 5-36, 2000.
13. Dandalis and V. K. Prasanna, “Run-time performance optimization of an FPGA-based deduction engine for SAT solvers”, ACM Trans. on Design Automation of Electronic Systems, vol. 7, no. 4, pp. 547-562, Oct. 2002.
14. P. H. W. Leong, C. W. Sham, W. C. Wong, H. Y. Wong, W. S. Yuen, and M. P. Leong, “A Bitstream Reconfigurable FPGA Implementation of the WSAT algorithm”, IEEE Trans. On VLSI Systems, vol. 9, no. 1, pp. 197-201, 2001.
15. M. Boyd and T. Larrabee, “ELVIS – a scalable, loadable custom programmable logic device for solving Boolean satisfiability problems”, in Proc. 8th IEEE Int. Symp. on Field-Programmable Custom Computing Machines, 2000.
16. J. de Sousa, J. P. Marques-Silva, and M. Abramovici, “A configware/software approach to SAT solving”, in Proc. 9th IEEE Int. Symp. on Field-Programmable Custom Computing Machines, 2001.
17. Skliarova and A. B. Ferrari, “A SAT Solver Using Software and Reconfigurable Hardware”, in Proc. the Design, Automation and Test in Europe Conference, 2002, p. 1094.
18. M. Davis and H. Putnam, “A Computing Procedure for Quantication Theory” ACM journal, vol. 7, 1960, pp. 201-215".
19. R. H. C. Yap, S. Z. Q. Wang, and M. J. Henz, “Hardware Implementations of Real-Time Reconfigurable WSAT Variants”, in Proc. 13th Int. Conf. on Field-Programmable Logic and Applications, Lecture Notes in Computer Science, vol. 2778, Springer, 2003. pp. 488-496.
20. Skliarova and A. B. Ferrari, “”, in Proc. 13th Int. Conf. on Field-Programmable Logic and Applications, Lecture Notes in Computer Science, vol. 2778, Springer, 2003. pp. 468-477.
21. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze: MicroBlaze Soft Processor.
22. N. A. Reis and J. T. de Sousa, “On Implementing a Configware/Software SAT Solver”, in Proc. 10th IEEE Int. Symp. Field-Programmable Custom Computing Machines, 2002, pp. 282-283.
23. R. C. Ripado and J. T. de Sousa, “A Simulation Tool for a Pipelined SAT Solver”, in Proc. XVI Conf. on Design of Circuits and Integrated Systems, Nov. 2001, pp. 498-503.

Evaluating Fault Emulation on FPGA

Peeter Ellervee, Jaan Raik, Valentin Tihhomirov, and Kalle Tammemäe

Department of Computer Engineering, Tallinn University of Technology
Raja 15, 12618 Tallinn, Estonia
{lrv, kalle}@cc.ttu.ee, jaan@pld.ttu.ee,
valentin@abelectron.com

Abstract. We present an evaluation of accelerating fault simulation by hardware emulation on FPGA. Fault simulation is an important subtask in test pattern generation and it is frequently used throughout the test generation process. In order to evaluate possible simulation speed possibilities, we made a feasibility study of using reconfigurable hardware by emulating circuit under analysis together with fault insertion structures on FPGA. Experiments showed that it is beneficial to use emulation for circuits/methods that require large numbers of test vectors, e.g., sequential circuits and/or genetic algorithms.

1 Introduction

Test generation is today one of the most complicated and time-consuming problems in the domain of digital design. The more complex the electronics systems are getting, the more important will be the problems of test and design for testability, as the costs of verification and testing are becoming the major component of design and manufacturing costs of a new product. This fact makes the research in the area of testing and diagnosis of integrated circuits (IC) a very important topic for both industry and academy.

As the sizes of circuits grow, so do the test costs. Test costs include not only the time and resources spent for testing a circuit but also time and resources spent to generate suitable test vectors. The most important sub-task of any test generation approach is the suitability analysis of a given set of test vectors. Many techniques exist to perform such an analysis. Circuit structure analysis gives good results but it is rather time consuming. Fault simulation is the most often used way of analysis and it is frequently applied throughout the entire test pattern generation cycle. Many techniques exist to speed up simulation, e.g., simulating multiple bits in parallel [1] or replacing the circuit model with a corresponding Binary Decision Diagram (BDD) [2]. Unfortunately, this approach is limited by the bit-width of processors, which limits the number of bits processed in parallel, and it requires additional circuit optimization to make use of the potential parallelism.

At the same time, reconfigurable hardware, e.g., FPGAs and PLDs, has been found useful as system-modeling environments (see, e.g., [3]). This has been made possible by the availability of multi-million-gate FPGAs, e.g., Virtex series from Xilinx. For academic purposes, both for research and education, cheaper devices with rather large capacity, e.g., new Spartan devices, can be used.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 354–363, 2004.

Taking into account the advances in the field of reconfigurable hardware, the next natural step to speed up fault simulation would be to emulate circuits on a chip. The availability of large devices allows to implement not only the circuit under test with fault models but also test vector generator and result analysis circuits on a single reconfigurable device. To study the possibility of replacing fault simulation with emulation, we first had to solve some essential issues – how to represent logic faults, how to feed the test vectors into the circuit, and how to analyze the results. Then we created the corresponding experimental environment, and finally performed experiments with some benchmark circuits.

The experiments showed that for circuits that require large numbers of test vectors, e.g., sequential circuits, it is beneficial to replace simulation with emulation. More work is needed to integrate the hardware part with the software part of the test generation environment. The emulation approach is planned for use in cooperation with diagnostic software Turbo Tester.

The paper is organized as follows. In Section 2, related work is discussed. The diagnostic software, Turbo Tester, is described in Section 3. The emulation environment is introduced in Section 4. In Section 5, the results of experiments are presented. Section 6 is dedicated for conclusions.

2 Related Work

A number of works on fault emulation for combinational circuits has been published in the past. They rely either on fault injection (see, e.g., [4, 5]) or on implementing specific fault simulation algorithms in hardware [6]. Recently, acceleration of combinational circuit fault diagnosis using FPGAs has been proposed in [7]. However, the need for hardware fault emulation has been driven mainly by large sequential designs whose fault grading run-times could extend to several years.

In many of the papers for sequential circuits, faults are injected either by modifying the configuration bitstream while the latter is being loaded into the device [8] or by using partial reconfiguration [9, 10, 11]. This kind of approach is slow due to the run-time overhead required by multiple reconfigurations. Other options for fault injection are shift-registers and/or decoders (used in this paper). A paper relying on the shift-register-based method was presented in [12]. Shift-registers are known to require slightly less hardware overhead than the decoders do. However, in [12] injection codes and test patterns are read from a PC and only 10-30 times speed-up in comparison to the software based fault simulation is achieved. Furthermore, it is unclear whether synthesis times are included to the results or not.

In addition to merely increasing the speed of fault simulation, the idea proposed in current paper can be used for selecting optimal Built-In Self-Test (BIST) structures. In an earlier paper [13] a fault emulation method to be used for evaluating the Circular Self-Test Path (CSTP) type BIST architectures has been presented. Different from the current approach, no fault collapsing was carried out and fault-injecting hardware was inserted to each logic gate of the circuit to be emulated.

In this paper, we propose an efficient FPGA-based fault emulation environment for sequential circuits. The environment has interfaces to the digital test package Turbo Tester [14, 15] developed at Tallinn University of Technology and described in Section 3. The main novelty of the emulation approach lies in implementing multiplexers and a decoder for fault injection that, unlike the shift register based injection, allows to activate faults in an arbitrary order. This feature is highly useful when applying the presented environment in emulating the test generation process. In addition, we use an on-chip input pattern generator as opposed to loading the simulation stimuli from the host computer. It is also important to note that the time spent for emulator synthesis is included to the experimental results presented in Section 5.

3 Overview of Turbo Tester

Turbo Tester (TT) is a test software package developed at the Department of Computer Engineering of Tallinn University of Technology [14, 15]. The TT software consists of the following test tools: test generation by different algorithms (deterministic, random and genetic), test set optimization, fault simulation for combinational and sequential circuits, testability analysis and fault diagnosis. It includes test generators, logic and fault simulators, a test optimizer, a module for hazard analysis, BIST architecture simulators, design verification and design error diagnosis tools (see Fig. 1). TT can read the schematic entries of various contemporary VLSI CAD tools that makes it independent of the existing design environment. Turbo Tester versions are available for MS Windows, Linux, and Solaris operating systems.

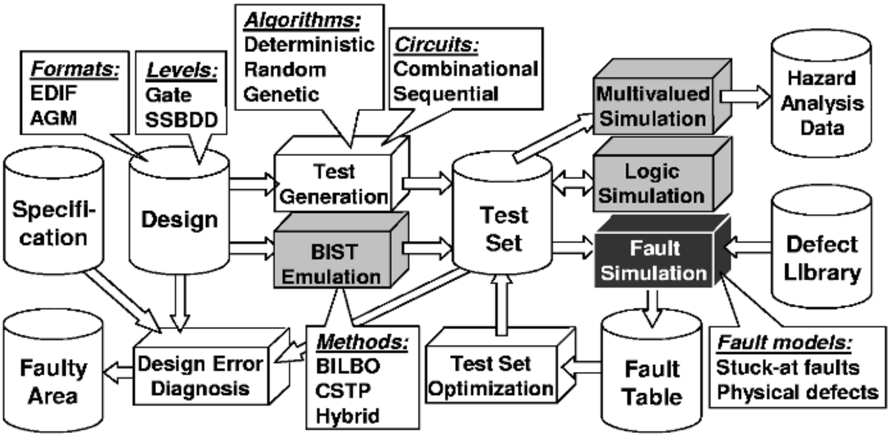


Fig. 1. Turbo Tester environment

The main advantage of TT is that different methods and algorithms for various test problems have been implemented and can be investigated separately of each other or working together in different work flows.

Model Synthesis. All the tools of TT use Structurally Synthesized BDD (SSBDD) models as an internal model representation. TT includes a design interface generating SSBDD-s in AGM format from EDIF netlists. The set of supported technology libraries can be easily extended.

Test Generation. For automatic test pattern generation (ATPG), random, deterministic and genetic test pattern generators (TPG) have been implemented. Mixed TPG strategies based on different methods can also be investigated. Tests can be generated both for combinational and sequential circuits.

Test Pattern Analysis. There are concurrent and parallel fault simulation methods implemented in the system. In current paper, we have experimented only with "Fault Simulation" part (black in Fig. 1). In the future, also the other simulation-related parts might be considered (gray in Fig. 1).

Test Set Optimization. The tool is based on static compaction approach, i.e. it minimizes the number of test patterns in the test set without compromising the fault coverage.

Multivalued Simulation. In Turbo Tester, multi-valued simulation is applied to model possible hazards that can occur in logic circuits. The dynamic behavior of a logic network during one single transition period can be described by a representative waveform on the output or simply by a corresponding logic value.

Design Error Diagnosis. After a digital system has been designed according to its specification, it might go through a refinement process in order to be consistent with certain design requirements, e.g., timing specifications. The changes introduced by this step may lead to undesired functional inconsistencies compared to the original design. Such design errors should be identified via verification.

Evaluation of Built-In Self-Test (BIST) Quality. The BIST approach is represented by applications for simulating logic BIST and Circular Self-Test Path (CSTP) architectures.

4 Emulation Environment

The emulation environment was created keeping in mind that the main purpose was to evaluate the feasibility of replacing fault simulation with emulation. Based on that, the main focus was put on how to implement circuits to be tested on FPGAs. Less attention was paid how to organize data exchange between hardware and Turbo Tester (or any other test software). For the first series of experiments, we looked at combinational circuits only. This could be done because when comparing how test vectors are fed into combinational and sequential circuits, the only principal difference is the need of set/reset signals for sequential circuits. Results of experiments with combinational circuits were presented in [5].

For sequential circuits, most of the solutions used for combinational circuits can be exploited. Few modifications were related to different testing strategies of combinational and sequential circuit. For instance, an extra loop was needed for the controller because sequential circuits require not a single input combination but a sequence consisting of tens or even hundreds of input combinations. Also, instead of hard-coded test sequence generators and output analyzers, loadable modules were introduced.

Before building the experimental environment, we had first to solve some essential issues – fault insertion, test vector generation, output data analysis and exchange with software part, and design flow automation. The issues, used solutions and discussions are presented below.

Fault insertion. The main problem here was how to represent non-logic features – faults – in such a way that they can be synthesized using standard logic synthesis tools. Since most of the analysis is done using stuck-at-one and stuck-at-zero fault models, the solution was obvious – use multiplexers at fault points to introduce logic one or zero, or pass through intact logic value. Also, since a single fault is analyzed at a time typically, decoders were introduced to activate faults. In the right side of Fig. 2, a fault point and multiplexer inserted into that point are shown. Some of the decoders are shown in the left side of Fig. 2. The fault insertion program takes TT generated netlist and list of faults as input, and outputs modified netlist. It also adds distributed fault selection decoder(s) and extra ports to control fault modeling.

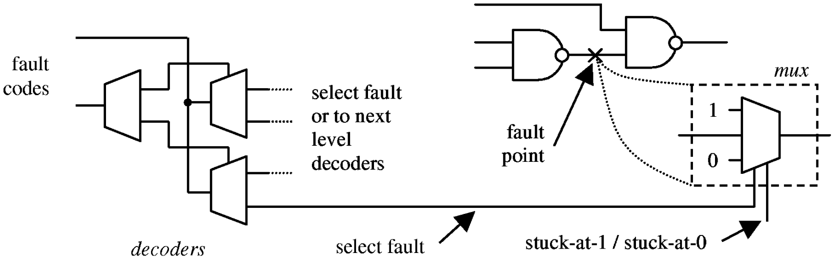


Fig. 2. Fault point insertion and fault activation decoders

The inserted multiplexers will add extra gates to the circuit and will make it slower. It should be noted that the increase both in size and delay is 5 to 10 times depending on the size of the original circuit and the number of fault points (see also Table 1). It is not a problem for smaller circuits but may be too prohibitive for larger designs – the circuit may not fit into target FPGA. Inserting not all of the fault points but only selected ones can solve this problem. Selection algorithm, essentially fault set partitioning, is a subject of future research.

Compared against shift-register based fault injection approaches (see, e.g., [12]), the use of multiplexers has both advantages and disadvantages. The main disadvantage is small increase both area and delay of the circuit. Although the delay increase is only few percents, execution time may increase significantly for long test cycles. The main advantage is that any fault can be selected in a single clock cycle, i.e., there is no need to shift the code of a fault into the proper register. Combining both approaches may be the best solution and one direction of future work will go in that direction.

Test vector generation and output data analysis. It is clear that not all generation and analysis approaches can be used, especially these ones that use large lookup tables and/or complex algorithms. Here we relied on a well-known solution for BIST

– Linear Feedback Shift Register (LFSR) is used both for input vector generation and output correctness analysis (see, e.g., [14, 16]). This is very beneficial because LFSR structures have been thoroughly studied. Also, their implementation in hardware is very simple and regular – only flip-flops and XOR gates. This simplifies data exchange with the software part – only seed and feedback polynomial vectors are needed to get a desired behavior. Output correctness analysis hardware needs first to store the expected output signature and then to report to the software part whether the modeled fault was detected or not. Fig. 3 illustrates a stage of used LFSRs. The input 'coefficient' is used for feedback polynomial. The input 'result' is used only for result analysis and is connected to zero for input vector generation.

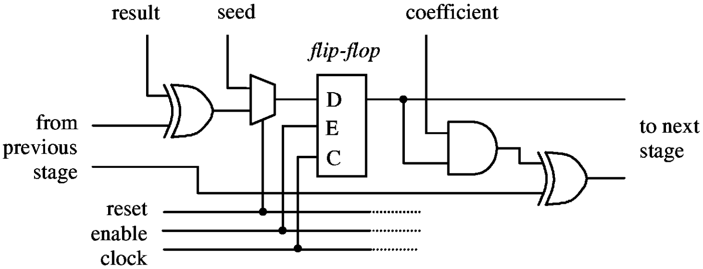


Fig. 3. Single stage of LFSRs

Design flow automation was rather easy because of the modular structure of the hardware part. All modules are written in VHDL that allows to parameterize design units. The structure consists of the following parts (see also Fig. 4):

- CUT – circuit under test, generated by the fault insertion program;
- CUT-top – wrapper for CUT to interface a specific module with generic test environment without changing the later, generated by wrapper program;
- CUT-pkg – parameters of CUT like the number of inputs and outputs, the length of test sequence and the number of fault points, generated by the wrapper program;
- Two LFSRs – one for test vector generator and one for output signature calculation; a generic VHDL module, used two times;
- Three counters – one to count test vectors, one to count test sequences (not used for combinational units), and one to count modeled faults; a generic VHDL module, used three (or two) times;
- Test bench with state machine (FSM) to connect all sub-modules, to initialize LFSRs and counters, and to organize data exchange with the external interface; a generic VHDL module; algorithms implemented by the state machine are depicted in Fig. 5; and
- Interface to organize data exchange between the test bench and the software part, FPGA type/board and host PC platform/OS dependent.

The interface is currently implemented only in part as further studies are needed to define data exchange protocols between hardware and software parts. The design flow for hardware emulator consists of three steps:

1. Fault insertion based on netlist and fault list;

2. Wrapper and parameter file generation based on the modified netlist; and
3. FPGA bit-stream generation using available logic synthesis tools, e.g., ISE from Xilinx [17].
4. Resulting bit-streams are loaded into FPGAs and controlling state machines are activated. In principle, any suitable FPGA board can be used but supporting interfaces should be developed.

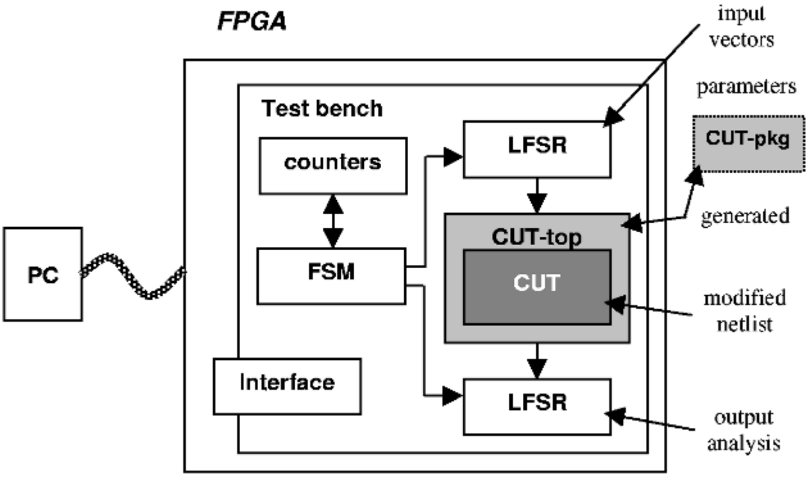


Fig. 4. Emulation environment structure

5 Results of Experiments

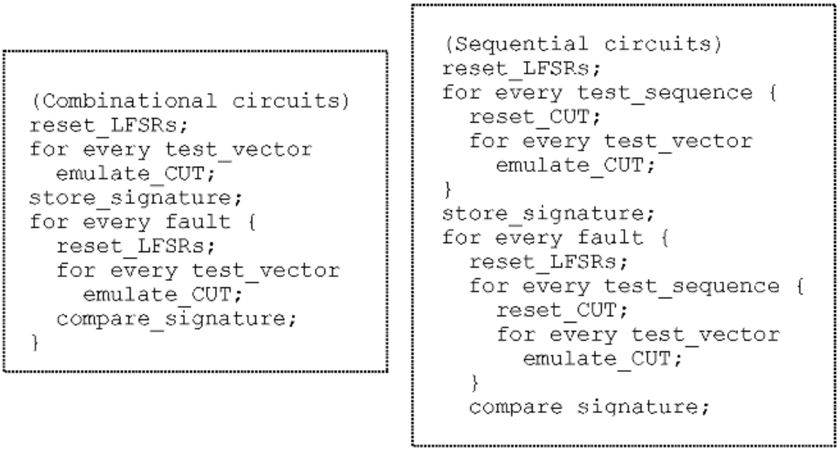
For experiments, two FPGA boards were used:

- A relatively cheap XSA-100 board with Spartan II chip XC2S100 with 600 CLBs from XESS [18], that can be used with any PC; and
- A powerful RC1000-PPE board with Virtex chip XCV2000E with 9600 CLBs and supporting software from Celoxica [19].

The first one is good for small experiments and to test principles of the environment but it does not fit any design of reasonable size.

Test circuits were selected from ISCAS'85, ISCAS'89 and HLSynt'91 benchmark sets to evaluate the speedup when replacing fault simulation with emulation on FPGA. Results of some benchmarks are presented in the paper to illustrate gains and losses of our approach. Synthesis results are presented in Table 1 to illustrate how the fault point modeling makes the circuits more complex. Columns "initial" and "faults" illustrate the increase both in size in equivalent gates and delay before and after fault point insertion (CUT only). The "FPGA" columns illustrate the size in FPGA logic blocks and clock frequency of the whole test bench.

Performance results are presented in Table 2 where the columns #I, #O, #ff, and #F represent the number of inputs (clock and reset are excluded), outputs, flip-flops, and



```
(Combinational circuits)
reset_LFSRs;
for every test_vector
  emulate_CUT;
store_signature;
for every fault {
  reset_LFSRs;
  for every test_vector
    emulate_CUT;
  compare_signature;
}
```

```
(Sequential circuits)
reset_LFSRs;
for every test_sequence {
  reset_CUT;
  for every test_vector
    emulate_CUT;
}
store_signature;
for every fault {
  reset_LFSRs;
  for every test_sequence {
    reset_CUT;
    for every test_vector
      emulate_CUT;
  }
  compare signature;
```

Fig. 5. Algorithms implemented by the state machine

Table 1. Synthesis results

circuit	# of gates		Delay [ns]		FPGA	
	initial	faults	Initial	faults	CLBs	MHz
c17	7	69	1.9	17.8	36	100
c2670	533	3315	25.4	67.6	951	25
c3540	727	4290	28.6	93.6	974	15
c5315	1285	8871	28.6	200	1784	15
c6288	2632	15.6k	97.6	509	3223	5
s5378	4933	12.4k	21.8	268	2583	10
s15850	17.1k	29.4k	66.8	633	6125	5
GCD (16-bit)	926	3331	16.6	74.0	588	25
GCD (32-bit)	2061	8513	20.0	203	1738	10
prefetch (16-bit)	796	2264	14.5	52.4	478	40
prefetch (32-bit)	1698	4608	20.0	72.7	941	25
diff-eq (16-bit)	4562	22.4k	25.7	566	4672	5
TLC	290	1089	9.5	39.0	215	50

fault points, respectively. For sequential circuits, the number of test vectors is given in two columns – the number of test sequences ("# of seq.") and the length of a test sequence ("seq.len."). The column SW gives the fault simulation time basing on the parallel algorithm running on a 366 MHz SUN UltraSPARC 20 server and "HW emul" emulation time for the same set of test vectors. Additionally, synthesis times have been added for comparison ("HW synt").

As it is shown in Table 2, the hardware emulation was in average 17.8 (ranging from 6.7 to 53.4) times faster than the software fault simulation. It should be noted

Table 2. Fault simulation: SW versus HW

Circuit	#I	#O	#ff	#F	# of vectors		SW	HW	
					# of seq.	seq. len.		synt	emul
c17	5	2	-	11	10M	1	18.7”	25”	3.1”
c2670	157	64	-	824	20k	1	34.8”	5.2’	1.33”
c3540	50	22	-	1036	10k	1	20.9”	7.8’	1.39”
c5315	178	123	-	2076	1000	1	5.63”	13’	0.28”
c6288	32	32	-	3559	1000	1	18.3”	35’	1.43”
s5378	35	49	179	2517	80	100	26.8”	22’	4.0”
s15850	77	150	534	6076	200	200	15.6’	55’	97”
GCD (16-bit)	33	16	49	723	80	50	5.28”	3.9’	0.23”
GCD (32-bit)	65	32	97	1675	10	400	22.6”	12’	1.34”
prefetch (16-bit)	34	48	64	420	40	100	1.34”	2.4’	0.10”
prefetch (32-bit)	66	96	128	923	40	400	9.46”	5.1’	1.19”
diff-eq (16-bit)	80	48	115	4789	20	200	87.9”	45’	7.7”
TLC	3	6	17	196	40	100	2.69”	1.2’	0.05”

that when considering also synthesis times, it might not be useful to replace simulation with emulation, especially for smaller designs. Nevertheless, taking into account that sequential circuits, as opposed to combinational ones, have much longer test sequences, the use of emulation will pay off. Our preliminary experiments show that for this type of circuits an average speed-up of about 70 times is expected. It is important to keep in mind that simulation-based test pattern generation for a sequential design of roughly 10 kgates takes tens of hours. Future research will mainly focus on test generation for sequential circuits using genetic algorithms.

6 Conclusions

The experiments showed that for circuits that require large numbers of test vectors, e.g., sequential circuits, it is beneficial to replace simulation with emulation. Although even for combinational circuits the simulation speedup is significant, there exist rather large penalty caused by synthesis time. Based on that, it can be concluded that the most useful application would be to explore test generation and analysis architectures based on easily reprogrammed structures, e.g., LFSRs. This makes fault emulation very useful to select the best generator/analyzer structures for BIST.

Another useful application of fault emulation would be genetic algorithms of test pattern generation where also large numbers of test vectors are analyzed. Future work includes development of more advanced on chip test vector generators and analyzers. Analysis of different fault point insertion structures is another direction of future work. Also, an automated synthesis flow development and integration into Turbo Tester environment, together with hardware-software interface development, has been initiated. For academic purposes where expensive state-of-the-art hardware can not be used, partitioning methods should be developed to enable emulation of fault list partitions.

Acknowledgments. This work was supported in part by Estonian Science Foundation grants No 5601 and 5637.

References

1. S. Seshu, "On an improved diagnosis program." IEEE Trans. on Electron. Comput., vol. EC-14, pp.76-79, 1965.
2. P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincetelli, P. Scaglia, "Fast Discrete Function Evaluation Using Decision Diagrams." Proc. of ICCAD'95, pp.402-407, Nov. 1995.
3. "Axis Systems Uses World's Largest FPGAs from Xilinx to Deliver Most Efficient Verification System in the Industry." Xilinx Press Release #0273 - http://www.xilinx.com/
4. R. Sedaghat-Maman, E. Barke, "A New Approach to Fault Emulation." Proc. of Rapid System Prototyping, pp.173-179, June 1997.
5. P. Ellervee, J. Raik, V. Tihhomirov, "Fault Emulation on FPGA: A Feasibility Study." Proc. of Norchip'03, Riga, Latvia, Nov. 11-12, 2003.
6. M. Abramovici, P. Menon, "Fault Simulation on Reconfigurable Hardware." Proc. of FPGAs for Custom Computing Machines, pp.182-190, April 1997.
7. S.-K. Lu, J.-L. Chen, C.-W. Wu, W.-F. Chang, S.-Y. Huang, "Combinational Circuit Fault Diagnosis Using Logic Emulation." Proc. of ISCAS'03, Vol.5, pp.549-552, May 2003.
8. M. Alderighi, S. D'Angelo, M. Mancini, G.R. Sechi, "A Fault Injection Tool for SRAM-based FPGAs." Proc. of 9th IEEE International On-Line Testing Symposium (IOLTS'03), pp.129-133, July 2003.
9. R. Wieler, Z. Zhang, R. D. McLeod, "Simulating Static and Dynamic Faults in BIST Structures with a FPGA Based Emulator." Proc. of FPL’94, Springer-Verlag, pp.240-250, Sept. 1994.
10. K.-T. Cheng, S.-Y. Huang, W.-J. Dai, "Fault emulation: a new approach to fault grading." Proc. of ICCAD’95, pp.681-686, Nov. 1995.
11. L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, O. Lepape, "Serial fault emulation." Proc. DAC’96, pp.801-806, Las Vegas, USA, June 1996.
12. S.-A. Hwang, J.-H. Hong, C.-W. Wu, "Sequential Circuit Fault Simulation Using Logic Emulation." IEEE Trans. on CAD of Int. Circuits and Systems, Vol.17, No.8, pp.724-736, Aug. 1998.
13. R. Wieler, Z. Zhang, R. D. McLeod, "Emulating Static Faults Using a Xilinx Based Emulator." Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, pp.110-115, April 1995.
14. G. Jervan, A. Markus, P. Paomets, J. Raik, R. Ubar, "Turbo Tester: A CAD System for Teaching Digital Test." Microelectronics Education, Kluwer Academic Publishers, pp.287-290, 1998.
15. "Turbo Tester" home page – URL: http://www.pld.ttu.ee/tt
16. D.K. Pradhan, C. Liu, K. Chakraborty, "EBIST: A Novel Test Generator with Built in Fault Detection Capability." Proc. of DATE'03, pp. 224-229, Munich, Germany, March 2003.
17. "Xilinx, Inc." home page – URL: http://www.xilinx.com/
18. "XESS Corp." home page – URL: http://www.xess.com/
19. "Celoxica, Ltd." home page – URL: http://www.celoxica.com/

Automating Optimized Table-with-Polynomial Function Evaluation for FPGAs

Dong-U Lee, Oskar Mencer, David J. Pearce, and Wayne Luk

Department of Computing, Imperial College, London, UK
{dong.lee, o.mencer, d.pearce, w.luk}@ic.ac.uk

Abstract. Function evaluation is at the core of many compute-intensive applications which perform well on reconfigurable platforms. Yet, in order to implement function evaluation efficiently, the FPGA programmer has to choose between a multitude of function evaluation methods such as table lookup, polynomial approximation, or table lookup combined with polynomial approximation. In this paper, we present a methodology and a partially automated implementation to select the best function evaluation hardware for a given function, accuracy requirement, technology mapping and optimization metrics, such as area, throughput and latency. The automation of function evaluation unit design is combined with ASC, A Stream Compiler, for FPGAs. On the algorithmic side, MATLAB designs approximation algorithms with polynomial coefficients and minimizes bitwidths. On the hardware implementation side, ASC provides partially automated design space exploration. We illustrate our approach for $\sin(x)$, $\log(1+x)$ and 2^x with a selection of graphs that characterize the design space with various dimensions, including accuracy, precision and function evaluation method. We also demonstrate design space exploration by implementing more than 400 distinct designs.

1 Introduction

The evaluation of functions can often be the performance bottleneck of many compute-bound applications. Examples of these functions include elementary functions such as $\log(x)$ or $\sqrt{x}$, and compound functions such as $(1-\sin^2(x))^{1/2}$ or $\tan^2(x)+1$. Hardware implementation of elementary functions is a widely studied field with many research papers (e.g. [1][10][11][12]) and books (e.g. [2][8]) devoted to the topic. Even though many methods are available for evaluating functions, it is difficult for designers to know which method to select for a given implementation.

Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications. Such customization can take place at run time by reconfiguring the FPGA, so that different functions, function evaluation methods, or precision can be introduced according to run-time conditions. Consequently, the automation of function evaluation design is one of the key bottlenecks in the further application of function evaluation in reconfigurable computing. The main contributions of this paper are:

- A methodology for the automation of function evaluation unit design, covering table lookup, table with polynomial, and polynomial-only methods.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 364–373, 2004.

- An implementation of a partially automated system for design space exploration of function evaluation in hardware, including:
  - Algorithmic design space exploration with MATLAB.
  - Hardware design space exploration with ASC.
- Method selection results for $\sin(x)$, $\log(1 + x)$ and 2^x.

The rest of this paper is organized as follows. Section 2 covers overview and background material. Section 3 presents the algorithmic design space exploration with MATLAB. Section 4 describes the automation of the ASC design space exploration process. Section 5 shows how ASC designs can be verified. Section 6 discusses results, and Section 7 offers conclusion and future work.

2 Overview and Background

We can use polynomials and/or lookup tables for approximating a function $f(x)$ over a fixed range $[a, b]$. On one extreme, the entire function approximation can be implemented as a table lookup. On the other extreme, the function approximation can be implemented as a polynomial approximation with function-specific coefficients. In our work, we use Horner's rule to reduce the number of multiplications.

Between these two extremes, we use a table followed by a polynomial. This table with polynomial method partitions the total approximation into several segments. In this work, we employ uniformly sized segments, which have been widely studied in literature [1][3][5]. Uniform segmentation performs well for functions that are relatively linear, such as the functions we consider in this paper. However, for highly non-linear functions, non-uniform segmentation methods such as the hierarchical segmentation method [4] have been found to be more appropriate.

In [7] the results show that for a given accuracy requirement it is possible to plot the area, latency, and throughput tradeoff and thus identify the optimal function evaluation method. The optimality depends on further requirements such as available area, required latency and throughput. Looking at Figure 1, if one desires the metric to be low (e.g. area or latency), one should use method 1 for bitwidths lower than x1, method 2 for bitwidths between x1 and x2, and method 3 for bitwidths higher than x2. We shall illustrate this approach using Figures 13 to 15, where several methods are combined to provide the optimal implementations in area, latency or throughput for different bit-widths for the function $\sin(x)$.

The contribution of this paper is the design and implementation of a methodology to automate this process. Here, MATLAB automates the mathematical side of function approximation (e.g. bitwidth and coefficient selection), while *A Stream Compiler (ASC)* [6] automates the design space exploration of area, latency and throughput. Figure 2 shows the proposed methodology.

3 Algorithmic Design Space Exploration with MATLAB

Given a target accuracy, or number of output bits so that the required accuracy is one unit in the last place (1 ulp), it is straightforward to automate the design of a sufficiently

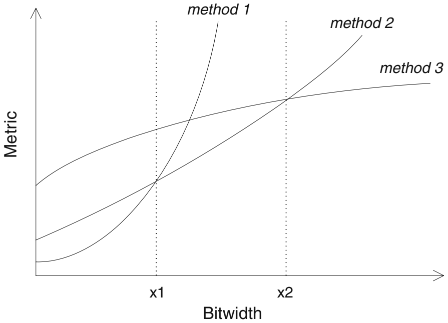


Fig. 1. Certain approximation methods are better than others for a given metric at different precisions.

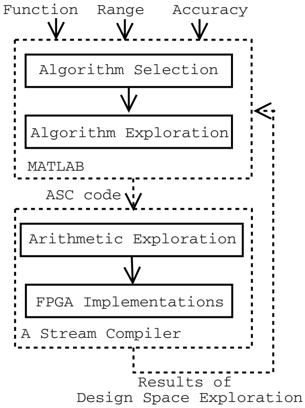


Fig. 2. Block diagram of methodology for automation.

accurate table, and with help from MATLAB, also to find the optimal coefficient for a polynomial-only implementation. The interesting designs are between the table-only and polynomial-only designs – those involving both a table and a polynomial. Three MATLAB programs have been developed: TABLE (table lookup), TABLE+POLY (table with polynomial) and POLY (polynomial only). The programs take a set of parameters (e.g. function, input range, operand bitwidth, required accuracy, bitwidths of the operations and the coefficients and the polynomial degree) and generate function evaluation units in ASC code.

TABLE produces a single table, holding results for all possible inputs; each input is used to index the table. If the input is n bits and the precision of the results is m bits, the size of the table would be $2^n \times m$. It can be seen that the disadvantage of this approach is that the table size is exponential to the input size.

TABLE+POLY implements the table with polynomial method. The input interval $[a, b]$ is split into $N = 2^I$ equally sized segments. The I leftmost bits of the argument x serve as the index into the table, which holds the coefficients for that particular interval. We use degree two polynomials for approximating the segments, but other degrees are possible. The program starts with $I = 0$ (i.e. one segment over the whole input range) and finds the minimax polynomial coefficients which minimize the maximum absolute error. I is incremented until the maximum error over all segments is lower than the requested error. The operations are performed in fixed point and in finite precision with the user supplied parameters, which are emulated by MATLAB.

POLY generates an implementation which approximates the function over the whole input range with a single polynomial. It starts with a degree one polynomial and finds the minimax polynomial coefficients. The polynomial degree is incremented until the desired accuracy is met. Again, fixed point and finite precision are emulated.

4 Hardware Design Space Exploration with ASC

ASC [6] enables a software-like programming of FPGAs. ASC is built on top of the module generation environment PAM-Blox II, which in turn builds upon the PamDC [9] gate library. While [6] shows the details of the design space exploration process with ASC, we now utilise ASC (version 0.5) to automate this process. The idea is to retain user-control over all features available on the gate level, whilst automating many of the tedious tasks involved in exploring the design space. Therefore ASC allows the user to specify the dimensions of design space exploration, e.g. bitwidths of certain variables, optimization metrics such as area, latency, or throughput, and in fact anything else that is accessible in ASC code, which includes algorithm level, arithmetic unit level and gate level constructs. For example, suppose we wish to explore how the bitwidth of a particular ASC variable affects area and throughput. To do this we first parameterize the bitwidth definition of this variable in the ASC code. Then we specify the detail of the exploration in the following manner:

$$\texttt{RUN0} \;=\; -\texttt{XBITWIDTH} = \{\texttt{8}, \texttt{16}, \texttt{24}, \texttt{32}\} \tag{1}$$

which states that we wish to investigate bitwidths of 8, 16, 24 and 32. At this point, typing 'make run0' begins an automatic exploration of the design space, generating a vast array of data (e.g. Number of 4-input LUTs, Total Equivalent Gate Count, Throughput and Latency) for each different bitwidth. ASC also automatically generates graphs for key pieces of this data, in an effort to further reduce the time required to evaluate it.

The design space explorer, or "user", in our case is of course the MATLAB program that mathematically designs the arithmetic units on the algorithmic level and provides ASC with a set of ASC programs, each of which results in a large number of implementations. Each ASC implementation in return results in a number of design space exploration graphs and data files. The remaining manual step, which is difficult to automate, involves inspecting the graphs and extracting useful information about the variation of the metrics. It would be interesting to see how such information from the hardware design space exploration can be used to steer the algorithmic design space exploration.

One dimension of the design space is technology mapping on the FPGA side. Should we use block RAMs, LUT memory or LUT logic implementations of the mathematical lookup tables generated by MATLAB? Table 1 shows ASC results which substantiate the view that logic minimization of tables containing smooth functions is usually preferable over using block RAMs or LUT memory to implement the table. Therefore, in this work we limit the exploration to combinational logic implementations of tables.

5 Verification with ASC

One major problem of automated hardware design is the verification of the results, to make sure that the output circuit is actually correct. ASC offers two mechanisms for this activity based on a software version of the implementation.

- **Accuracy Graphs**: graphs showing the accuracy of the gate-level simulation result (SIM) compared to a software version using double precision floating point (SW), automatically generated by MATLAB, plotting:

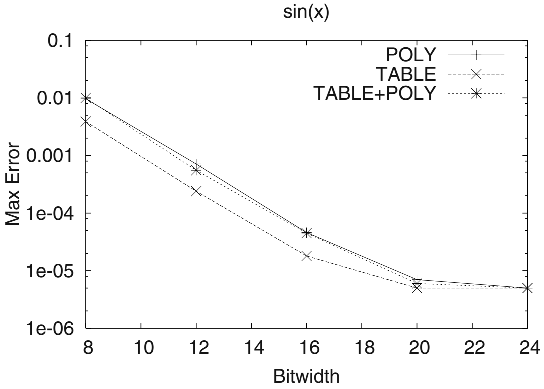


Fig. 3. Accuracy graph: maximum error versus bitwidth for $\sin(x)$ with the three methods.

Table 1. Various place and route results of 12-bit approximations to $\sin(x)$. The logic minimized LUT implementation of the tables minimizes latency and area, while keeping comparable throughput to the other methods, e.g. block RAM (BRAM) based implementation.

ASC optimization	memory type	4-input LUTs	clock speed [MHz]	latency [ns]	throughput [Mbps]
latency	block RAM	**919 + 1BRAM**	17.89	**111.81**	250.41
	LUT memory	1086	15.74	63.51	220.43
	LUT logic	**813**	16.63	**60.11**	232.93
throughput	block RAM	**919 + 1BRAM**	39.49	177.28	**552.79**
	LUT memory	1086	36.29	192.88	508.09
	LUT logic	**967**	39.26	178.29	**549.67**

`max.error` $= max(|SW - SIM|)$, or
`max.error` $= max(|SW - FPGA|)$
when comparing to an actual FPGA output ($FPGA$).

Figure 3 shows an example graph. Here the precisions of the coefficients and the operations are increased according to the bitwidth (e.g. when bitwidth=16, all coefficients and operations are set to 16 bits), and the output bitwidth is fixed at 24 bits.

– **Regression Testing**: same as the accuracy graph, but instead of plotting a graph, ASC compares the result to a maximally tolerated error and reports only ‘pass’ or ‘fail’ at the end. This feature allows us to automate the generation and execution of a large number of tests.

6 Results

We show results for three elementary functions: $\sin(x)$, $\log(x+1)$ and 2^x. Five bit sizes 8, 12, 16, 20 and 24 bits are considered for the bitwidth. In this paper, we implement designs with n-bit inputs and n-bit outputs. However, the position of the decimal (or binary) point in the input and output formats can be different in order to maximize the precision that can be described. The results of all 400 implementations are post place and route, and are implemented on a Xilinx Virtex-II XC2V6000-6 device.

In algorithmic space explored by MATLAB, there are three methods, three functions and five bitwidths, resulting in 45 designs. These designs are generated by the user with hand-optimized coefficient and operation bitwidths. ASC takes the 45 algorithmic designs and expands them into over 400 implementations in the hardware space. With the aid of the automatic design exploration features of ASC (Section 4), we are able to generate all the implementation results in one go with a single 'make' file. It takes around twelve hours on a dual Athlon XP 2.13GHz PC with 2GB RAM.

The following graphs are a subset of the full design space exploration which we show for demonstration purposes. Figures 4 to 15 show a set of FPGA implementations resulting from a 2D cut of the multidimensional design space.

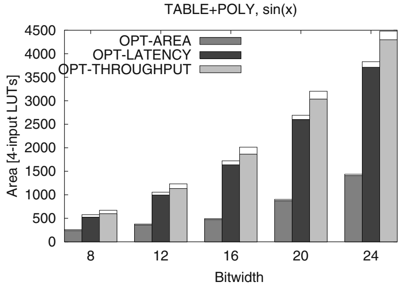
In Figures 4 to 6, we fix the function and approximation method to $\sin(x)$ and TABLE+POLY, and obtain area, latency and throughput results for various bitwidths and optimization methods. Degree two polynomials are used for all TABLE+POLY experiments in our work.

Figure 4 shows how the area (in terms of the number of 4-input LUTs) varies with bitwidth. The lower part shows LUTs used for logic while the small top part of the bars shows LUTs used for routing. We observe that designs optimized for area are significantly smaller than other designs. In addition, as one would expect, the area increases with the bit width. Designs optimized for throughput have the largest area; this is due to the registers used for pipelining. Figure 5 shows that designs optimized for latency have significantly less delay, and the increase in delay with the bitwidth is lower than others. Designs optimized for area have the longest delay, which is due to hardware being shared in a time-multiplexed manner. Figure 6 shows that designs optimized for throughput perform significantly better than others. Designs optimized for area perform worst, which is again due to the hardware sharing. An interesting observation is the fact that throughput is relatively constant with bitwidth. This is due to increased routing delays as designs get larger with increased precision requirements.

Figures 7 to 9 show various metric-against-metric scatter plots of 12-bit approximations to $\sin(x)$ with different methods and optimizations. For TABLE, only results with area optimization are shown since the results for other optimizations applied are identical. With the aid of such plots, one can decide rapidly what methods to use for meeting specific requirements in area, latency or throughput.

In Figures 10 to 12, we fix the approximation method to TABLE+POLY, and obtain area, latency and throughput results for all three functions at various bitwidths. Optimum optimization methods are used for all three experiments (e.g. area is optimized to get the area results).

From Figure 10, we observe that $\sin(x)$ requires the most and 2^x requires the least area. The difference gets more apparent as the bitwidth increases. This is because 2^x



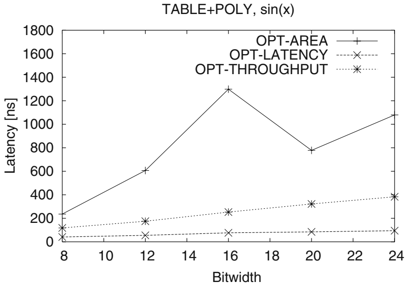
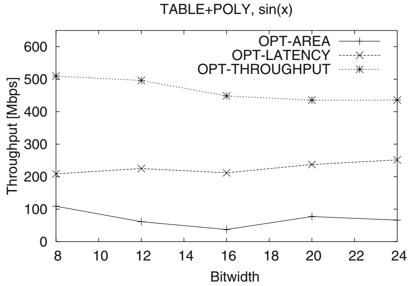


Fig. 4. Area versus bitwidth for $\sin(x)$ with TABLE+POLY. OPT indicates for what metric the design is optimized for. Lower part: LUTs for logic; small top part: LUTs for routing.

Fig. 5. Latency versus bitwidth for $\sin(x)$ with TABLE+POLY. Shows the impact of latency optimization.



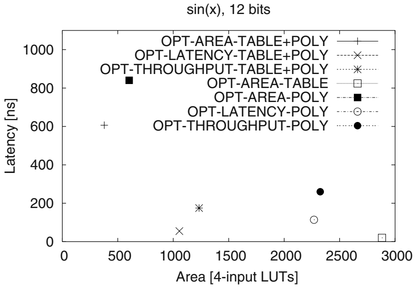
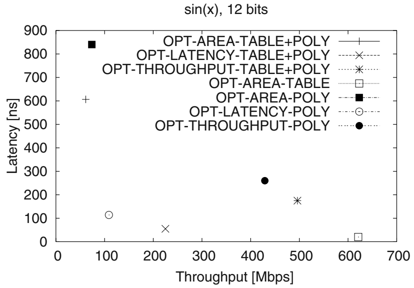


Fig. 6. Throughput versus bitwidth for $\sin(x)$ with TABLE+POLY. Shows the impact of throughput optimization.

Fig. 7. Latency versus area for 12-bit approximations to $\sin(x)$. The Pareto optimal points in the latency-area space are shown.



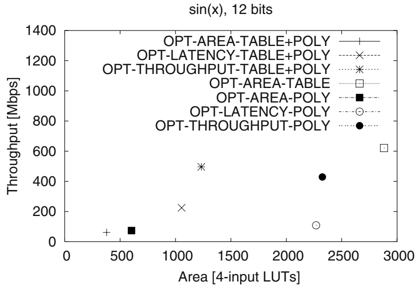


Fig. 8. Latency versus throughput for 12-bit approximations to $\sin(x)$. The Pareto optimal points in the latency-throughput space are shown.

Fig. 9. Area versus throughput for 12-bit approximations to $\sin(x)$. The Pareto optimal points in the throughput-area space are shown.

is the most linear of the three functions, hence requires fewer number of segments for the approximations. This leads to a reduction in the number of entries in the coefficient table and hence less area on the device.

Figure 11 shows the variations of the latency with the bitwidth. We observe that all three functions have similar behavior. In Figure 12, we observe that again the three functions have similar behavior, with 2^x performing slightly better than others for bitwidths higher than 16. We suspect that this is because of the lower area requirement of 2^x, which leads to less routing delay.

Figures 13 to 15 show the main emphasis and contribution of this paper, illustrating which approximation method to use for the best area, latency or throughput performance. We fix the function to $\sin(x)$ and obtain results for all three methods at various bit widths. Again, the optimum optimization is used for a given experiment. For experiments involving TABLE, we have managed to obtain results up to 12 bits only, due to memory limitations of our PCs.

From Figure 13, we observe that TABLE has the least area at 8 bits, but the area increases rapidly making it less desirable at higher bitwidths. The reason for this is the exponential increase in table to the input size for full lookup tables. The TABLE+POLY approach yields the least area for precisions higher than eight bits. This is due to the efficiency of using multiple segments with minimax coefficients for each. We have observed that for POLY, roughly one more polynomial term (i.e. one more multiply-and-add module) is needed every four bits. Hence, we see a linear behavior with the POLY curve.

Figure 14 shows that TABLE has significantly smaller latency than others. We expect that this will be the case for bitwidths higher than 12 bits as well. POLY has the worst delay, which is due to computations involving high-degree polynomials, and the terms of the polynomials increase with the bitwidth. The latency for TABLE+POLY is relatively low across all bitwidths, which is due to the fact that the number of memory accesses and polynomial degree are fixed.

In Figure 15, we observe how the throughput varies with bitwidth. For low bitwidths, TABLE designs result in the best throughput, which is due to the short delay for a single memory access. However, the performance quickly degrades and we predict that at bit widths higher than 12 bits, it will perform worse than the other two methods due to rapid increase in routing congestion. The performance of TABLE+POLY is better than POLY before 15 bits and gets worse after. This is due to the increase in the size of the table with precision, which leads to longer delays for memory accesses.

7 Conclusions

We present a methodology for the automation of function evaluation unit design, covering table lookup, table with polynomial, and polynomial-only methods. An implementation of a partially automated system for design space exploration of function evaluation in hardware has been demonstrated, including algorithmic design space exploration with MATLAB and hardware design space exploration with ASC.

We conclude that the automation of function evaluation unit design is within reach, even though there are many remaining issues for further study. Current and future work includes optimizing polynomial evaluation, exploring the interaction between range reduction and function evaluation, including more approximation methods, and developing a complete and seamless automation of the entire process.

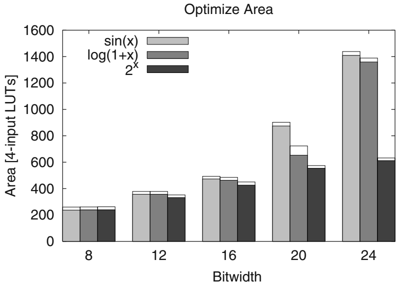


Fig. 10. Area versus bitwidth for the three functions with TABLE+POLY. Lower part: LUTs for logic; small top part: LUTs for routing.

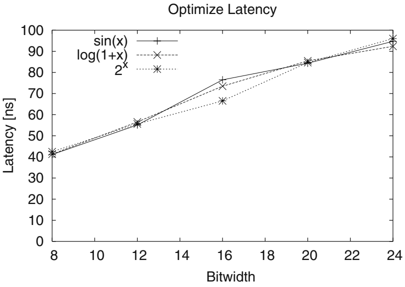


Fig. 11. Latency versus bitwidth for the three functions with TABLE+POLY.

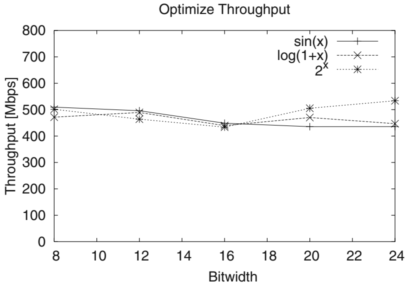


Fig. 12. Throughput versus bitwidth for the three functions with TABLE+POLY. Throughput is similar across functions, as expected.

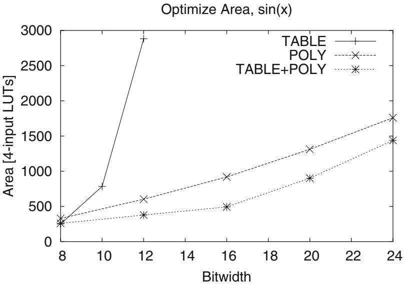


Fig. 13. Area versus bitwidth for $\sin(x)$ with the three methods. Note that the TABLE method gets too large already for 14 bits.

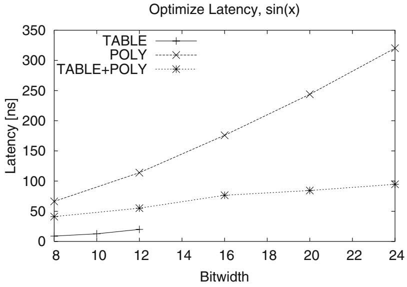


Fig. 14. Latency versus bitwidth for $\sin(x)$ with the three methods.

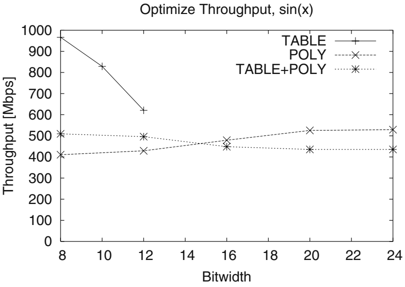


Fig. 15. Throughput versus bitwidth for $\sin(x)$ with the three methods.

Acknowledgements. The authors thank George Constantinides and Ray Cheung for their assistance. The support of Xilinx Inc. and the U.K. Engineering and Physical

Sciences Research Council (Grant number GR/N 66599, GR/R 55931 and GR/R 31409) is gratefully acknowledged.

References

1. J. Cao, B.W.Y. We and J. Cheng, " High-performance architectures for elementary function generation", *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.
2. M.J. Flynn and S.F. Oberman, *Advanced Computer Arithmetic Design*, John Wiley & Sons, New York, 2001.
3. V.K. Jain, S.A. Wadecar and L. Lin, "A universal nonlinear component and its application to WSI", *IEEE Trans. Components, Hybrids and Manufacturing Tech.*, vol. 16, no. 7, pp. 656–664, 1993.
4. D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation", *Proc. IEEE Int. Conf. on Field-Prog. Tech.*, pp. 92–99, 2003.
5. D.M. Lewis, "Interleaved memory function interpolators with application to an accurate LNS arithmetic unit", *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974–982, 1994.
6. O. Mencer, D.J. Pearce, L.W. Howes and W. Luk, "Design space exploration with a stream compiler", *Proc. IEEE Int. Conf. on Field-Prog. Tech.*, pp. 270–277, 2003.
7. O. Mencer and W. Luk, "Parameterized high throughput function evaluation for FPGAs", *J. of VLSI Sig. Proc. Syst.*, vol. 36, no. 1, pp. 17–25, 2004.
8. J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Verlag AG, 1997.
9. B. Patrice, R. Didier and V. Jean, "Programmable active memories: a performance assessment", *Proc. ACM Int. Symp. on Field-Prog. Gate Arrays*, 1992.
10. M.J. Schulte and J.E. Stine, "Approximating elementary functions with symmetric bipartite tables", *IEEE Trans. on Comput.*, vol. 48, no. 9, pp. 842–847, 1999.
11. P.T.P. Tang, "Table lookup algorithms for elementary functions and their error analysis", *Proc. IEEE Symp. on Comput. Arith.*, pp. 232–236, 1991.
12. W.F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers", *IEEE Trans. on Comput.*, vol. 43, pp. 278–294, 1994.

Multiple Restricted Multiplication

Nalin Sidahao, George A. Constantinides, and Peter Y.K. Cheung

Department of Electrical & Electronic Engineering, Imperial College, London, UK,
{nalin.sidahao, g.constantinides, p.cheung}@imperial.ac.uk

Abstract. This paper focuses on a class of problem relating to the multiplication of a single number by several coefficients that, while not constant, are drawn from a finite set of constants that change with time. To minimize the number of operations, we present the formulation as a form of common sub-expression elimination. The proposed scheme avoids the implementation of full multiplication. In addition, an efficient implemenation is presented targeting the Xilinx Virtex / Virtex-II family of FPGAs. We also introduce a novel use of Integer Linear Programming for finding solutions to the minimum-cost of such a multiplication problem. Our formulation results in area savings even for modest problem sizes.

1 Introduction

For Digital Signal Processing (DSP) or arithmetic intensive applications, multiplication is considered as an expensive operation. This is because, typically, the main part of the area consumed in their implementation comes from multipliers. For a constant coefficient multiplication, instead of using a full multiplier, a general method to efficiently reduce the hardware usage is to use a series of binary shifts and adders. A shift operation may have almost negligible cost since it is hard-wired. Therefore, the total hardware cost is approximately corresponding to the area of adders required.

Reducing the number of adders in constant multiplication is an optimization problem. The key point of most existing research is the minimization of this quantity, which is an NP-hard problem [1].

By contrast, our approach introduces a form of the common sub-expression (CSE) elimination problem, which we refer to as multiple restricted multiplication (MRM). This refers to a situation where a single variable is multiplied by several coefficients which, while not constant, are drawn from a relatively small set of values. Such a situation arises commonly in synthesis due to resource sharing, for example in a folded implementation of a FIR filter [2] or a polynomial evaluation using Estrin's method [3,4]. Recent FPGA architectures have included dedicated multiplier blocks. By exploiting our technique, these blocks are freed to be used for true general multiplications.

Existing approaches to CSE are unable to take advantage of such a situation, resulting in the use of expensive general multipliers, as shown in Fig. 1. Fig. 1(a) shows a Data Flow Graph (DFG) with input x tied together, input sets of

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 374–383, 2004.

constant multiplicands labelled as $\{c_{11}, c_{12}, \ldots, c_{1T}\}$ and $\{c_{21}, c_{22}, \ldots, c_{2T}\}$. The first subscript here refers to the spatial index and the second to the time index, i.e. c_{it} is the value of multiplicand i at time t. A standard technique using ROMs and multipliers is depicted in Fig. 1(b) and our approach performing equivalently is shown as a "black block" in Fig. 1(c).

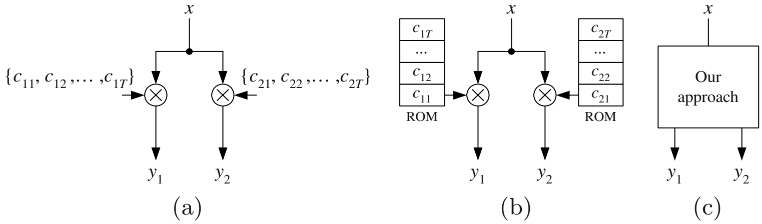


Fig. 1. (a) A DFG with only multiplier nodes, one input x, and other two inputs of multipliers. (b) The standard implementation with ROMs and Multipliers. (c) A black box interpretation of our approach.

In this paper, it is shown that the MRM problem can be addressed through extending the basic unit of operation from an addition, used in multiple constant multiplication (MCM) [7,8], to a novel adder-multiplexer combination. It is further demonstrated that for Xilinx-based implementations, the Xilinx Virtex / Virtex-II slice architecture [9] can be used to implement such a basic computational unit with no area overhead compared to the equivalent adder used in MCM.

A similar technique was previously presented by R.H. Turner, *et al.* [5] for implementing multipliers with a limited range of coefficients, which we extend by making use of the dedicated AND gate presented in the slice. The key is to exploit the set of primitive components: the supplementary logic gates, next to each LUT, and the dedicated carry-chain logic. Full utilization of these allows the implementation of an adder and/or a substractor along with a multiplexer in a novel configuration. This can be applied to constant multiplication using sub-expression sharing to achieve efficient FPGA implementation. A recent work by S.S. Demirsoy, *et al.* [6] has begun to address this problem using the type of computational node demostrated in [5].

Since MCM is NP-hard [1] and is a special case of MRM, it follows that MRM is NP-hard. Thus in order to find the area-optimal implementation of a given MRM block, a novel formulation of the optimization problem as a class of Integer Linear Program (ILP) is proposed. This approach allows us to leverage the recent advances in the field of ILP solution.

This paper therefore has the following novel contributions: 1. the introduction of the MRM problem, and its solution using a novel extension of adder-multiplexer cells. 2. the formulation of the minimum-area MRM as ILP formulation. 3. an efficient use of the Xilinx Virtex / Virtex-II slice structure to implement MRM.

This paper has the following structure. Section 2 describes the background of MCM. Section 3 describes the proposed architectural solution to the MRM problem, and Section 4 demonstrates that this solution can be efficiently implemented in the Xilinx Virtex family of FPGAs. Section 5 formulates the optimization problem in ILP, Section 6 collects and discusses results from MRM, and Section 7 concludes the paper.

2 Background

2.1 Multiple Constant Multiplication

MCM, a special case of the MRM problem address in this paper, has been used in many research fields, especially in DSP applications. A common use of MCM is within the design of fully unfolded digital filters [2]. The main idea of this technique involves removing the redundancy inherent in the re-computation of common sub-expressions. Applying this approach provides a significant reduction of area necessary to implement multiple constant multiplications. Even within a single multiplication, common sub-expressions exist. As an example, consider integer multiplication with constant coefficient 10100101. Let us denote left-shift as <<. Instead of performing $(x << 7) + (x << 5) + (x << 2) + x$ where x is an input variable, we can perform $(y << 5) + y$ where $y = (x << 2) + x$. Hardware is then saved due to the elimination of the 101 $(x << 2) + x$ sub-expression. Sharing such sub-expressions across several coefficients results in significant savings.

2.2 Representing Multiplicands with Data Flow Graphs

DFGs are the basis of a computational model used extensively in DSP. A DFG is a directed graph, with nodes in one to one correspondence with operations and edges in one to one correspondence with data flow. We shall consider edge-weighted DFGs, where the edge weight corresponds to a shift operation. For example, the CSE case considered above may be represented as a DFG in Fig. 2.

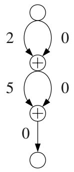


Fig. 2. DFG representation of coefficient 10100101

The topmost and bottommost nodes are the initial node (input node) and terminal node (output node), respectively. In general, a DFG may have more than one terminal node, corresponding to the different constant coefficients.

Each intermediate node (an adder in standard MCM), has two input edges and at least one output edge. Each DFG represents a way of sequencing addition operations such that the required coefficients are produced. We may then ask the optimization question: What is the minimum number of nodes required to compute a given set of constant coefficients? This is the problem addressed by existing work on MCM [7,8], which we extend here to the case of MRM.

3 MRM with Adder/Multiplexer Nodes

We now extend the DFG model from using adder nodes to using adder/multiplexer nodes. Each node's internal structure now consists of not only an adder, but also a 4-1 multiplexer, as shown in Fig. 3.

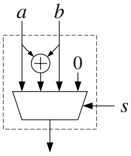


Fig. 3. An adder/multiplexer node

Using such circuit provides operations that perform adding $(a + b)$, passing one of two input values (a, b) through the multiplexer, and generating a zero to output value. Each operation is selected by a 2-bit selector s. Applying this technique provides the flexibility to move from MCM to MRM. Each node may perform a different operation at each different time frame. For instance, when applied to Fig. 2, the multiplicand can be 10100101 (lower node and upper node adding), 10000100 (lower node adding, upper node passing through the left-hand input) or some other coefficients; depending on what the selectors are. The DFG structure, and the shift quantities, remain constant over all time steps. This means that the structure can be directly mapped into a circuit, and the shifts remain cost-free.

4 An Efficient Xilinx Virtex / Virtex-II Implementation

This section describes an efficient adder/multiplexer implementation in a Xilinx Virtex / Virtex-II device. The Virtex series utilizes a Configurable Logic Block (CLB) architecture. Each CLB consists of two slices, each containing two logic cells (LCs). A diagram shown in Fig. 4 is a simplified Virtex-II architecture; more information can be found in [9]. An LC has several logic components including one 4-input look-up table (LUT), some MUXes and some dedicated logic, including one MUXCY, one XORCY, and one MULT_AND gate.

The adder/multiplexer node is designed using a similar idea to the way a ripple-carry adder is implemented using the Virtex carry chain. A full adder/mul-

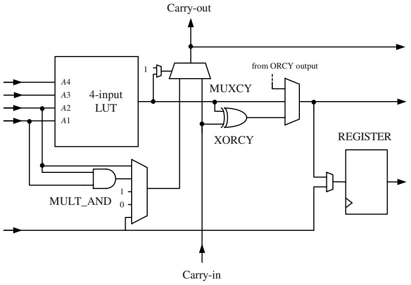
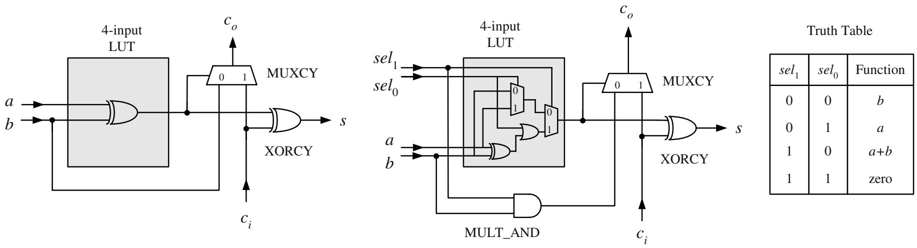


Fig. 4. Simplified Virtex-II structure (top half of a slice)

tiplexer can be efficiently implemented using the same logic hardware resources as a common adder. The key is leveraging MULT_AND gate, an additional 2-input dedicated AND gate (typically used to implement an efficient 1-bit multiplier), and the carry-chain logic.

Fig. 5(a) illustrates a simple 1-bit full-adder which performs addition two inputs a, b and carry-in c_i, and results the summation s and carry-out c_o. We can extend this structure to perform a bit-slice of the entire function shown in Fig. 3. The four operations are controlled by a 2-bit selector (sel_1, sel_0) absorbed into the 4-input LUT. A 1-bit adder/multiplexer can fit in one LC as depicted in Fig. 5(b).



sel_1	sel_0	Function
0	0	b
0	1	a
1	0	a+b
1	1	zero

(a) full adder (b) adder/multiplexer (c)

Fig. 5. 1-bit adder and adder/multiplexer implementation in an LC

In order to obtain the operation of generating logic “0” at all bit outputs of an adder/multiplexer node, an extra AND gate is required. Two selector signals, sel_1 and sel_0, are the inputs of this gate. When both signals are “1”, we make use of a “1” on its output to force on the carry-in of LC that computes the LSB of adder/multiplexer, and thus also on an input of the XOR gate (XORCY) in the carry-chain logic. Meanwhile, a logic “1” is obtained on LUT output connected to the other input of each XORCY gate. This yields “0” at the output s. The logic of LUT output also selects carry-in value pass through multiplexer (MUXCY) for forcing the carry-in of the next upper cell to operate in a similar manner.

B one-bit adder/multiplexers are implemented vertically providing a B-bit adder/multiplexer to perform fast carry-chain addition and multiplexing. The carry-in signal is applied to at the bottom LC of the structure, and is cascaded upwards by the Virtex architecture. This allows the design to have minimum propagation delay.

The proposed structure is an efficient implementation on the Virtex device by fully utilizing the resources of the cell. This provides more functionality with minimal extra cost required. Although an additional AND gate is required, this is a very small logic overhead compared to the size of B-bit add/multiplexer.

5 Transformation into ILP Formulation

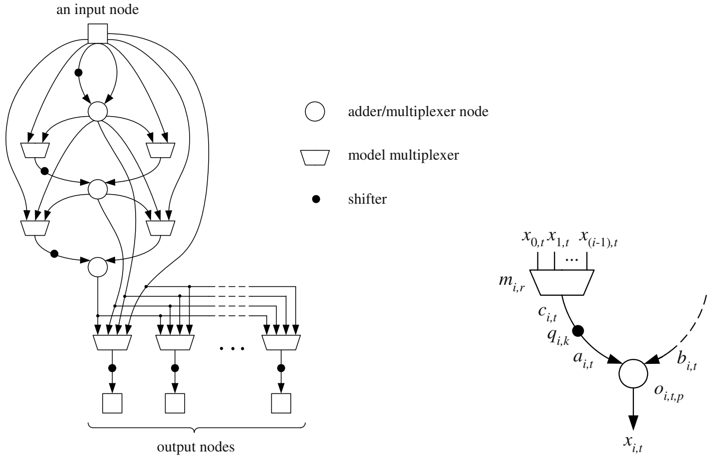
A given set of MRMs may or may not be implementable using a fixed number N of computational nodes. In this section, we propose an ILP model, the solution to which corresponds to a time-ordered sequence of multiplexer select lines to implement the required behaviour, if one exists.

5.1 Representing General DFGs

Fig. 6(a) depicts a general structure for describing all computations containing three adder/multiplexer nodes (higher node structures can be developed in a similar fashion). For clarity, we use a square box to represent each input and output node, and a path with a big black dot to perform shift operation. The multiplexers in Fig. 6(a), labelled "model multiplexers", will not be realized in the final circuit; they provide a model for ILP problem thus allowing all DFGs of N nodes to be modelled. Once implemented, these multiplexers are replaced by wires, as only one value of the select lines is active, for all time. The proposed model therefore contains three main components: shifter, adder/multiplexer and model multiplexer. This structure can perform various operations depending on path selection of all adder/multiplexers and model multiplexers. The number of outputs, which we shall denote C, corresponds to the number of sets of time-varying coefficient(s).

5.2 Encoding the Problem

An instance of the problem is encoded as a $T \times C$ matrix, where T is the number of rows corresponding to the number of time steps and C is number of columns representing outputs. This implies that a $1 \times C$ matrix corresponds to the standard MCM problem. For example, Fig. 1 is a $T \times 2$ problem. Since the MRM problem includes a time element, the first step of our algorithm is to unroll over time. This is accomplished by repeating the general graph. For T-time steps, we require overall T repetitions; all signals that control each corresponding shifter and model multiplexer are tied together. This ensures that shifting and routing for all graphs (all T) are the same. The only select line allowed to change with the time is the select line internal to each adder/multiplexer node, which can be changed to achieved the desired output values.



(a) (b)

Fig. 6. (a) A general DFG of three adder/multiplexer nodes. (b) A portion of general structure.

5.3 ILP Modelling

ILP models are able to provide a formal method to describe and solve the MRM problem. Our approach is to model the N-node problem as an ILP for fixed N, and then to iterate ILP solver to find the lowest value of N resulting in a feasible solution. For minimizing the $T \times C$ matrix problem, suppose that there are total N add/multiplexer nodes operating in a B-bit number system.

A portion of the general structure is depicted in Fig. 6(b) and its notations, described below, will be required for understanding the ILP model.

Both integer and binary variables are used within the model:

- The integer variables $a_{i,t}$ and $b_{i,t}$ are the inputs of i^{th} adder/multiplexer node at step t and its output is represented by variable $x_{i,t}$. The model multiplexer has i inputs, corresponding to the previous node outputs, and its own output is represented by variable $c_{i,t}$.
- The binary decision variables $o_{i,t,p}$ represent which of the four operations to be performed at adder/multiplexer node i during time step t, where $p \in \{0, 1, 2, 3\}$. Variables $m_{i,r}$ represent the selection of input $x_{r,t}$ to the model multiplexer, where $r \in \{0, 1, \ldots, i-1\}$. Finally, variables $q_{i,k}$ represent the degree of shifting : $q_{i,k} = 1$ means that this input of node i should be shifted left by k bits, where $k \in \{0, 1, \ldots, B-1\}$.

We therefore have the following constraints, which are not yet in linear form:
1. Model multiplexer function

$$c_{i,t} = x_{r,t} \text{ if } m_{i,r} = 1. \tag{1}$$

2. Shifter function

$$a_{i,t} = 2^k c_{i,t} \text{ if } q_{i,k} = 1. \tag{2}$$

3. Adder/multiplexer function

$$x_i = \begin{cases} b_{i,t} & \text{if } o_{i,t,0} = 1 \ \ \text{(operation 0)} \\ a_{i,t} & \text{if } o_{i,t,1} = 1 \ \ \text{(operation 1)} \\ a_{i,t} + b_{i,t} & \text{if } o_{i,t,2} = 1 \ \ \text{(operation 2)} \\ 0 & \text{if } o_{i,t,3} = 1 \ \ \text{(operation 3).} \end{cases} \tag{3}$$

A problem now arises since all above constraints are nonlinear problems, due to the "if" selectors. However, these constraints can be reformulated as linear constraints in the following way. For example, in the model multiplexer,

$$c_{i,t} = x_{r,t} \text{ if } m_{i,r} = 1 \quad \Rightarrow \quad c_{i,t} - x_{r,t} = 0 \text{ if } m_{i,r} = 1 \tag{4}$$

which is equivalent to

$$c_{i,t} - x_{r,t} \leq \alpha(1 - m_{i,r}) \tag{5}$$

and

$$c_{i,t} - x_{r,t} \geq \beta(1 - m_{i,r}). \tag{6}$$

where α and β are known finite lower and upper bound on the left-hand side of (5) and (6), respectively. For the unsigned binary number system, $\alpha = 2^B - 1$ and $\beta = -2^B + 1$ are sufficient. We can see that $m = 1$ reduces (5) and (6) to (4). When $m = 0$, $c_{i,t}$ and $x_{r,t}$ can be any values (0 to $2^B - 1$) and still satisfy (5) and (6). Extending this approach to other constraints results the reduction of (1)– (3) to linear constraints problem to be an ILP.

There are a number of additional equality constraints that need to be added:

$$\text{For all nodes } i, \qquad \sum_{r=0}^{i-1} m_{i,r} = 1 \tag{7}$$

$$\text{For all nodes } i, \qquad \sum_{r=0}^{B-1} q_{i,k} = 1 \tag{8}$$

$$\text{For all nodes } i \text{ and time steps } t, \qquad \sum_{p=0}^{3} o_{i,t,p} = 1 \tag{9}$$

where constraint (7) states that multiplexer has to select only one input, (8) states that shifter must be shifted by only one k, (9) states that only one operation has to be performed at any one time step.

The minimum area solution for a $T \times C$ problem can be obtained by proceeding as follows:

1. Set $N = 1$.
2. Determine whether a feasible solution exists.
3. If it does, terminate the process, otherwise increase N and repeat from step 2.

6 Results

We compare our approach to two methods: the ROM and general multiplier approach shown in Fig. 1(b), and the unfolded use of MCM. The latter approach consists of using MCM to create the optimum implementation of all TC coefficients (for $T \times C$ problem), and then using T-to-1 MUXes to select the output at each time step.

Note that both comparative approaches can be considered as special cases of the general MRM structure used. $N \times M$ general multiplier can be considered as a cascade of M N-bit adder/multiplexer nodes, where the shift is always unity and the select line, controlled by the equivalent bit in the multiplicand, selects between options 1 and 2 in (3). The MCM-MUX comparative approach is also a special case, where MCM is performed by adder/multiplexer nodes always fixed at option 2 of (3), and the multiplexing is performed by adder/multiplexer nodes which can choose between options 0 and 1 of (3).

These approaches were tested using sets of 4-bit coefficients generated randomly. All ILP models are solved using the MOSEK optimization software [11]. Table 1 shows the synthesis results of average area and delay targeting Xilinx Virtex-II XC2V1000-4 device [9].

As with the MCM problem, it is expected that the area improvement grows with problem size [8]. However, even for the small benchmarks, our approach compared to MCM-MUX provides less area for the larger instances shown in Table 1. Compared to ROM and multiplier, the crossover point occurs at the 3×3 problem which results in a 24% improvement. It is likely that for large problems, even greater saving will be possible.

Table 1. Average area and propagation delay. The upper figure is the area (in slices), the lower figure is the delay (nanoseconds)

		ROM & Multiplier			our approach			MCM-MUX		
		Number of outputs C			Number of outputs C			Number of outputs C		
		1	2	3	1	2	3	1	2	3
Number of time steps T	1	3.3, 12.27	4.7, 11.66	6.4, 11.71	5.9, 16.65	6.5, 16.51	8.9, 17.71	5.9, 16.59	6.5, 15.92	8.9, 17.32
	2	6.6, 15.22	11.4, 17.18	19.4, 16.91	10.2, 18.26	12.6, 19.14	21.1, 19.99	11.2, 18.15	15.7, 18.17	23.6, 19.15
	3	9.9, 17.05	18.7, 16.82	29.5, 18.04	10.8, 18.89	18.7, 20.47	22.3, 20.70	14.0, 18.53	23.0, 18.24	30.5, 18.02

Since we do not explicitly target delay, the maximum average delay of our approach is 51% longer than that of ROM and general multiplier and 22% of MCM-MUX approach. However, it would be straight-forward to incorperate DFG path length based delay into the ILP objective function, if this were a problem.

7 Conclusion

To our knowledge, there is no existing algorithm to directly deal with the MRM problem. We introduce a new approach to optimize this multiplication problem by formulation into ILP one and employ an efficient ILP software to find the solution. Although such technique have limitations when the problem becomes very large, the results obtained give us important measures of optimality for future developments of a heuristic approach. We also present how to take advantage of all of the hardware present in the Virtex / Virtex-II family to ensure optimal area results. Our further work aims to develop such heuristic approaches, and to exploit dedicated registers for further time-step based optimization.

References

1. K. Matsuura and A. Nagoya. Formulation of the Addition-Shift-Sequence Problem and Its Complexity. In *Proc. 8th Int. Symp. on Algorithms and Computation (ISAAC)*, pages 42–51, Singapore, December 1997.
2. K.K. Parhi. *VLSI Digital Signal Processing Systems : Design and Implementation.* Wiley-Interscience, 1998.
3. J. Villalba, G. Bandera, M. A. Gonzalez, J. Hormigo, and E. L. Zapata. Polynomial evaluation on multimedia processors. In *Proc. Int. Conf. Application Specific Systems, Architectures and Processors (ASAP)*, San Jose, California, 2002.
4. G. Corbaz, J. Duprat, B. Hochet and J.-M. Muller. Implementation of a VLSI polynomial evaluator for real-time applications. In *Proc. Int. Conf. Application Specific Array Processors*, pages 13–24, September 1991.
5. R.H. Turner, T. Courtney, R. Woods. Implementation of fixed DSP functions using the reduced coefficient multiplier. In *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, pages 881–884, May 2001.
6. S.S. Demirsoy, A.G. Dempster and I. Kale. Design guidelines for reconfigurable multiplier blocks In *Proc. Int. Symp. on Circuits and Systems (ISCAS)*, volume 4, pages IV-293–IV-296, May 2003.
7. M. Potkonjak, M.B. Srivastava and A.P. Chandrakasan. Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 15(2):151–165, 1996
8. R. Pasko, P. Schaumont, V. Derudder and D. Durackova. Optimization method for broadband modem FIR filter design using common subexpression elimination In *Proc. 10th Int. Symp. on System Synthesis (ISSS)*, pages 100–106, September 1997.
9. Xilinx Documentation and Literature, available from http://www.xilinx.com/literature/index.htm; accessed 15 March 2004.
10. R.S. Garfinkel and G.L. Nemhauser. *Integer Programming.* Wiley and Sons, 1972.
11. MOSEK ApS optimization software, available from http://www.mosek.com; accessed 15 March 2004.

Area*Time Optimized Hogenauer Channelizer Design Using FPL Devices

Uwe Meyer-Bäse[1], Suhasini Rao[1], Javier Ramírez[2], and Antonio García[2]

[1] Department of Electrical and Computer Engineering
FAMU-FSU College of Engineering, Florida State University, USA
{umb,rsuhas}@eng.fsu.edu

[2] Department of Electronics and Computer Technology
University of Granada, Spain
javierrp@ugr.es, agarcia@ditec.ugr.es

Abstract. Field-programmable logic devices (FPLDs) are on the verge of revolutionizing the digital signal processing (DSP) industry as programmable DSP microprocessor did nearly two decades ago. Historically, FPLDs were considered to be only a rapid prototyping and low-volume production technology. FPLDs are now attempting to move into the mainstream DSP as their density and performance envelope have steadily improved. While evidence now supports the claim that FPLDs can accelerate selected low-end DSP applications, the technology remains limited in its ability to realize high-end DSP solutions. This is primarily due to systemic weaknesses in FPLD-facilitated arithmetic processing. It will be shown that in such cases, a modified carry save adder (MCSA) arithmetic can become an enabling technology for realizing embedded high-end FPLD-centric DSP solutions. This thesis is developed in the context of a demonstrated MCSA/FPLD synergy and the application of the new technology to communication signal processing. Design synthesis results for Xilinx and Altera FPLDs are provided and show 22-164% speed improvement compared to 2C designs and require lower costs (A*T) in most study cases.

1 Introduction

Compared to a cascaded collection of multirate FIRs, a Hogenauer [1] channelizer design (sometimes called, a cascade integrator comb (CIC) filter) can potentially run at a high input data rate and be of lower complexity. CIC filters are popular communication "building blocks" and are available as two's complement arithmetic designs as Xilinx IP block [2] and as a COST IC by Harris/Intersil as HSP43220 [3]. A typical communication configuration for the use as a high decimation rate filter is shown in Fig. 1. The Hogenauer channelizer is well understood but, unlike their simple FIR counterparts, represents a significant design challenge because CIC filters require that

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 384–393, 2004.

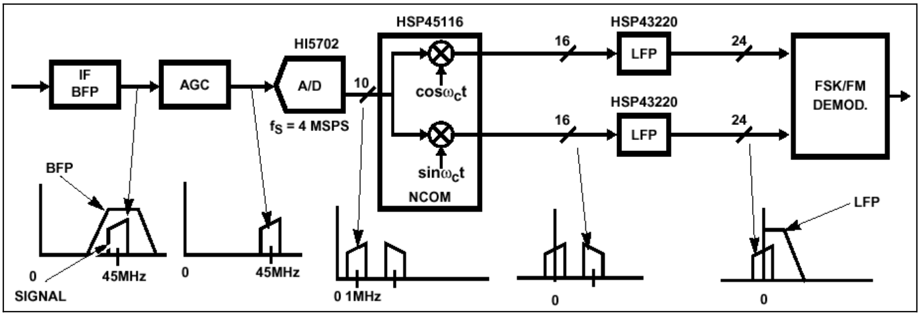


Fig. 1. The Harris/Intersil HSP43320 Hogenauer decimating filter used in IF conversion.

all arithmetic, which can often exceed 66-bits word-widths, be *exact*. Large arithmetic word-widths immediately create a barrier to (non-pipelined) implementation which are generally relegated to low-precision applications (e.g., 8-bits). The design of such a filter using traditional methods and an ASIC/FPLD would be compromised due to the bandwidth and latency problems associated with high precision arithmetic. Our previous attempt [4,5] to solve this design problem involved the use of the residue number system (RNS) arithmetic. Although the RNS implementation improved the speed of the design, the overall cost measured as a product of area and time (area*time) was not favorable. We designed for instance a 3-stage CIC decimator using 26 bits. The 2C design cost metric was 343 LCs/49.3 MHz = 6.9, while the RNS metric was 559 LCs/76.3 MHz = 7.3. In addition the RNS design had benefited from the following two assumptions:

- A second pole at $\omega=\pi$ was introduced to improve the speed of the modulo adders.
- The output conversion from RNS arithmetic to two's complement was not included in the area calculation of the design,

while the first assumption may be valid due to the mandatory anti-aliasing filter in front of the CIC, the second assumption is only valid if the next processing step is also done in RNS. If the CIC filter is used as an embedded filter application within a two's complement arithmetic system, input and output conversion need to be included in the cost (i.e. area*time product) of the design.

A carry save adder (CSA) concept, however, provides a potential solution to this dilemma because CSA does not only provide essential speed improvement, via the absence of any carry propagation in the CSA design and the area penalty for CSA systems is less when compared with the RNS system, yielding an overall better cost measured by the area*time product.

2 Hogenauer CIC Filter Theory

A CIC filter devised by Hogenauer [1] is a multiplier free structure. The principal blocks of a CIC filter are an integrator and a comb or a differentiator with a rate changer in between. The transfer function of a CIC decimation filter with *S* stages is given by,

$$H(z) = H_I(z)H_C(z) = \left(\frac{1-z^{-RD}}{1-z^{-1}}\right)^S \tag{1}$$

where *D* is the number of delays in the comb section and *R* is the down-sampling factor. From the above equation it can be seen that even though the integrators by themselves have an infinite impulse response, a CIC filter as a whole is equivalent to "*S*" moving average FIR filters. Figure 2 shows the step response of a single stage CIC filter without the rate changer. It can be seen that although the response of the integrator is infinity and shows overflow, but the final output y[n] is as expected due to the comb. Hence the filter's response is a moving average defined over *D* contiguous sample values. Such a moving average is a very simple form of lowpass filter.

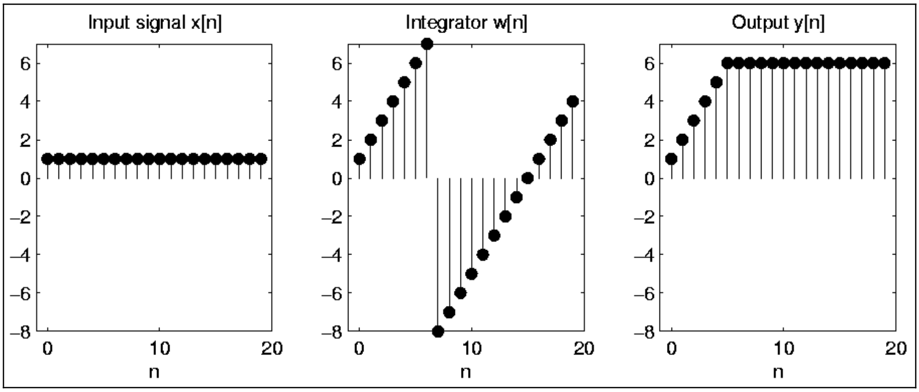


Fig. 2. Step response of the first order Hogenauer filter without decimation.

Due to the presence of integrators and differentiators, register growth is a very important factor. In order to insure that no data is lost due to register overflow, the total internal word-width is calculated using the formula,

$$\begin{aligned} B_{\text{intern}} &= B_{\text{input}} + S \times \log_2(RD) \\ B_{\text{intern}} &= B_{\text{input}} + B_{\text{growth}} \end{aligned} \tag{2}$$

Thus the adder in the CIC filter design is crucial as it has to perform exact arithmetic with this word width at all levels so that no run-time overflow occurs at the output.

2.1 Hogenauer Filter Design Using Carry Save Adders

Carry Save adders (CSA) are popular in array multipliers due to the reduced latency provided with equivalent or superior speed performance. We have employed three different adder designs for the 5-stage, 16 bit input CIC filter design with a rate change factor of 1000 available as commercially IC from Harris/Intersil HSP43220. Each of these designs is synthesized for Xilinx's and Altera's FPLD and the results are tabulated. The CIC filter with two's complement adder uses the least number of logic cells but with the increase in the number of stages and the number of input bits, the adder becomes very slow due to carry ripple. Several techniques for multiple operand addition that attempts to lower the carry propagation penalty have been proposed and implemented [6]. Among these, the CSAs are the fastest since there is no carry propagation until the last stage, while in the other stages a partial sum and a sequence of carries are generated separately. A CSA is nothing but a parallel counter employing parity function, i.e., the k^{th} significant output bit is the parity function of one-bit 2^k tuples in the vector [7,8]. We have incorporated these adders only in the integrator section in our design for study purpose however we have used two's complement addition in the comb section. Due to the presence of the feedback in the integrator, the parallel counter design grows bigger with the number of stages in the CIC design. The first stage has a (3,2) CSA, the second stage has a (5,3) CSA and the third and the consecutive stages will have (6,3) CSAs as shown in the CIC structure in Fig. 3.

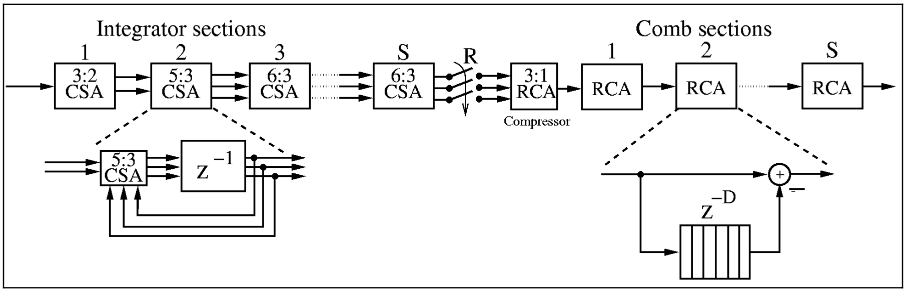


Fig. 3. Cascaded integrator comb filter using carry save adders.

Thus, with increase in the number of stages, the performance of this adder deteriorated using more silicon resource and decreased speed of computation. This drawback was overcome by the use of "Modified CSA" (MCSA) which is obtained by combining multiple (3,2)CSAs in a so-called Wallace tree [6]. In this tree, the number of operands is reduced by a factor of 2/3 at each level. Putting different, the number of operands in level (k+1) can be at most $\lfloor N_k 3/2 \rfloor$. Starting with the level 1 with one (3,2) CSA it follows that the maximum number of operands at level 2 is $\lfloor 9/2 \rfloor$=4. The resulting sequence is therefore 3,4,6,9,13,19,28 etc. For the CIC design 2 levels of CSA are sufficient. Fig. 4 shows the resulting MCSA structure.

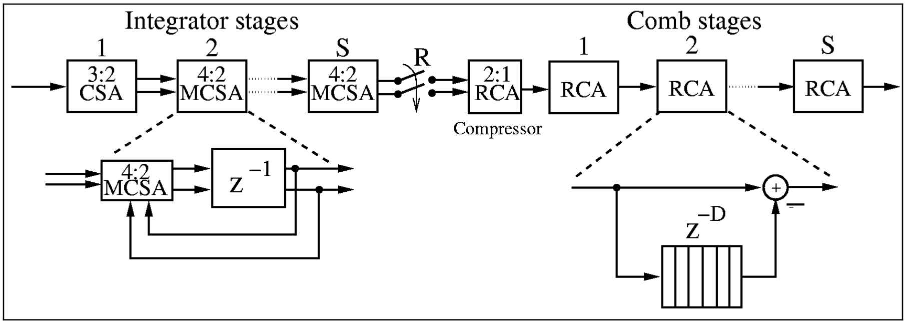


Fig. 4. Cascaded integrator comb filter using modified carry save adders.

In general, input and the output bit width of a CIC filter are in the same range. Hence two different methods are employed in order to make the input and output word-width the same, pruning in the final stage and by pruning some LSBs at the previous stages.

2.2 Hogenauer's Pruning Theory for Two's Complement

The quantization introduced through pruning in the final stage is very large when compared with the quantization introduced in the output by pruning some LSBs at the previous stages. If $\sigma^2{}_{T,2S+1}$ is the quantization noise introduced through pruning in the output, Hogenauer suggested to set it equal to the sum of the (truncation) noise $\sigma^2_{T,k}$ introduced by all previous sections. For a CIC filter with S integrator and S comb sections, it follows that,

$$\sum_{k=1}^{2S} \sigma^2_{T,k} = \sum_{k=1}^{2S} \sigma^2_k P^2_k \le \sigma^2_{T,2S+1} \tag{3}$$

$$\sigma^2{}_{T,k} = \frac{1}{2S} \sigma^2{}_{T,2S+1} \tag{4}$$

$$P^2_k = \sum_S (h_k[n])^2 \qquad k = 1,2,\ldots,2S \tag{5}$$

where P^2_k is the power gain from stage k to the output. Compute next the number of bits B_k which should be pruned by,

$$\begin{gathered} B_k = \left\lfloor 0.5\log_2\left(P_k^{-2} \times \frac{6}{N} \times \sigma^2_{T,2S+1} \right) \right\rfloor \\ \sigma^2_{T,k}\Big|_{k=2S+1} = \frac{1}{12} 2^{2B_\text{k}} = \frac{1}{12} 2^{2(B_\text{in} - B_\text{out} + B_\text{growth})} \end{gathered} \tag{6}$$

The power gain P_k^2 for k=S,S+1,....2S for the comb sections can be computed using the binomial coefficient,

$$H_k(z) = \sum_{n=0}^{2S+1-K} (-1)^n \binom{2S+1-k}{n} z^{-kRD} \qquad (7)$$
$$k = S, S+1, \ldots 2S.$$

2.3 CSA Pruning Technique

From Figures 3 and 4 it can be seen that the CSA and MCSA designs introduce more noise sources than the original two's complement design. More precisely, the MCSA introduces one additional noise source in all integrator sections, i.e., a total of S additional noise sources. The CSA configuration has one additional noise source in the first integrator, while all other CSA integrator sections have two additional noise sources, or a total of (2S-1) additional noise sources. We can take care of this additional noise source by adjusting (4) in Hogenauer's pruning equations. There seemed to be two viable approaches that remove the degradation through the additional noise sources.

In the *error distribution* technique we distribute the additional S or (2S-1) noise sources for MCSA and CSA, respectively, over all stages, including the comb sections, i.e., we replace (4) with

$$\sigma^2{}_{T,k} = \begin{cases} \dfrac{1}{3S}\sigma^2{}_{T,2S+1} & \text{for MCSA} \\ \dfrac{1}{4S-1}\sigma^2{}_{T,2S+1} & \text{for CSA} \end{cases} \qquad \text{for} \quad k = 1,\ldots,2S \qquad (8)$$

In the second approach (the *direct quantization noise adjustment*), we reduce the extra noise in each stage by scaling all noise sources to the allocated noise margin for that stage. We would then replace (4) by

$$\sigma^2{}_{T,k} = \begin{cases} \dfrac{1}{2}\dfrac{1}{2S}\sigma^2{}_{T,2S+1} & \text{for CSA } k=1 \textit{ and } \text{MCSA } k = 1,\ldots,S \\ \dfrac{1}{3}\dfrac{1}{2S}\sigma^2{}_{T,2S+1} & \text{for CSA with } k = 2,\ldots,S \end{cases} \qquad (9)$$

The comb section will be unchanged in this case.

The `cic.exe` program from [9] was modified in order to compute the modified bit width for the CSA and MCSA designs using the above two methods. The program provides the maximum bit growth as well as the number of bits to be retained at each stage for the CIC design using pruning. For the 5-stage design with 16 bit input and output bit width and a rate change factor of 1024, B_{max} is 66. The result of executing this program is tabulated below in Table 1. We note that one more guard bit in the integrator section and comb sections is sufficient to implement the (M)CSA design with the same quantization error as the two's complement design. Comparing the error distribution techniques with the direct method we see that error distributions yield larger required bit width in the comb sections and therefore we used the direct quantization method (shown bold in Table 1) without error distribution for our designs.

Table 1. Carry save adder pruning data.

Type	Distribute Error	Integrator sections					Comb sections				
		1	2	3	4	5	1	2	3	4	5
2C	No	63	53	43	35	26	22	21	20	19	19
CSA	Yes	63	53	44	35	27	23	22	21	20	19
MCSA	Yes	63	53	44	35	27	22	22	21	20	19
CSA	No	63	53	44	35	27	22	21	20	19	19
MCSA	No	63	53	44	35	27	22	21	20	19	19

3 Synthesis Results

Circuits for 5-stage CIC filters using 2C, MCSA, and CSA arithmetic in full bit width and pruning technique have been developed using generic VHDL coding. Circuits have then been synthesized from their VHDL descriptions and optimized for speed and area using synthesis tools from Altera and Xilinx. To have an first impression on the possible performance gain we have compiled the data for single adders using CSA, MCSA, and 2C which are graphically interpreted in Fig. 3. We note the speed improvement especially for large bit-width adders of the CSA and MCSA when compared with the two's complement adder (2C).

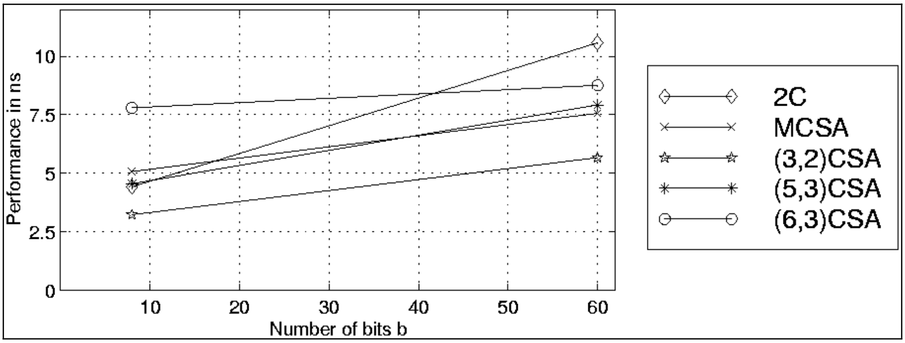


Fig. 5. Result of synthesis of the adder designs on Xilinx's FPLD.

3.1 CIC Xilinx Synthesis Results

The synthesis results for the Xilinx Virtex Device XCV300e-pq240-6 compiled with the ISE web-pack tool set shows similar results as with Altera devices and software. Because the Xilinx logic cells have two 4-input and one 3-input tables, the design area used is the equivalent number of gates from the Xilinx "Mapping Report File." To have reliable timing data we use the "Post Place&Route Static Timing Report" rather than the map time estimations. As mentioned before, when designed using a two's complement adder, the number of inputs to each stage remains the same regardless of the number of stages. Whereas, when designed using a CSA, the first stage has (3,2) CSA, the next stage uses (5,3) CSA and the third (6,3) CSA and for further stages it remains the same. When used with MCSA, the first stage has (3,2) CSA, the second stage uses (4,2) MCSA and this remains the same for all the further stages.

Table 2. Synthesis data of 5-stage CIC filter on Xilinx's FPLD.

Design	**Opti-miza-tion**	Data no pruning				Data with pruning			
		clk/ ns	Gates	clk* Kgates	R>	clk/ ns	Gates	clk* Kgates	R>
CIC5	Speed	12.64	15917	201.24	2	9.82	16093	158.05	1
	Area	12.17	15901	193.48	2	10.70	16059	171.78	2
MCSA5	Speed	5.88	23329	137.20	3	7.52	23517	176.85	2
	Area	11.42	23297	266.08	2	7.41	23455	173.83	2
CSA5	Speed	9.96	45167	449.95	2	8.60	21531	185.12	2
	Area	10.40	45063	468.84	2	9.52	21358	203.24	2
Gain % CIC/MCSA		106.9		41.0		32.5		-9.1	
Gain % CIC/CSA		22.1		-57.0		14.2		-14.6	

The design field indicates the CIC filter design with the best synthesis option using CSA (with parallel counter logic), MCSA (as in Modified CSA) and CIC (using two's complement adder).

We notice the speed improvement both for the CSA as well the MCSA design with and without pruning. The cost measured by the time*area product is improved only for the MCSA design without pruning. Table 2 also includes the required minimum sampling rate reduction between the integrator and comb section as measured by the quotient of integrator clock and comb clock, i.e., ceil($\mathrm{clk}_I/\mathrm{clk}_C$). For all designs the required minimum sampling rate reduction is 3, which is most likely well below the usual high decimation rate factor CIC are used in communication systems.

3.2 CIC Altera Synthesis Results

The synthesis results of 5-stage CIC filters using the above mentioned means of arithmetic is shown in Table 3. The designs are synthesized for Altera's FPLD device EPF10K130EQC240-1. The designs include both the pruning methods mentioned above. The best performance of each of these designs for the different speed and style options is tabulated. LCs gives the number of logic cells used, Fmax is the Registered Performance, and Cost = LCs/Fmax(10^{-6}) gives the cost of the design. From the above table it can be seen that pruning at each stage decreases the total LCs used and thereby improves the speed. CIC filter design using two's complement adder uses minimal resource but at the same time the speed is the least compared to other designs. Though MCSA uses twice the number of LCs, the speed is three times faster than the design using two's complement adder, thereby making the design more cost effective.

Table 3. Synthesis data of 5-stage CIC filter on Altera's FPLD.

Design	**LCs**	**Fmax/ MHz**	**LCs/ Fmax**	**Synthesis Options**	**LCs**	**Fmax/ MHz**	**LCs/ Fmax**	**Synthesis Options**
Cic5r	1062	48.54	21.88	Fast-10	467	52.08	8.97	Fast-0
Csa5r	5884	95.23	61.79	Norm-10	2761	96.15	28.72	Norm-10
Mcsa5r	2138	128.2	16.68	Fast-10	1145	135.13	8.47	Fast-0
Gain % CIC/CSA		96.19	-64.59			84.62	-68.77	
Gain % CIC/MCSA		164.1	31.20			159.47	5.83	

Table 3 shows only the best results regarding cost metric area*time. For a complete listing including the optimum synthesis results for maximum speed optimization we reference to [10].

4 Conclusions

The Hogenauer's [1] design of two's complement cascade integrator comb filter was extended to carry save adder design. Using a digital signal processing scheme with CSA provides fast filter building blocks. These filters are of low complexity and are multiplier free, so that fast compact decimators and interpolators can be implemented without the high cost of RNS implementation as previously proposed [4,5].

The quantization error analysis for CSA shows that no more than one additional guard bit precision is needed when compared with Hogenauers pruning for two's complement.

Synthesis results for a typical design example used in the Harris/Intersil HSP43220 have been compiled and show an improvement in speed from 84% to 164% and up to 31% costs improvements for Altera FPLDs. Improvements in speed from 22% to

106% and up to 41% for costs metric (A*T) for Xilinx Virtex FPLDs when compared with the conventional two's complement design are reported.

Acknowledgements. The authors would like to thank Altera and Xilinx for their support under the University programs. A. García was supported by the Ministerio de Ciencia y Tecnología (MCyT, Spain) under project TIC2002-02227.

References

1. E.B. Hogenauer, "An Economical Class of Digital Filters for Decimation and Interpolation". IEEE Transactions Acoustics, Speech and Signal Processing, Vol. 29(2) (1981) 155-162
2. www.xilinx.com/ipcenter
3. http://www.intersil.com/design/parametric/deviceinfo.asp?pn=HSP43220
4. U. Meyer-Baese, A. Garcia, F. Taylor, "Implementation of a Communications Channelizer using FPLDs and RNS Arithmetic", Journal of VLSI Signal Processing, Vol. 28, (2001) 115–128
5. García, A., Meyer-Bäse, U., Taylor, F. J., "Pipelined Hogenauer CIC Filters Using Field-Programmable Logic and Residue Number System," IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, WA, Vol. 5, (1998) 3085-3088
6. I. Koren, "Computer Arithmetic Algorithms", Prentice Hall, Eaglewood Cliffs, New Jersey (1993)
7. M. Mehta, V. Parmar, E. Swartzlander Jr, "High-Speed Multiplier Design using Multi-Input Counter and Compressor Circuits", in Proceedings of the 10th International Symposium Computer Arithmetic*,* (1991) 43-50
8. Y. Leblebici, H. Ozdemir, A. Kepkep, U. Cilingiroglu, "A compact High-Speed (31,5) Parallel Counter Circuit Based on Capacitive Threshold-Logic Gates", IEEE Journal of Solid-State Circuits, Vol. 31(8), (1996) 1177-1183
9. U. Meyer-Baese, "Digital Signal Processing with Field Programmable Gate Arrays", Springer-Verlag, Heidelberg (2001)
10. S. Rao, "Multirate Filter Design on Field Programmable Gate Arrays," Master's Thesis, Florida State University, Tallahassee (2003)

A Steerable Complex Wavelet Construction and Its Implementation on FPGA

C.-S. Bouganis[1], P.Y.K. Cheung[1], J. Ng[2], and A.A. Bharath[2]

[1] Department of Electrical & Electronic Engineering, Imperial College,
Exhibition Road, London SW7 2BT, U.K.
[2] Department of Bioengineering, Imperial College,
Exhibition Road, London SW7 2AZ, U.K.

Abstract. This work addresses the design of a novel complex steerable wavelet construction and its implementation on reconfigurable logic. The wavelet decomposition uses pairs of bandpass filters that display symmetry and antisymmetry about a steerable axis of orientation. The design is targeted for implementation in hardware, thus one of the desired properties is the small number of unique kernels. A detailed description of the implementation of the design in hardware is given. Moreover, results regarding the speed of our design compared to a software implementation, and the error in the filter responses due to fixed point representation, are reported. To show the applicability of the design to real life situations, a corner detection algorithm is illustrated.

1 Introduction

The applications of wavelets to signal and image compression are well researched [1,2,3]. The work described here contains several points of departure in both the construction and application of steerable filters to feature detection. The main point of departure is that the filter kernels are specified by separable angular and radial functions in the frequency domain which have not been jointly reported in a multi-rate scheme. In addition, an implementation of the algorithm in hardware is performed, targeting real-time applications.

The need for more flexibility and fast prototyping of signal processing algorithms has lead the FPGA community to investigate tools for easy mapping of signal processing algorithms to FPGAs. One approach is to provide the designers with building blocks that are common in DSP applications [5]. Another approach is to provide tools that allow the engineers to describe their design in a high level language [6]. In this work, Handel-C [9] is used as the main tool to describe the steerable complex wavelet pyramid on hardware. Handel-C is based on the syntax of ANSI C with additional extensions in order to take advantage of the specific characteristics of the hardware. It is independent of the targeting platform which makes the design easily transferable to other hardware platforms. The originality of this work is in the design of a novel steerable wavelet construction and the investigation of mapping such a design to reconfigurable logic, targeting real-time applications.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 394–403, 2004.

The paper is organized as follows. In section 2.1 the motivation for implementing a complex wavelet decomposition is given. In section 2.2 a detailed description of the pyramid is given as well as how this design differs from other constructions. In section 3, the application of the pyramid to feature detection is demonstrated using as example a corner detection algorithm. The implementation of the design on FPGA is described in section 4. Finally, the impact of the quantization of the variables on the overall performance of the algorithm and a comparison between the hardware implementation to the software implementation of the algorithm regarding the speed is given.

2 The Pyramid Design

2.1 Motivation

The motivation for a new pyramidal decomposition has been a subband decomposition of images into orientation and scale-selective channels that can then be used for analysis purposes. Although there are numerous decompositions that satisfy such a requirement, we are seeking a construction that utilises polar separable functions, so that the orientation selectivity can be specified independently of radial frequency (or scale selectivity), and at the same time a small number of unique kernels for construction is required for implementation of the design in hardware. To provide a variety of scale-selective channels, we have chosen to rely on standard multi-rate techniques, which enhance the computational and representational efficiency of such decompositions. Within each orientation and frequency channel, we wish to estimate the local image symmetry/antisymmetry about an axis. Using a small number of kernels, the axis should be tunable, depending on the local image content. These requirements are met by a steerable quadrature wavelet decomposition, of which some examples can be found in [3,7].

2.2 Design Overview

The design of the pyramid employs decimation in the lowpass channel in order to achieve the scaling of filter response through repeated application. The nature of the decomposition is illustrated in Figure 1. The decomposition is repeated four times in order to detect symmetric and antisymmetric regions in the image in different scales. The design of the filter kernels is performed in the Fourier domain and the inverse two-dimensional Fourier Transform is applied to compute the spatial impulse responses. For convenience in tuning angular and radial characteristics of the filters, we impose Fourier domain polar separability, so that a filter $G_{0,k}(\omega, \phi)$ in the k^{th} direction in a filter set can be specified as the product of a radial frequency function $\Omega_0(\omega)$ and an angular frequency function $\Phi_{0,k}(\phi)$, i.e. $G_{0,k}(\omega, \phi) = \Omega_0(\omega)\Phi_{0,k}(\phi)$.

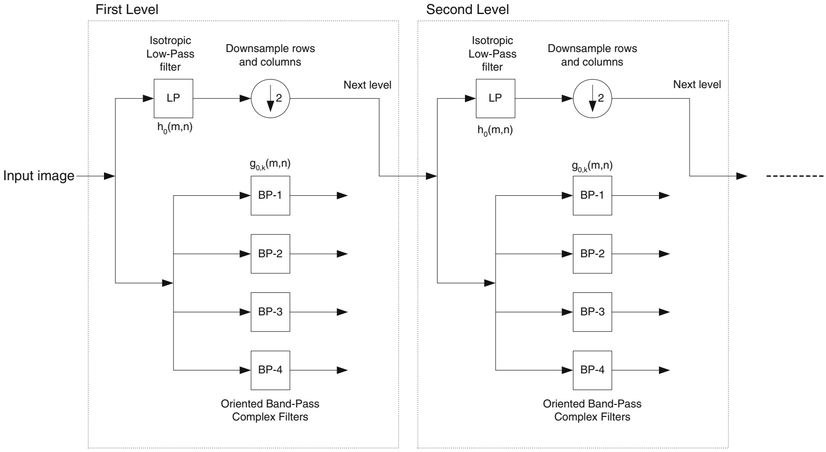


Fig. 1. Pyramid layout. The boxes with the dotted lines illustrate a single level of decomposition. The full decomposition consists of four levels. The responses of the bandpass filters detect symmetric and antisymmetric features in the image.

2.3 Radial Frequency Response

For the isotropic lowpass radial frequency response we have used the following function in the radial frequency domain on the interval $-\pi < \omega \leq \pi$.

$$H_0(\omega, \phi) = H_0(\omega) = \frac{1}{1 + (\omega/\omega_c)^6} \tag{1}$$

where $\omega_c = 3\pi/8$. It was chosen to provide a reasonably flat power response, when used in combination with the bandpass radial frequency response, defined later, for radial frequency components in the range $[0, \omega_{max}]$. ω_{max} is the peak frequency of the bandpass radial frequency response.

The radial response of the bandpass filters, $\Omega_0(\omega)$, is based on Erlang functions which are one sided, smooth, and have the property that $\Omega_0(0) = 0$. The joint localisation of Gaussian kernels in both spatial and frequency domain causes transform coefficients to fall off in magnitude as scale is increased [4]. This is undesirable for a hardware implementation since more coefficients are required to represent the kernels of the pyramid. Using an α value smaller than one (Poisson and Erlang, $\alpha = 0.5$) biases the localisation towards the frequency domain and provides an increased stability of transform coefficients across scales. The filters employed here have radial frequency response

$$\Omega_0(\omega) = \left(\frac{e}{14}\right)^7 \omega^7 e^{-\omega/2} U(\omega) \tag{2}$$

where $U(\omega)$ is the unit step function.

2.4 Angular Frequency Response

The prototype general angular frequency characteristic

$$\Omega_0(\phi) = \cos^3(\phi) rect(\phi/\pi) \tag{3}$$

has been used, where $rect(\phi) = U(\phi + \frac{1}{2})U(\frac{1}{2} - \phi)$. This is a generalization of the sin and cosine angular characteristics used in derivative of Gaussian processing, but with tunable angular selectivity. The prototype angular frequency is rotated to generate the angular characteristics of oriented filters for a full filter set by the following:

$$\Phi_{0,k}(\phi) = \Omega_0(\phi - \phi_k) \tag{4}$$

In our design four orientations are used at $0, \pi/4, \pi/2$ and $3\pi/4$.

2.5 Filter Kernels

For each of the filter prototypes in the Fourier domain, a sampling on the two-dimensional interval $[-\pi, \pi] \times [-\pi, \pi]$ was performed, with a grid spacing of $\pi/64$ in each cartesian direction. The choice of an odd matrix size for constructing the Fourier domain representation is tied to the symmetry of the filter kernels, which we have observed to be better on odd-sized grids.

The inverse two dimensional discrete Fourier Transform was computed to extract 65×65 spatial frequency responses. These responses were each truncated to fit a set of four 7×7 complex arrays. The larger the size of the kernels the better the properties of the filters are preserved, but more area is required to implement these kernels on hardware. The kernels thus extracted are illustrated in Figure 2. The first row corresponds to the real component of the kernels which detect even-symmetric features in the image such as lines. The second row corresponds to the imaginary component responsible for detecting the parts of the image with odd-symmetric content, such as edges. The symmetry properties of the filters fall into various classes. We have identified five classes of coefficient symmetries. More details about kernel construction, the symmetry classes of the filters and the coefficients of the filter blocks can be found in [10,11]. However, the current hardware design has not been optimized with respect to these symmetric properties. Future work will take into account these properties in order to reduce the computational load in the FPGA.

3 Generating Feature Maps

3.1 Corner Likelihood Response

The output of the filters may be used to generate a measure that may be treated as being proportional to the likelihood of a particular location in an image being the corner of some structure. We construct the following feature map

$$C^{\ell}(m,n) = \frac{\prod_{k=0}^{3} |f_k^{(\ell)}(m,n)|}{p + \sum_{k=0}^{3} |f_k^{(\ell)}(m,n)|^4} \tag{5}$$

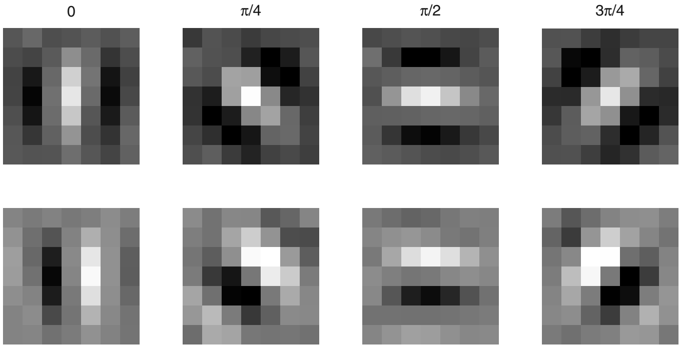


Fig. 2. Real and Imaginary parts of complex bandpass filter kernels. Top row shows the real part of the kernels, where the bottom row shows the imaginary part.

$f_k^{(\ell)}(m,n)$ denotes the response of the filter k of level ℓ. m, n denote the indexes inside the image. p is fixed to be 4% of the maximum pixel intensity in the image preventing the feature map to take large values for small values of $\sum_{k=0}^{3} |f_k^{(\ell)}(m,n)|^4$. The denumerator normalizes the response to a local anisotropic energy. Moreover, we may choose to weight the corner response by anisotropic energy computed at the same, or another scale.

4 Implementation on FPGA

The complex steerable wavelet was designed to be "hardware-friendly" by targeting to a minimum number of distinct and symmetric kernels. A hardware implementation using reconfigurable logic was investigated to accelerate the decomposition part of the algorithm which leaves "high-level" decisions such as the implementation of the feature maps to the host CPU. Only one level of the pyramid is implemented in hardware, and the full decomposition is realised through reuse of the same hardware, having as input the decimated image from the previous iteration.

The target board that is used for implementation is the RC1000-PP from Celoxica. It is a PCI bus plug-in card for PC's with a Virtex V1000 FPGA and four memory banks of 2 MBytes each. All four memory banks are accessible by both the FPGA and any device on the PCI bus. However, at any time instance only one device can access a memory bank. The Handel-C language is used to describe the design.

4.1 FPGA Design

The quantization of the variables in the design is as follows: eight bits are used to represent a pixel in the image and ten bits are used to represent the coefficients of each filter and also the output of each convolution. The impact to the final accuracy of the algorithm by selecting these numbers of bits to represent the variables is discussed in section 4.3. In order for the decomposition to be performed as fast as possible, the whole design is pipelined in order to produce three convolution results per clock cycle.

Figure 3(a) shows an overview of the design. The pixels are stored in a raster scan in sets of four in the memory allowing the FPGA to fetch four pixels per read cycle. The *ManageBuffer* process is responsible to buffer the data such as to provide a region of the image to the next process in the design. The FIFOs are mapped to block RAMs in the FPGA for a more effective use of resources. The next process, the *ProcessWindow*, performs the convolution between a window in the image and the appropriate masks. It contains three programmable processes *FilterBankA (FBA)*, *FilterBankB (FBB)* and *FilterBankC (FBC)* that each one can apply three different filters by loading a specific set of coefficients. A shift register and a RAM to store the coefficients is selected to form the appropriate masks for each level of the pyramid. The final results are concatenated and stored in the external RAM. Figure 3(b) shows a detailed diagram of the *FilterBank* process. Moreover, the result from the last filter, which represents the input image for the next level of the pyramid, is decimated, saturated in the range $[0, 255]$ and stored in the external memory by the *NextLevelImage* process.

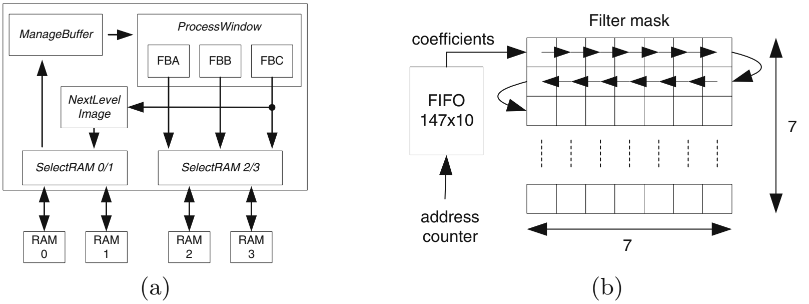


Fig. 3. (a) shows the top level diagram of the design. (b) shows the *FilterBank* process. The FIFO contains the coefficients for the three kernels that are realised in the filter bank.

4.2 Host Control

The CPU controls the operation of the FPGA by a handshake protocol. Due to the associated latency of each transfer through the PCI bus, the data are transferred using DMA access between the CPU and the board [12]. In order to speed up the process, the decomposition of the image and the transfer of the data to/from the host are performed in parallel. The following scheme is used. Out of the four memory banks, the first two are used to store the new frame that is sent by the host for processing, the previous frame that is being processed by the FPGA, and the decimated images that are used for the different levels of the pyramid. The other two banks are used by the FPGA to store the result of the convolutions. The handshake protocol operates as follows. When a new frame is available, the CPU sends the data to RAM 0 or 1 and signals to the FPGA that a new frame is waiting for processing. In the meantime, the FPGA processes the previous frame from RAM 1 or 0 respectively. The results from the convolutions are stored in RAMs 2 and 3. The output data are distributed between RAMs 2 and 3 such that while the FPGA writes the results from a convolution to one RAM the CPU performs a DMA transfer to the already calculated results from the other RAM. The distribution of the results is necessary, since the design should be able to handle images with size 640 by 480 pixels.

4.3 Implementation Analysis

Experiments were performed to investigate the impact of the number of bits that are used to represent the kernel coefficients (N_c) and the bits that are used to represent the result of the convolution (N_o) to the filter responses. The mean square error of the estimation of each filter response between full precision and fixed point for each combination of N_c and N_o is estimated using the Lena image. Figure 4 shows the average mean square error over all filters using the same combination of N_c and N_o. In our design, N_o is set to 10 in order to be able to store the results of three parallel convolutions by performing only one 32-bit access to the external memory. From the figure, it can be concluded that the number of bits used for the coefficients has a small effect on the error of the response compared to the number of bits used to represent the result of the filters. Also, it should be mentioned that the error in the filter responses increases after changing levels since the decimated result of the low-pass channel is reused for the next bandpass decomposition.

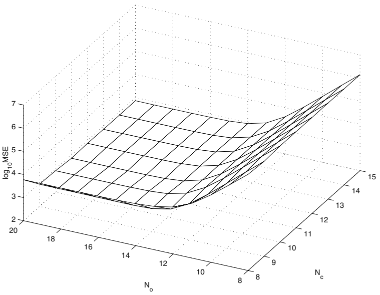


Fig. 4. Mean square error in the filters' response using the Lena image between fixed-point and floating-point arithmetic.

The overall design uses 12,286 slices. Due to the large size of the design compared to the available space in Virtex V1000, the optimum clock rate can not be achieved. The synthesis results of the design using Xilinx ISE 6.1 gives 99%

usage of slices. It is clear that there is not enough available space for optimum routing, which reduces the optimal clock frequency. Furthermore, the slow RAMs that are available on the board reduce the effective speed of the design. Due to the nature of the decomposition, the data that are generated correspond to an equivalent size of 14.6 times the input image. This amount of data cannot be stored in the internal memory of the FPGA and should be transferred to the external RAMs. The available bandwidth to the external memories reduces the effective speed of the current design. The required memory bandwidth by the design for VGA resolution (640x480) at 25 frames per second is 109 MBytes/sec, where the available bandwidth is 66 MBytes/sec assuming only one available memory bank. A rate of 16.6MHz was achieved giving 13.1 frames per second in VGA resolution.

5 Results

5.1 Performance Analysis

Experiments were performed to compare the speed of the new design to a software implementation. A decomposition with four orientations and four levels is performed on two test images with size 256x256 and 512x512. Table 1 shows a summary of the results. The first row of the table corresponds to a machine with Dual Hyperthreading Xeons at 2.66GHz with 2GB of RAM. The software runs under MATLAB and it is optimized using the *Intel SIMD Integrated Performance Primitives* library which also takes advantage of multiprocessors. The second row corresponds to a similar machine but without hyperthreading technology. The software version of the design was implemented using single, dual and quad threads. The RC1000-PP board is placed on a Dual Pentium III machine at 450MHz and 512MB of RAM. The results for the software is the average over 40 frames, where for the hardware the results is the average of 4000 frames. The timing for the FPGA include the DMA transfers. In both cases the required time to read the data from the hard disk is excluded. The *speed up factor* is calculated with respect to the best performance of the software implementation in each row. It can be seen that an average improvement of 2.6 times in the speed can be achieved. Moreover, we placed our design in an XC2V6000 to investigate how fast the current design can be clocked without any restrictions from the size of the FPGA device or by the timing constraints of the external memories. The synthesis tool showed that the design can be clocked up to 50MHz giving an average *speed up factor* of 8 compared to the software implementations.

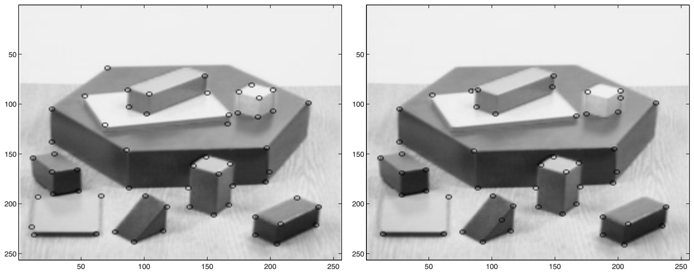
5.2 Corner Detection

Further experiments are performed to assess the performance of the design to real-life situations. The application under consideration is corner detection using the algorithm described in section 3. We investigate how precisely the corner of a structure in the image is detected given the limited number of bits that are used

Table 1. Comparison results in speed between software and hardware implementation. Using a XC2V6000 device a *speed up factor* of 8 is achieved.

Image size	HT	Single Thr.	Dual Thr.	Quad Thr.	FPGA Accel.	Speed up factor
Lena 256x256	Yes	0.06035s	0.04897s	0.05088s	0.0170s	2.88
	No	0.05953s	0.04737s	-		2.78
Boats 512x512	Yes	0.21074s	0.21281s	0.16113s	0.0653s	2.46
	No	0.20720s	0.15724s	-		2.40

to represent the coefficients and the response of the filters. Figure 5 shows the performance of the above design compared to a software implementation. The image on the left is the result of the corner detection when the whole algorithm is implemented in software. The image on the right is the result of the corner detection when the decomposition of the image is performed in the FPGA. It can be seen that most of the features have been detected correctly except of 8 mismatches. Further investigation revealed that by assigning 16 bits to represent the output of the filters gives zero mismatches. However, a 16 bit representation for the results would involve access to two memory banks simultaneously, forcing the FPGA to wait for each DMA transfer to finish. This results in a reduction in performance by a factor of 1.5, using the RC1000-PP board.



(a) (b)

Fig. 5. (a) shows the result of the corner detection using software. (b) shows the same result using the hardware implementation.

6 Conclusions

In this paper we have presented a novel steerable pyramid for image decomposition and feature detection. For speeding up the algorithm, a mapping to reconfigurable logic was performed. We investigated the impact of the quantization of the variables to the filter responses and pointed out potential problems in the design of such multi-level transforms. Due to the nature of the algorithm, a huge amount of data is produced and can be stored only in the external RAMs. The current design is limited by the available bandwidth to the external memories.

The current prototype will be used as a platform for research in word-length optimization [8] over multiple coefficient masks that use the same paths. Moreover, future work involves the investigation of automated tools that optimize the design of wavelet transforms taking into account the symmetry properties of the filters in the case where the same part of hardware is used by different kernels.

Acknowledgement. The authors would like to thank George Constantinides from Imperial College for his help and support. Also, we would like to thank Wayne Luk and Altaf Abdul Gaffar for lending us the RC1000-PP board. This work was funded by the UK Research Council under the Basic Technology Research Programme "Reverse Engineering Human Visual Processes" GR/R87642/02.

References

1. L. Sendur and I. W. Selesnick, "Bivariate shrinkage functions for wavelet-based denoising exploiting interscale dependency," IEEE Transactions on Signal Processing, Vol. 50, No. 11, November 2002.
2. N. Kingsbury, "Image processing with complex wavelets," Philosophical Transactions Of The Royal Society Of London - Series A, Vol. 357, No. 1760, September 2002, pp. 2543-2560.
3. W. T. Freeman and E. H. Adelson, "The design and use of steerable filters," IEEE Transactions on Pattern Analysis and Machine Inteligence, Vol. 13, No. 9, 1991, pp. 891-906.
4. T. Lindeberg, "Principles for automatic scale selection," in Handbook on Computer Vision and Applications. Academic Press, Vol. 2, 1999, pp. 239-274.
5. M. Nibouche, A. Bouridane, D. Crookes and O. Nibouche, "An FPGA-based wavelet transforms coprocessor," IEEE International Conference on Image Processing, Vol. 3, 2001, pp. 194-197.
6. P. Bellows and B. Hutchings, "Designing run-time reconfigurable systems with JHDL," Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, Vol. 28, No. 1-2, May/June, 2001, pp. 29-45
7. T. C. Folson and R. B. Pinter, "Primitive features by steering, quadrature and scale," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 20, No. 11, 1998, pp. 1161-1173
8. G. A. Constantinides, P. Y. K. Cheung and W. Luk "Wordlength Optimization for Linear Digital Signal Processing," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 22, No. 10, October 2003
9. I. Page and W. Luk, "Compiling occam into FPGAs," in Will Moore and Wayne Luk (Eds) 'FPGAs', pp. 271-283, Abingdon EE&CS books, 1991
10. J. Ng and A. A. Bharath, "Steering in Scale Space to Optimally Detect Image Structures," ECCV 04. In Lecture Notes in Computer Science. May 2004
11. http://www.bg.ic.ac.uk/Publications/TIP-00621-2003/AnalysisFilterCoefs.htm
12. http://www.celoxica.com/techlib/files/CEL-W0307171JKX-33.pdf

Programmable Logic Has More Computational Power than Fixed Logic

Gordon Brebner

Xilinx Research Labs, San Jose, U.S.A.
gordon.brebner@xilinx.com

Abstract. In 1964, Elgot and Robinson introduced the Random-Access Stored Program (RASP) machine model "to capture some of the most salient features of the central processing unit of a modern digital computer." After four decades of progress in computer science, this model is now somewhat outdated. Intriguingly though, the 1964 paper presented two theorems showing that programs of 'finitely determined' instructions are properly more powerful if modification of addresses in instructions is permitted during execution than when it is forbidden. In this paper, we celebrate the 40th birthday of these results by using them to prove that allowing programmability of circuits during execution adds extra computational power. To do this, we accord front-line computational status to programmable circuitry, and conduct a theoretical study based on a tradition dating back to Gödel, Turing and Church in the 1930s. In particular, we introduce a new Local Access Stored Circuit (LASC) model of programmable circuitry, intended to form a solid basis for a broad range of future computational research.

1 Introduction

As explained in the abstract, the Random-Access Stored Program (RASP) machine model was introduced by Elgot and Robinson in 1964 [1], their aim being to provide a framework for "the rational discussion of programming languages." It would be fair to say that the RASP model has not maintained any dominant status, either in the area of formal semantics of programming languages, where there has been a move toward abstractions that are higher level than machine code execution, or in the area of computational complexity, where there has been a focus on simpler machine models with more easily quantifiable behavior. A further nail in the RASP coffin has been its specific incorporation of support for program modification at run time, deemed an anathema in the software engineering community.

It is, however, this particular feature of the RASP model that attracted our attention. The current state of research involving field-programmable logic is that the benefits or otherwise of run-time reconfiguration are much debated. Moreover, researchers lack any notable higher-level abstractions of the basic functionality of programmable logic devices. Thus, an investigation at the level of modifiable machine code is very apt at present. To this end, we define the Local Access Stored Circuit (LASC) model, a surprisingly close but non-artificial relative of the RASP model, and thus are able to replicate Elgot and Robinson's proof of a proper difference in computational power, depending on whether run-time programmability is allowed or not.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 404–413, 2004.

2 Background RASP Results

In this section, we set the scene for the two main RASP results of interest. The necessary and sufficient definitions are given in the original mathematics of Elgot and Robinson, accompanied by our layman's interpretations in English. This extensive background review is necessary to bring it to a wider audience, in order to convey a full understanding both of the main results and of the potential for translating the RASP model to a machine model for programmable circuitry.

2.1 Basic RASP Definition

The definition of a random-access stored program machine (RASP) is in Section 3 of [1]. We present the relevant parts of the definition here, and include in-line some earlier definitions from Section 2 of the paper.

A RASP is defined as an ordered sextuple $P = (A, B, b_0, K_0, h^1, h^2)$, where A and B are (usually, countably infinite and, possibly, overlapping or coinciding) sets of abstract objects called *addresses* and *words*, respectively; and b_0 is an element of B, called the *empty word*. K_0 is a subset of K, the set of all functions $k(x)$ which are defined on A and take values in B. Finally, h^1 is a mapping from $\Sigma_0 \times B$ into K_0 and h^2 is a mapping from $\Sigma_0 \times B$ into A, where $\Sigma_0 = K_0 \times A$. These mappings are combined into a mapping $h = (h^1, h^2)$ from $\Sigma_0 \times B$ into Σ_0, and then for any $b \in B$, the mapping h_b, where $h_b(x) = h(x, b)$ for all x [1], is called an (atomic) *instruction*.

Interpreting this, the RASP machine has a random-access memory, with A being the set of addresses of individual storage locations in the memory and B being the set of values that can be stored in a memory location. The contents of the memory *at a specific time* are represented by a function $k(x)$ from A into B, with unused memory addresses being mapped to the 'empty word' value b_0. The computational state of a RASP at a specific time is represented by a pair (k, a) in Σ_0, k being the contents of the memory and a the memory address of the current instruction (i.e., a program counter). The effect of executing a RASP instruction represented by the word $b \in B$ is captured by a mapping h_b, which maps a computational state $\sigma \in \Sigma_0$ to a new state σ'.

Note that this execution semantics does not force the instruction executed to be the one stored at the current instruction address in memory, which is the expected case in practice. To capture this, Section 3 of [1] defines a mapping g from Σ_0 into Σ_0 by $g(k, a) = h((k, a), k(a))$ [2]. Thus, g captures the movement from a current memory state to the next memory state, after executing the current instruction.

2.2 Finitely Determined Instructions

One particular property of instructions is crucial to the main result of interest. As defined in Section 3 of [1], for a given b, an instruction h_b is said to be *finitely determined* if for every a there exists a finite sequence $A_{a,b} = (a_1, \ldots, a_j)$ of elements of A such

[1] Beware that there two different meanings of placeholder variable x in this paragraph, as in [1].

[2] To be precise, Section 3 defines mappings g_1 and g_2 in terms of the mappings h^1 and h^2, and then the definition $g = (g_1, g_2)$ is carried forward from Section 2.

that for any $b_1, \ldots, b_j$ in B there exist elements $a', a_{i_1}, \ldots, a_{i_\ell}$ of A and $b_{i_1}, \ldots, b_{i_\ell}$ of B (ℓ a natural number which also depends on $a, b_1, \ldots, b_j$) for which the following condition is satisfied. If (k, a) is an element of Σ_0 for which $k(a_1) = b_1, \ldots, k(a_j) = b_j$, then $h_b^2(k, a) = a'$, and $h_b^1(k, a) = k'$ where $k'(a_{i_m}) = b_{i_m}$ for $m = 1, \ldots, \ell$ and $k'(x) = k(x)$ for all other x.

Interpreting this, the effect of a finitely determined instruction h_b is that control moves from address a to address a', and the next memory state k' is obtained from the current state k by making a finite number, ℓ, of changes. These depend on the current address a and on the value of k at a finite number, j, of places. The key feature of this definition is to say that the execution of an instruction involves reading from a finite number of *fixed* memory addresses and writing to a finite number of memory addresses. Note that the particular choices of fixed memory addresses read from are allowed to be different at different current execution addresses a.

2.3 RASP Computations

Section 2 of [1] defines a computation of a RASP[3]. An infinite sequence of states $\text{comp}(\sigma_0) = (\sigma_0, \sigma_1, \ldots)$, where each $\sigma_i = (k_i, a_i)$, of a RASP is called a *computation* if $\sigma_{i+1} = g(\sigma_i)$ for $i = 0, 1, \ldots$. Now, if E is a finite subset of A, then define $\text{comp}_E(\sigma_0) = \text{comp}(\sigma_0)$ if for all i, $a_i \notin E$; and define $\text{comp}_E(\sigma_0) = (\sigma_0, \ldots, \sigma_n)$ if $a_n \in E$ and for all $i < n$, $a_i \notin E$. In the latter case, $\text{comp}_E(\sigma_0)$ is said to be *successful* and to *terminate in* σ_n. When E is a singleton set $\{e\}$, define the shorthand notation $\text{comp}_e = \text{comp}_E$.

Interpreting this, a computation of a RASP machine is the set of states that it passes through as instructions are executed. The set of addresses E represents a set of exit points in the computation, introducing the notion of execution continuing until it reaches one of the exit addresses.

2.4 RASP Programs

Section 4 of [1] contains the fairly difficult definition of a RASP program. Let H be the set of all instructions of the RASP. Then a *program* π is the $(m + 2)$-tuple $(p, a_0, e_0, \ldots, e_{m-1})$. p is a mapping from a finite subset $\underline{D}\pi$ (also $\underline{D}p$) of A (the *domain* of π and of p) into $H \cup B$ (here assuming that H and B are disjoint sets), $a_0 \in \underline{D}p$, $p(a_0) \in H$, $i \neq j$ implies $e_i \neq e_j$, and $e_i \in A - \underline{D}p$ for $0 \leq i < m$. Let k *holds* π (and also p) mean: for all $a \in p^{-1}H$, $h_{k(a)} = p(a)$ while for all $a \in p^{-1}B$, $k(a) = p(a)$. If k holds π, $\text{comp}_E(k, a_0)$ is called a computation of π, where $E = \{e_0, \ldots, e_{m-1}\}$. Finally, p must satisfy the property that, if $((k_0, a_0), \ldots, (k_n, a_n))$ is a successful computation of π then $a_i \in p^{-1}H$ for all $i < n$ and, if $((k_0, a_0), (k_1, a_1), \ldots)$ is an unsuccessful computation of π then $a_i \in p^{-1}H$ for all i.

This definition requires interpretation. A stored program consists of instructions and parameters, and is represented by the mapping p. Instructions are stored at the addresses a where $p(a)$ is in H, and parameters are stored at the addresses a where $p(a)$ is in B.

[3] In fact, Section 2 defines this for the IMP, a less general machine model than the RASP, but the definition is inherited in this case.

The starting execution address is a_0, and must contain an instruction of the program. The set of exit addresses is $\{e_0, \ldots, e_{m-1}\}$. These addresses are constrained to be outside the set of addresses containing the program, to ensure certain properties of programs that we need not consider here. The final property guarantees that the sequence of instruction addresses during execution all contain instructions of the program.

A program π is called *fixed* if whenever $(\sigma_1, \ldots, \sigma_n)$ is a successful computation of π, then $k_1, \ldots, k_n$ all agree on $\underline{D}\pi$, where $\sigma_i = (k_i, a_i)$. This captures the situation where the stored program is not changed at all during execution.

2.5 RASP Functions

The preceding definitions of a RASP, its computations and its programs allow progression to the definition of the functions that can be computed by a RASP, which is also in Section 4 of [1]. Given a RASP P and a program $\pi = (p, a, e)$ for P, f, a function from a subset of B^r into B^s, $d_0, \ldots, d_{r-1}, v_0, \ldots, v_{s-1}$, a finite sequence of distinct elements of A, then π *is said to compute* f at *datum locations* $d_0, \ldots, d_{r-1}$ and *value locations* $v_0, \ldots, v_{s-1}$ provided that $d_i \notin \underline{D}p, 0 \leq i < r$, and the following condition holds. If k holds π, $k(d_0) = b_0, \ldots, k(d_{r-1}) = b_{r-1}$ and letting $b = (b_0, \ldots, b_{r-1})$, $\sigma = (k, a)$, $\sigma_i = (k_i, a_i)$, then (i) if f is defined for b and $f(b) = (b'_0, \ldots, b'_{s-1})$ then $\mathrm{comp}_e(\sigma)$ is successful and if it equals $(\sigma, \sigma_1, \ldots, \sigma_n)$ then $k_n(v_i) = b'_i, 0 \leq i < s$; and (ii) if f is not defined for b, then $\mathrm{comp}_e(\sigma)$ is unsuccessful.

Interpreting this, the function f computed by program π has r input arguments, which are stored in the datum memory locations, and s output results, which are stored in the value memory locations. The datum memory locations are disjoint from the program (instruction and parameter) memory locations. Note that there is no explicit input and output in this machine model. The main part of the definition says that the computation starts with the input arguments in memory, and then terminates with the output results in memory if f is defined for these input arguments, and does not terminate otherwise. The alert reader will notice that this formal definition from [1] overloads the variable name b_0 here, but there is no real confusion introduced.

2.6 Sequential Functions

Before proceeding to the theorems of interest, some final definitions are required to introduce the notion of sequential functions, the subject of Section 7 of [1]. A *sequential function over* B is a mapping f from a subset of $B_\infty = \bigcup_{i=1}^{\infty} B^i$ into B_∞, where B^i is the set of all i-tuples of elements of B. The intention behind defining sequential functions was to capture the notion of computing a function that takes as input a finite sequence of arguments that has arbitrary length, as opposed to taking a fixed number of arguments. Similarly, the output can be a finite sequence of arbitrary length. It is then necessary to generalize the definition of the functions computed by a RASP, to accomodate this more general class of functions.

Definition 7.1 of [1] states that a *program* $\pi = (p, a, e)$ of a RASP P *is said to compute a sequential function* over $B' = B - \{b_0\}$ *at* (the infinite sequence of distinct) *datum locations* $d_0, d_1, \ldots$ *and* (the infinite sequence of distinct) *value locations* $v_0, v_1, \ldots$, $v_i \neq d_j$, provided that $d_i \notin \underline{D}p$ for $i \geq 0$ and the following condition holds. If k holds p, $k(d_i) = b_i \in B'$ for $0 \leq i < r$, $k(d_i) = b_0$ for $i \geq r$ and (i) if f is defined for $b = (b_0, \ldots, b_{r-1})$, $f(b) = (b'_0, \ldots, b'_{s-1})$, $k(v_i) = b_0$ for $i \geq s$, then $\mathrm{comp}_e(k, a)$ is successful and if it terminates in state (k_n, a_n), then $a_n = e$ and $k_n(v_i) = b'_i, 0 \leq i < s$ and $k_n(v_i) = b_0$ for $i \geq s$; and (ii) if f is not defined for b, then $\mathrm{comp}_e(k, a)$ is unsuccessful.

Interpreting this, there are now an infinite number of non-overlapping datum locations and value locations. For a particular input sequence of length r, the first r datum locations contain the r input values, and all of the others contain the empty word b_0. Similarly, for a particular output sequence of length s, the first s value locations contain the s output values, and all of the others contain the empty word. Aside from these input and output conventions, the definition of the function computation as presented in Section 2.5 is unchanged. Note that the variable name b_0 was again overloaded in [1], here in a potentially confusing sense.

3 Main RASP Results

In this section, we present the two theorems (7.4 and 7.5) from [1] that establish a proper difference in computational power between RASPs with fixed programs and RASPs allowed to modify their programs during execution. The proofs of the theorems are given here in English interpretion only, as a precursor to examining their relevance to programmable circuitry.

3.1 RASPs with Fixed Programs

Theorem 7.4 of [1] shows that not all (recursive[4]) sequential functions are computable by fixed programs of RASPs with finitely determined instructions. It states that, if

(a) the program $\pi = (p, a, e)$ computes the sequential function f (in the sense of Section 2.6 above) at $d_0, d_1, \ldots$ and $v_0, v_1, \ldots$
(b) the instructions of π are finitely determined (in the sense of Section 2.2 above) and
(c) π is fixed (in the sense of Section 2.4 above)

then there exists r such that if $b, b' \in B'_\infty$, $b = (b_0, \ldots, b_s), s \geq r - 1$, $b' = (b'_0, \ldots, b'_{s'}), s' \geq r - 1$, $b_i = b'_i$ when $0 \leq i < r$, then $f(b) = f(b')$.

The theorem is saying that, for a RASP with a fixed program containing finitely determined instructions, there is some r such that the sequential function computed by the program ignores all but the first r arguments in cases where there are more than r arguments. Thus, a simple example of a sequential function that cannot be computed is the function which takes $(x_1, \ldots, x_n)$, $x_i > 0$ for all i, into $\sum_{i=1}^{n} x_i$ for arbitrary n.

The proof of the theorem hinges on the finitely determined property of the program's instructions. One can therefore define a finite set A' containing all of the memory addresses that can be read by all of the instructions. Then, r can be chosen such that $d_i \notin A'$ for all $i \geq r$, thus ensuring that datum locations $d_r, d_{r+1}, \ldots$ are not taken into account[5]. In summary, this theorem shows the restricted memory addressing available in fixed programs containing instructions with fixed memory addressing.

[4] Defined precisely, albeit implicitly, in Section 7 of [1], but the details are omitted here.

[5] In [1], a smaller value of r is chosen in the proof, but we believe that our choice here is safer.

3.2 RASPs with Modifiable Programs

While Theorem 7.4 of [1] is the key negative result, Theorem 7.5 is the key positive result, which shows that all (partial recursive[6]) sequential functions over the positive integers are computable by a RASP which is not subject to the restriction that its program π is fixed.

The proof uses a particular RASP P_0 defined in Section 4 of [1], and an explicit program for it which is illustrated as a flowchart in Figure 2 of [1]. We will skip all of the programming detail here, both the recursion theory setting and the instruction mechanics, and just point out how program modification is used. The necessity is to ensure that all datum locations can be read and all value locations can be written. Initially, the program packs all of the input arguments into a single argument, using a loop in which an instruction is modified to point at each datum location in turn. Finally, the program unpacks a single result into the separate output arguments, using a loop in which an instruction is modified to point at each value location in turn.

This is of course a highly theoretical piece of programming, relying on an infinite word size for the packing and unpacking of inputs and outputs respectively. However, the practical technique used is in fact just the emulation of an index register for memory access, through treating part of the modified instruction as such a register. As pointed out by Elgot and Robinson [1], if the basic RASP model is augmented so that instruction capabilities are extended to include some type of indirect or indexed memory access, then “it appears rather clear” (their words) that all (partial recursive) sequential functions can be computed by fixed programs, thus rendering Theorem 7.4 redundant.

4 The LASC Programmable Circuit Model

We define the Local Access Stored Circuit (LASC) machine as a computational model of programmable circuitry, following the spirit of the RASP machine. The essence of our model translation is to move from ‘computing in time’ to ‘computing in space’, a familiar concept in the field programmable logic world. We seek to ensure that the essential properties required for the computational differentiation supplied by Theorems 7.4 and 7.5 of [1] are preserved, without making our new model artificial. To stress the correspondence, our LASC definition closely follows the RASP definition, though in the future, we envisage devising an equivalent definition that is more elegant, both structurally and notationally. Our presentation here follows the ordering of Section 2.

4.1 Basic LASC Definition

We retain the RASP random-access memory model intact. Interpreted in terms of programmable circuitry, this memory will hold both programming information (corresponding to circuit configuration memory) and data values (corresponding to registers and other stores). The main change is to add new LASC structure for the parallel circuit style of computing in space, and remove RASP structure for the sequential program style of computing in time.

[6] Defined precisely in Section 7 of [1], but the details are omitted here.

Specifically, we introduce the notion of *nodes*, which represents the connected gates or processing elements in a programmable circuit, each one computing a function from inputs to outputs at each time step. Each node has a disjoint set of local memory addresses associated with it, precisely one of which contains an instruction. The empty word b_0 will now also be used to represent a null instruction, allowing a node to be idle. A node's instruction is allowed to access any memory addresses as inputs, but only local node memory addresses as outputs. Note that this model means that memory addresses not associated with any node will be unmodifiable.

In tandem with adding nodes, we remove the now redundant notion of the current instruction address being part of the machine state.

We define a LASC as an ordered septuple $P = (A, B, b_0, K_0, N, c, h)$, where A, B, b_0 and K_0 are as in the RASP definition. N is a (usually, countably infinite and, possibly, overlapping or coinciding with A and/or B) set of abstract objects called *nodes*, and c is a mapping from N into $(2^A - \emptyset) \times A$. Finally, h is a mapping from $(\Sigma_0 \times N) \times B$ into Σ_0, where $\Sigma_0 = K_0$. For any $b \in B$, the mapping h_b, where $h_b(x) = h(x, b)$ for all x, is called an (atomic) instruction.

In this definition, the mapping c represents the non-empty set of memory addresses associated with a node, together with the distinguished address that holds the instruction for the node. We finalize the capture of the node concept with three defined implications. For any $n \in N$ if $c(n) = (A', a)$ then: (i) $a \in A'$; (ii) for any $k \in K_0$, if $h_b(k, n) = k'$ then $k'(a') = k(a')$ for all $a' \notin A'$; and (iii) for all $n' \neq n$ if $c(n') = (A'', a')$ then $A'' \cap A' = \emptyset$. The first implication ensures that the instruction address is local to the node; the second ensures that the instruction only affects addresses local to the node; and the third ensures that there are no local address overlaps between nodes.

Finally, we can define a mapping g from Σ_0 into Σ_0 that is entirely analogous to the same-named mapping in the RASP model, and captures the movement from a current global memory state to the next global memory state. For any $k \in K_0$, for each $n_i \in N$ let $c(n_i) = (A_i, a_i)$, $k'_i = h((k, n_i), k(a_i))$ if $k(a_i) \neq b_0$ and $k'_i = k$ otherwise, and define $g(k) = k'$ where $k'(a) = k'_i(a)$ if $a \in A_i$ for some i and $k'(a) = k(a)$ for all other a. This definition expresses the fact that all of the changes to the global memory state arise from the collective changes to the local memory state at nodes.

4.2 Finitely Determined Instructions

We can directly carry forward the definition of a finitely determined instruction from Section 2.2, with just a small modification (indeed slight simplification) to shift from the notion of execution of an instruction at the current instruction address to the notion of execution of an instruction at a node.

Thus, for a given b, we say that an instruction h_b is *finitely determined* if for every n there exists a finite sequence $A_{n,b} = (a_1, \ldots, a_j)$ of elements of A such that for any $b_1, \ldots, b_j$ in B there exist elements $a_{i_1}, \ldots, a_{i_\ell}$ of A and $b_{i_1}, \ldots, b_{i_\ell}$ of B (ℓ a natural number which also depends on $a, b_1, \ldots, b_j$) for which the following condition is satisfied. If k is an element of Σ_0 for which $k(a_1) = b_1, \ldots, k(a_j) = b_j$, then $h_b(k, n) = k'$ where $k'(a_{i_m}) = b_{i_m}$ for $m = 1, \ldots, \ell$ and $k'(x) = k(x)$ for all other x.

4.3 LASC Computations

A computation of a LASC, $\mathrm{comp}(\sigma_0)$ can be defined exactly as in Section 2.3. However, we must change the notion of termination. In the RASP model, termination occurs when the current execution address reaches any address e in a set E. In the LASC model, termination will occur when the contents of any memory location with its address e in a set E becomes not equal to b_0. Thus, the set E models a set of 'done' flags.

Let $\mathrm{comp}(\sigma_0) = (k_0, k_1, \ldots)$. If E is a finite subset of A, then define $\mathrm{comp}_E(\sigma_0) = \mathrm{comp}(\sigma_0)$ if for all i, $k_i(e) = b_0$ for all $e \in E$; and define $\mathrm{comp}_E(\sigma_0) = (k_0, \ldots, k_n)$ if $k_n(e) \neq b_0$ for some $e \in E$ and for all $i < n$, $k_i(e) = b_0$ for all $e \in E$.

4.4 LASC Programs

A LASC program differs from a RASP program because, rather than one instruction being executed at each time step, all non-null instructions at all nodes are executed at each time step. Also, the structure of the LASC model incorporates the addresses of the instructions through the mapping c. Given this, the definition of a program can be defined analogously to (and slightly more simply than) the RASP definition in Section 2.4.

Let H be the set of all instructions of the LASC. Then a *program* π is the $(m+1)$-tuple $(p, e_0, \ldots, e_{m-1})$. p is a mapping from a finite subset $\underline{D}\pi$ (also $\underline{D}p$) of A (the *domain* of π and of p) into $H \cup B$ (here assuming that H and B are disjoint sets), $p^{-1}H \subseteq \cup_{n \in N}\{a|c(n) = (A', a)\}$, and $i \neq j$ implies $e_i \neq e_j$. Let k *holds* π (and also p) mean: for all $a \in p^{-1}H$, $h_{k(a)} = p(a)$ while for all $a \in p^{-1}B$, $k(a) = p(a)$, and $k(a) = b_0$ for all a such that $c(n) = (A', a)$ for some n and $a \notin p^{-1}H$. If k holds π, $\mathrm{comp}_E(k)$ is called a computation of π, where $E = \{e_0, \ldots, e_{m-1}\}$.

A program π is called *fixed* if whenever $(k_1, \ldots, k_n)$ is a successful computation of π, then $k_1, \ldots, k_n$ all agree on $\underline{D}\pi$.

4.5 LASC Functions

We can define the functions computed by a LASC almost directly using the definition for a RASP from Section 2.5, the only change being to remove the current execution address a wherever it appears in the definition.

Given a LASC P and a program $\pi = (p, e)$ for P, f, a function from a subset of B^r into B^s, $d_0, \ldots, d_{r-1}, v_0, \ldots, v_{s-1}$, a finite sequence of distinct elements of A, then π *is said to compute* f at *datum locations* $d_0, \ldots, d_{r-1}$ and *value locations* $v_0, \ldots, v_{s-1}$ provided that $d_i \notin \underline{D}p, 0 \leq i < r$, and the following condition holds. If k holds π, $k(d_0) = b_0, \ldots, k(d_{r-1}) = b_{r-1}$ and letting $b = (b_0, \ldots, b_{r-1})$, then (i) if f is defined for b and $f(b) = (b'_0, \ldots, b'_{s-1})$ then $\mathrm{comp}_e(k)$ is successful and if it equals $(k, k_1, \ldots, k_n)$ then $k_n(v_i) = b'_i, 0 \leq i < s$; and (ii) if f is not defined for b, then $\mathrm{comp}_e(k)$ is unsuccessful.

4.6 Sequential Functions

We can define the sequential functions computed by a LASC almost directly using the definition for a RASP from Section 2.6, the only change again being to remove the current execution address a wherever it appears in the definition.

A *program* $\pi = (p, e)$ of a LASC P is said to *compute a sequential function* over $B' = B - \{b_0\}$ *at* (the infinite sequence of distinct) *datum locations* $d_0, d_1, \ldots$ *and* (the infinite sequence of distinct) *value locations* $v_0, v_1, \ldots$, $v_i \neq d_j$, provided that $d_i \notin \underline{Dp}$ for $i \geq 0$ and the following condition holds. If k holds p, $k(d_i) = b_i \in B'$ for $0 \leq i < r$, $k(d_i) = b_0$ for $i \geq r$ and (i) if f is defined for $b = (b_0, \ldots, b_{r-1})$, $f(b) = (b'_0, \ldots, b'_{s-1})$, $k(v_i) = b_0$ for $i \geq s$, then $\text{comp}_e(k)$ is successful and if it terminates in state k_n, then $k_n(v_i) = b'_i, 0 \leq i < s$ and $k_n(v_i) = b_0$ for $i \geq s$; and (ii) if f is not defined for b, then $\text{comp}_e(k)$ is unsuccessful.

5 Main LASC Results

We now present LASC versions of the two main RASP theorems, plus an extra new theorem, in order to establish a proper difference in computational power between LASCs with fixed programs and LASCs allowed to modify their programs during execution.

5.1 LASCs with Fixed Programs

Theorem 1. *Not all (recursive) sequential functions are computable by fixed programs of LASCs with finitely determined instructions. Specifically, if*

(a) the program $\pi = (p, e)$ computes the sequential function f at $d_0, d_1, \ldots$ and $v_0, v_1, \ldots$
(b) the instructions of π are finitely determined and
(c) π is fixed

then there exists r such that if $b, b' \in B'_\infty$, $b = (b_0, \ldots, b_s)$, $s \geq r - 1$, $b' = (b'_0, \ldots, b'_{s'})$, $s' \geq r - 1$, $b_i = b'_i$ when $0 \leq i < r$, then $f(b) = f(b')$.

The proof of this theorem follows exactly the proof of Theorem 7.4 in [1], as sketched in Section 3.1, both in terms of the basic idea and the actual mathematics. All that is required is simplification by crossing out all references to the current execution address component of the state in the RASP model. This very direct mapping of the proof follows from the care we have taken to define the LASC model and its surrounding computational concepts entirely analogously to the RASP model.

5.2 LASCs with Modifiable Programs

Theorem 2. *All (partial recursive) sequential functions over the positive integers are computable by a LASC which is not subject to the restriction that its program π is fixed.*

The proof of Theorem 7.5 in [1] involves demonstrating a particular program for a particular RASP. The proof of this LASC theorem can follow in at least two ways: because there is an equivalent program for a particular LASC, or because any RASP machine P can be simulated by a LASC machine P'.

Space limitations preclude a presentation of the technical detail needed to justify either of these claims. Therefore, to confirm concretely here the extra power derived from the LASC program not being fixed, we end by presenting an existence theorem for the integer summation function, which is a sequential function not computable by a LASC with a fixed program, for the RASP-based reason given in Section 3.1.

Theorem 3. *There is a non-fixed LASC program which computes the sequential function that takes* $(x_1, \ldots, x_n)$*,* $x_i > 0$ *for all* i*, into* $\sum_{i=1}^{n} x_i$ *for arbitrary* n*.*

With A, B the non-negative integers and $b_0 = 0$, a suitable LASC has one node with three local addresses $\{0, 1, 2\}$ and instruction at 0. For datum locations $3, 4, \ldots$ and value location 2, a suitable program has $e = 1$ and: (i) a (finitely determined) instruction at 0 that is $([3] = 0)?([1] \leftarrow 1) : ([2] \leftarrow [2] + [3]; [0] \leftarrow \phi([0]))$, where [] is memory access, $()?() : ()$ is a C-style conditional, $\leftarrow$ is assignment and ϕ increments both of the "[3]" addresses in this instruction; and (ii) parameters at 1 and 2 equal to 0.

6 Discussion

An immediate practical qualm about the LASC model might concern the potentially infinite features. However, these have always been present in such models — since the dawn of the Turing machine in 1936 — because if everything is made finite, all models collapse to be finite state machines and lose their computational subtlety. We must regard infinite notions as capturing the practical idea of *arbitrarily large*. Note also that here we focused on finitely determined instructions, thus ruling out one fairly dubious infinite possibility that imbues instructions with essentially unlimited power.

Regarding the practicality of our specific LASC model, we note four points. First, the one-instruction node seems a good model for both fine-grain (logic gate) and coarse-grain (processing element) programmable circuitry. Second, allowing arbitrary inputs to nodes represents a 'programmable netlist' scheme, as a generalization of, say, a more restricted 'programmable array neighbor' scheme. We feel that this is not unreasonable. Third, there is no explicit notion of input or output (as normal for such models), which may seem unnatural for circuits. Note that fixed programs do not become all-powerful if input data are allowed to arrive successively at the same address (e.g. consider the function $\sum_{i=1}^{n} x_i x_{i+n}$ that forces arbitrary buffering of inputs in memory). Fourth, our least practical feature is the model of programmability during execution, both because it is allowed at each time step and because of the programmable netlist scheme.

As mentioned in Section 3.2, for the RASP model, the separation in computational power collapses if the practical notion of indirect or indexed addressing is added. Some good news for LASC is that, due to the restrictive support for run-time reconfigurability in practical programmable circuitry devices, there is no direct practical equivalent of an index register. The nearest equivalent would be adding a separate memory with address register beside the programmable circuitry, in an extended LASC model. Such a modification would then remove the separation in computational power.

In terms of our main result, we did not find it a surprise that the program modification during execution concerns the interconnection of the circuitry rather than the node functionality in the circuitry. As a general opinion, we feel that the true power and interest of programmable circuitry lies in the interconnection.

References

1. C. Elgot and A. Robinson. Random-Access Stored-Program Machines, an Approach to Programming Languages. *Journal of the ACM* **11**:4, Oct 1964, pp.365–399.

JHDLBits: The Merging of Two Worlds

Alexandra Poetter[1], Jesse Hunter[1], Cameron Patterson[1], Peter Athanas[1],
Brent Nelson[2], and Neil Steiner[1]

[1] Configurable Computing Lab, Virginia Tech, Blacksburg, VA 24061, USA
{apoetter, jehunte3, cdp, athanas, nsteiner}@vt.edu,
[2] Configurable Computing Lab, Brigham Young University, Provo, UT 84602, USA
nelson@ee.byu.edu

Abstract. This paper introduces JHDLBits, the integration of two prominent FPGA design tools: JHDL and JBits. JHDLBits offers the low-level access and control provided by JBits with the high-level structural circuit design of JHDL. Furthermore, the JHDLBits flow provides greater control of resource manipulation, placement, and routing, and gives researchers a "sandbox" to explore advanced interactions with FPGA bitstreams. This paper presents the overall architecture of the open-source JHDLBits project. Details are provided on how the core components - JHDL, JBits3 for Virtex-II, and the ADB connectivity database - are linked together to provide a cohesive design environment.

1 Introduction

Investigators involved in FPGA-related research often require a testbed for exploring and evaluating new tools, new algorithms, and new ways to interact with FPGA bitstreams. Historically, this has often proven to be a difficult task. Exceptional exploratory environments have been created, such as VPR by Betz and Rose [1] that provide realistic models of FPGAs. With the infrastructure created by VPR, researchers could determine the effects of, for example, placement enhancements on wire length. The relevance of such exploration was sometimes diminished since the results from the experimental environment could not be definitively confirmed on a real FPGA since FPGA vendors tend to be secretive on the low-level architectural details of their products. Many languages, IDEs, and compilers have emerged in recent years that offer interesting environments for creating FPGA bitstreams, but most rely on the FPGA vendors' implementation flows to map, place, and route the final design.

The authors of JBits [2] addressed this problem to a certain degree by providing an API to the FPGA bitstream. As a result, researchers were empowered with the ability to manipulate FPGA resources at the lowest level. With JBits, researchers could develop, say, their own FPGA router, and test it on real devices. JBits also enabled researchers to interact with FPGAs in ways prohibited by the vendor's prescribed implementation tool flow.

While JBits was in many ways an "enabling technology" for exploring nontraditional uses of FPGAs, the abstraction presented to the developer was at a

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 414–423, 2004.

fairly low level (at least in early versions of JBits), and sometimes required detailed architectural knowledge. In contrast to this, tools such as Brigham Young's JHDL [3] use a much higher level of abstraction, often making it a more conducive environment for large application development.

JHDLBits is an open-source endeavor striving to merge the salient features of these two prominent FPGA research environments. JHDLBits blends the low-level control of JBits with the high-level design abstractions of JHDL. Users can take advantage of the run-time and partial reconfiguration features of JBits without having to work entirely at the bitstream level. Furthermore, full control over placement and routing is still possible. The JHDLBits project consists of a collection of tightly integrated components that together provide an end-to-end pathway for creating, manipulating, and debugging FPGA bitstreams. More importantly, because most of the components of this project are open-source, researchers investigating architecture-specific placers/planners/routers/compilers have the advantage of replacing the stock JHDLBits components at will.

JHDLBits integrates JBits low-level bit manipulation capabilities into the JHDL integrated development environment. The extensible JHDL netlister has been modified to produce bitstreams directly. The JHDL debugger communicates to a target FPGA through JBits XHWIF [4]. JHDL is naturally extended so that instead of generating an EDIF file, which is run through the traditional vendor tools to generate a bitstream, the primitive and net information is extracted into a database that is suitable for JBits to process. A bitstream is created by elaborating, placing, and routing the corresponding JBits primitives all within the JHDLBits environment. Figure 1 illustrates the components that collectively make up the JHDLBits project.

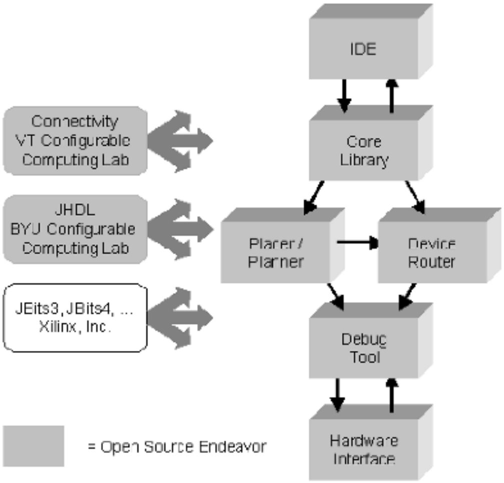

IDE
Core Library
Connectivity VT Configurable Computing Lab
JHDL BYU Configurable Computing Lab
Placer / Planner
Device Router
JBits3, JBits4, ... Xilinx, Inc.
Debug Tool
Hardware Interface
= Open Source Endeavor

Fig. 1. Relationships of the open source constituents of JHDLBits

This paper presents the overall architecture of the JHDLBits project. Details are provided on how the core components - JHDL, JBits3 for Virtex-II, and the

ADB connectivity database [5] - are linked together to provide a cohesive design environment. Sections 2, 3, and 4 provide background information on JHDL, JBits, and ADB, respectively. Section 5 introduces the FPGA device simulator that can be used to verify the JHDLBits design. The architecture of JHDLBits is discussed in Section 6. A JHDLBits design implementation and results are included in Section 7 and Appendix A.

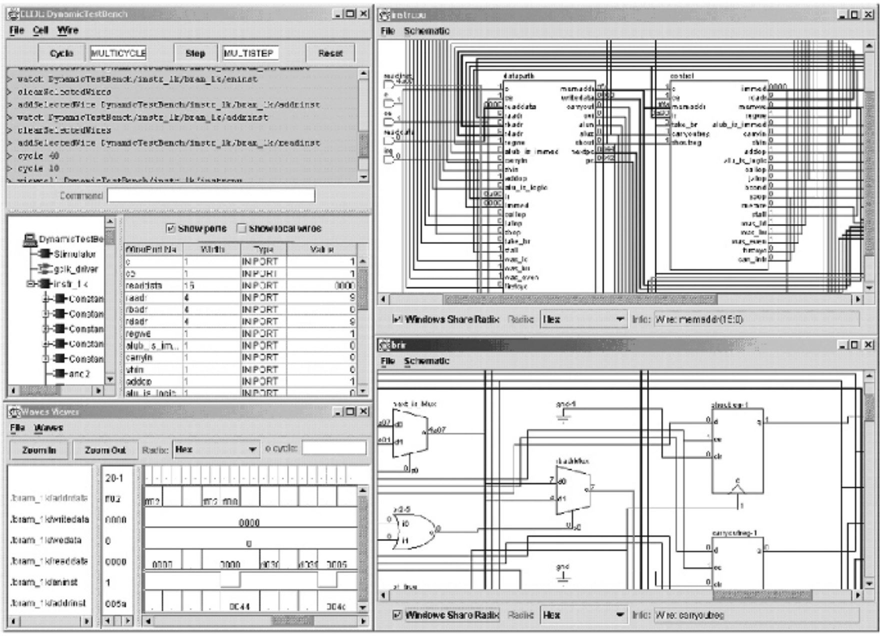
2 JHDL

JHDL is a Java-based design language for FPGAs developed at Brigham Young University. Java was selected because it is object-oriented, easy to use, has built-in documentation capabilities, is portable, and has a rich set of GUI APIs that are integral to the language. The primary distinction of JHDL is the creation of a single integrated API that allows the designer to express circuit organizations that dynamically change over time [3].

JHDL [10] is a structural HDL - that is, circuits are described by structurally instantiating lower-level building blocks. In JHDL, two basic Java classes form the basis for all circuits: `Logic` and `Wire`. Designers create a new logic cell in their design by extending `Logic` (creating a new class), defining a cell interface (essentially a VHDL entity declaration), and defining the architecture of the cell (essentially a VHDL architecture body). Wires support an API for creation and manipulation: users can create single- or multi-bit wires, and concatenate or extract wires from existing wires.

The body of a design instantiates predefined circuit modules selected from the JHDL design libraries. The JHDL design libraries were created in a layered fashion and range from a library of Xilinx primitives such as gates and flip flops, to parameterized logic generator methods, to technology-independent and technology-specific module generators for the creation of larger elements (such as counters, memories, floating point arithmetic modules, CORDIC units, FFTs). JHDL circuit descriptions are based on Java; thus the full range of Java language features are available to construct the circuit. These features include file I/O, recursion, control flow constructs (for, while, do loops), functions, user-defined types (objects), and reflection.

A key feature of JHDL is its ability to operate in either simulation or hardware mode. When in simulation mode, the values of all elements in the circuit data structure are computed by the JHDL simulator. However, JHDL also supports a hardware-in-the-loop mode of execution. In this case, starting execution of the circuit downloads a bitstream to the FPGA platform, and stepping the clock triggers a clock step on the actual hardware. Between clock steps, bitstream readback extracts the hardware state, which is back annotated into the circuit data structure for graphical display. As a result, the same GUI interface and tools can be used for debugging in either in simulation mode or hardware execution mode. This simplifies the transition from simulation to hardware debug, and streamlines the development process. Figure 2 presents a screen capture of the JHDL GUI.



```
> watch DynamicTestBench/instr_1k/bram_1k/eninst
> clearSelectedWires
> addSelectedWire DynamicTestBench/instr_1k/bram_1k/addrinst
> watch DynamicTestBench/instr_1k/bram_1k/addrinst
> clearSelectedWires
> addSelectedWire DynamicTestBench/instr_1k/bram_1k/readinst
> cycle 40
> cycle 10
```

Wire/Port Na	Width	Type	Value
c	1	INPORT	1
c5	1	INPORT	1
readdata	16	INPORT	0000
raadr	4	INPORT	9
rbadr	4	INPORT	0
rdadr	4	INPORT	9
regwe	1	INPORT	1
alub_is_im...	1	INPORT	0
carryin	1	INPORT	0
shin	1	INPORT	0
addop	1	INPORT	1
alu_is_logic	1	INPORT	0

Fig. 2. JHDL graphical user interface screen shot

3 JBits Background and Enhancements

JBits is a collection of Java classes that build upon an API that provides access to every configurable resource in a Xilinx FPGA. The JBits3 SDK [9] is the latest release from Xilinx that provides support for the Virtex-II FPGA family. This API permits device resources to be programmed and probed individually at run-time, even with the FPGA active in a working system. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. Through this mechanism, JBits supports the run-time reconfiguration of Xilinx FPGAs.

For JHDL to function with JBits, several extensions and enhancements were needed. A bridging object, called the `Bitstream` class, was created to allow abstract access to both the JBits and router objects, enabling the creation of primitives without dependencies on architecture-specific classes. The `Net` class was extended to allow the connection of primitives by maintaining a list of the source and sink pins that form a net.

To bridge the gap between JHDL and JBits, a library of primitive cores was created that map the JHDL primitives directly to the FPGA fabric. For completeness, each JHDL primitive requires a corresponding JBits primitive. The primitives created for JHDLBits have one constructor specifically designed for the JHDL-to-JBits flow, with the other constructors catering to a JBits-only design flow. A primitive requires two pieces of information to be implemented:

a list of input and output nets, and placement information. The primitive uses the placement location to assign source and sink pins to the associated nets, and then accesses the JBits object from the `Bitstream` class to configure the internal logic and resources using JBits calls.

4 ADB Background

ADB is an alternate wire database for Xilinx Virtex, Virtex-E, Virtex-II, and Virtex-II Pro FPGAs [6]. Its purpose is to provide exhaustive coverage of device wiring and connectivity, while remaining fast, compact, and compatible with JBits. The exhaustive coverage is derived from the same proprietary data that the Xilinx mainstream tools use, and presents an improvement over the coverage available in past and present JBits wire databases. In addition, the database files used by ADB are more than an order of magnitude smaller than their counterparts in the mainstream tools.

Although ADB is primarily a wire database, it also includes support services for routing, unrouting, and tracing. These interfaces are publicly exposed, so they can be extended or replaced as necessary by the user. An ADB-based router is included with JHDLBits, and is based on a robust search algorithm enhanced with heuristics suited to each FPGA family. The unrouter provides services not normally found in static tool flows, by allowing the user to disconnect existing nets, in order to reconnect other nets at runtime. This may be especially useful in embedded systems that have to dynamically reconfigure themselves [7]. The tracer may also be used to support reconfiguration, by inferring connectivity information from a configuration bitstream, in order to safely modify designs without causing contention.

Because JBits 3.0 ships without a router, ADB currently provides the only routing option available to JBits users. However, ADB may also be of interest to researchers wanting to evaluate routing algorithms with wiring data from commercial FPGAs. In the context of JHDLBits, the ADB router simply implements `RouterInterface` as defined by JBits, which makes it reasonable to plug ADB into the project without concern for its inner workings.

5 FPGA Device Simulator

Included in the JHDLBits project is the design of a Virtex-II device simulator (VTsim) [11]. Initially, a functional simulator has been created using a globally synchronous event-driven model with CLB granularity. The simulator can be used as part of the JHDLBits design flow, but can also function independently because it only requires a bitstream as input.

The simulator first invokes the ADB tracer on the bitstream to be simulated. After the tracer builds a database of all internal connections, the simulator constructs the netlist from this information. An optional JHDLBits simulation file can be used to provide the simulator with specific information such as net names and placement information. A simulation cycle begins with the evaluation of all clocked elements within the CLBs, followed by the evaluation of all non-clocked elements. Upon completion of a simulation cycle, the simulator can either write the updated results back to a bitstream, or allow GUI access to the modified information. It should be noted that this simulator can simulate any Virtex-II bitstream, regardless of the flow used to create the bitstream.

6 JHDLBits Design Flow

The traditional JHDL design flow produces an EDIF file, which is used by the Xilinx implementation tools to generate a bitstream. The JHDLBits flow instead relies upon JBits extensions to generate a bitstream. A high-level design is created and simulated with JHDL libraries and graphical debugging tools. The JHDL primitives, instances, and nets are extracted and mapped to equivalent JBits primitives and nets. JBits primitives are placed using either placement directives or an extendable placement algorithm, and then routed using ADB. A bitstream is generated, along with net and simulator files. Figure 3 shows the steps involved in the JHDLBits design flow.

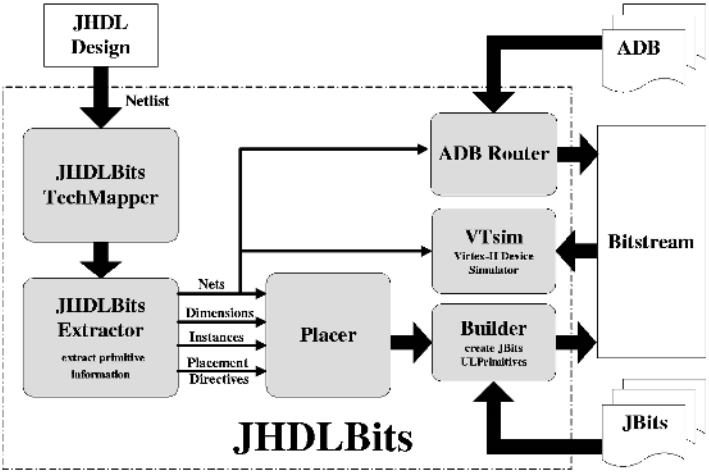


Fig. 3. JHDLBits design flow

The first step in the JHDLBits design flow is the `JHDLBitsTechMapper`. The `JHDLBitsTechMapper` extends the JHDL `Virtex2TechMapper` by overriding the netlist method, which invokes the `JHDLBitsExtractor` instead of the `EDIFNetlister`. Overriding device specific helper methods, such as add and subtract, allows for more optimized primitives.

The `JHDLBitsExtractor` uses JHDL libraries such as the `Cell` and `Net` classes to extract primitive, instance and net information, which are stored in `HashMap` collections. The primitive `HashMap` contains the ports and directions, and the instance HashMap contains the associated primitive, nets, and dimensions. The net HashMap contains ports of the net and a JBits net of the same name. The instance HashMap is then iterated in order to specify placement of the instances.

The default placement algorithm stores the assigned CLB, slice, LUT, and LE (logic element) to the instance HashMap. The assigned coordinate information is also maintained for an instance that is ExternallyUpdateable, which refers to a JHDL cell (flip-flops, memories, etc.) that changes value and can be read back using the JHDL simulator in hardware execution mode. A JBits primitive is now created for each instance. Once all of the JBits primitives have been created, a router utilizing the connectivity information in ADB routes all nets connected to the JBits primitives. After the design has been successfully routed, JBits is used to generate a bitstream.

JHDLBits and VTsim tie into the JHDL simulator through the JHDL HardwareInterface. VTsim is given the bitstream generated, along with coordinate information for the cells that are ExternallyUpdateable, and returns values for these cells after each clock cycle. The updated values are displayed in the JHDL GUI interface.

Partial reconfiguration will be supported in JHDLBits through an incremental design flow. In an incremental design methodology, a user has completed part of a design, and would like to "lock it down" by no longer having to repeatedly place and route this part of the design. This would be accomplished by including a reserved area with defined inputs and outputs into the current JHDL design. This design is run through the JHDLBitsExtractor, but the placer and router are constrained to avoid the reserved area. At a later time, the user can then include additional logic into the reserved area, and once again run through the JHDLBitsExtractor in order to generate a bitstream.

7 JHDLBits Implementation and Results

JHDLBits has been used to generate bitstreams for simple JHDL designs such as the following partial NBitAdder code:

```
// Simple model of a simple parameterized n-bit adder:
Wire carries = wire(width);  // Intermediate 'carry' wires
for (int i=0; i < width; i++) {
      if (i==0)
          new FullAdder(this,a.gw(i),b.gw(i),gnd(),
              sum.gw(i), carries.gw(0));
      else
          new FullAdder(this, a.gw(i), b.gw(i), carries.gw(i-1),
              sum.gw(i), carries.gw(i));
}
```

Once the above design is compiled, the user needs to "wire" the NBitAdder design into the top-level system as shown below:

```
 1  public class tb_nbitadder extends Logic implements TestBench {
 2          static JBitsTechMapper tm;
 3          static Cell mynadd;
 4          static HWSystem hw;
 5
 6          public static void main(String args[]) {
 7              hw = new HWSystem();        // create a new system
 8              tb_nbitadder tb = new tb_nbitadder(hw);
 9              tm.netlist(mynadd, true, "testnbitadd.txt");
10          }
11          public tb_nbitadder(Node parent) {
12              super(parent);
13              tm = new JBitsTechMapper();
14              Logic.setDefaultTechMapper(tm);
15              Wire a = wire(4, "a");
16              Wire b = wire(4, "b");
17              Wire sum = wire(4, "sum");
18              mynadd = new NBitAdder(this, a, b, sum); // connect
19          }                                  // NBitAdder design
20  }
```

Lines 2 and 13 indicate how JHDLBits is invoked from a JHDL design. As previously mentioned, the `JHDLBitsBitsTechMapper` netlist method calls the `JHDLBitsExtractor` instead of the `EDIFNetlister`. The `JHDLBitsExtractor` obtains all the information required from the JHDL design to create, place and route JBits primitives. A bitstream is then generated. The output generated for this sample run is included in Appendix A. As shown at the start of the output file, the ADB database is imported for the Virtex-II XC2V40 device. Next, the `JHDLBitsExtractor` begins traversing the hierarchical logic blocks at the top-level `NBitAdder` cell, and then through all successive children. When a new `gnd` or `and2` primitive is encountered, the ports and directions are stored. The specific instance information of the `gnd` and `and2` primitives is also retained. This is done recursively for all primitives and instances. Next, the `b<1>` net, then all remaining nets are extracted and converted to JBits nets. The placer assigns CLB, slice, LUT, and LE (logic elements) locations for each instance. After placement, JBits primitives are created, such as the `and2`, and associated nets are assigned source and sink pins. An example is the `FullAdder-3/and_out-2` net, which is assigned to pin `CLB.X1[0][2]`. This pin refers to the X output in CLB row zero, column two, slice one. Once all JBits primitives are created and pins assigned, ADB is invoked to route the nets. Finally, a bitstream is generated.

A few pertinent statistics are placements per second, and routes per second. The average placement rate is 512 primitives in 11 milliseconds, or 46,545 primitives per second. Please note that this design was run using a simple placer. A 100% utilization stress test on a Pentium 3.2 GHz processor running Sun Java HotSpot(tm) Client VM (build 1.4.2_04-b05, mixed mode) yielded the ADB routing results in Table 1. The stress test consisted of a design that filled up every LE in the FPGA device with logic gates, which were then routed in a random fashion.

8 Conclusions and Future Work

A major goal of the JHDLBits project has been to retain the properties and philosophies of JBits. Following this principle, it is essential to provide the ability to reconfigure the device through the JHDLBits design flow. This area is

Table 1. ADB routing performance for a 100% utilization stress test

Device	Route Time	Total Nets	Total Wires	Nets/sec	Wires/sec
XC2V40	5,462 ms	511	37,213	94	6,813
XC2V500	80,865 ms	6,143	495,774	76	6,130
XC2V3000	419,080 ms	28,671	2,479,424	68	5,916

important and will be addressed in the near future. Another goal is to allow for partitioning of the design, so that the user can run part of the JHDL design through the mainstream tools and part through JBits via the JHDLBits extractor. The developers would like to investigate using JHDLBits in an embedded system, possibly using a small subset of JHDL and ADB rewritten in a language other than Java to reduce memory usage.

In the current version of JHDLBits, placement is performed in a greedy suboptimal manner. Placement directives inherent in JHDL along with cell properties are currently being investigated and may possibly need refinement. Controlled placement will allow the user to identify a specific location in the device, permitting easier debugging as well as optimizing performance. The placer could potentially benefit from the use of previously designed placement algorithms, such as a core-based incremental placement algorithm [8].

Additional work is also needed to complete the JBits primitive library. The net interaction with the router must be improved to decrease memory usage and reduce routing time. The device simulator will continue to be developed in order to become more robust. It has been successfully applied to small static designs. The next step in the simulator design is to decrease memory usage, improve execution time, and provide a more flexible interface between JHDL and the router. The JHDLBits source code along with additional information can be found at http://sourceforge.net/project/jhdlbits.

Acknowledgements. This project is supported by a grant from Xilinx, Inc.; JBits (Xilinx and Virginia Tech) and JHDL (Brigham Young University) were both funded under the DARPA ACS program.

References

1. V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *International Workshop on Field Programmable Logic and Applications*, pages 213–222, September 1997.
2. S. A. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware," *Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering*, Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, volume 3526, pages 97–102, Bellingham, WA, November 1998.
3. B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A CAD Suite for High-Performance FPGA Design," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, Napa, CA, April 1999.

4. D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems," *Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering*, Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, volume 3526, pages 239–246, Bellingham, WA, November 1998.
5. N. Steiner, "A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs," Master's thesis, Virginia Tech, August 2002.
6. N. Steiner, "An Alternate Wire Database for Xilinx FPGAs," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, April 2004.
7. R. Fong, S. Harper, and P. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, pages 117–123, San Diego, CA, June 2003.
8. J. Ma, "Incremental Design Techniques with Non-Preemptive Refinement for Million-Gate FPGAs," Doctoral dissertation, Virginia Tech, January 2003.
9. The JBits SDK, Xilinx, Inc., http://www.xilinx.com/products/jbits/
10. The JHDL Home Page, Brigham Young University, http://www.jhdl.org/
11. J. Hunter, "A Device-Level FPGA Simulator," Master's thesis, Virginia Tech, June 2004.

Appendix A: Sample JHDLBits Run (Partial Listing)

```
Opening ADB database ...edu\vt\ADB\Virtex2\XC2V40.db...
In JBitsExtractor expand function.
   cell NBitAdder
In JBitsExtractor expand function.
   cell gnd
In JBits extractPrimitive function.
          (port GROUND  (direction 1))
In JBits extractInstance function.
   Instance Name gnd-1
   cellRef gnd
In JBitsExtractor expand function.
   cell and2
In JBits extractPrimitive function.
          (port i0  (direction 0))
          (port i1  (direction 0))
          (port o  (direction 1))
In JBits extractInstance function.
   Instance Name FullAdder/andX_g/andX/and2
   cellRef and2

Net name b<1>
portName: i1   instanceName: FullAdder-1/xor3
portName: i1   instanceName: FullAdder-1/andX_g-2/andX/and2
portName: i0   instanceName: FullAdder-1/andX_g/andX/and2

Placing in progress...
Instance 18  FullAdder-3/andX_g-2/andX/and2
Port 0: Xwire<2>
Port 1: b<3>
Port 2: FullAdder-3/and_out-2
Creating JBits and2 Object...
Adding SINK on Net(Xwire<2>): CLB.F1_B1[0][2]
Adding SINK on Net(b<3>): CLB.F2_B1[0][2]
Adding SOURCE on Net(FullAdder-3/and_out-2): CLB.X1[0][2]

Calling router...
Net name  = FullAdder-3/and_out-2
Sourcepin = CLB.Y2[0][4]
Sinkpins:
CLB.G1_B1[0][2]
Routing this connection...

Evaluated 71 groups in 8 tiles.
Bitstream Generated.
```

A System Level Resource Estimation Tool for FPGAs

Changchun Shi[1], James Hwang[2], Scott McMillan[2], Ann Root[2], and Vinay Singh[2]

[1] Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
ccshi@eecs.berkeley.edu
[2] Xilinx Inc., 2100 Logic Drive, San Jose, CA, USA
{Jim.Hwang, Scott.McMillan, Ann.Root, Vinay.Singh}@xilinx.com

Abstract. High level modeling tools make it possible to synthesize a high performance FPGA design directly from a Simulink model. Accurate estimates of the FPGA resources required provides the system designer important feedback on area and cost, which is valuable even during early design iterations. Previous approaches to hardware resource estimation suffer a combination of inaccuracy, slowness, and/or high complexity, which limits their practicality at the algorithm definition stage. We address these restrictions with a fast resource estimation tool incorporated in the Xilinx System Generator. Implemented using MATLAB code, the estimator run time is proportional to the Simulink compilation time, and typically takes from seconds to minutes depending upon the size of the Simulink model. Estimates are conservative, and accurate to within 10% of the post-mapping implementation report. In this paper, we explain how block resource information is characterized in a MATLAB function. This characterization also estimates logic that will be trimmed during synthesis and mapping. Finally, we describe how these estimation functions are integrated within Simulink in a user-friendly and automated infrastructure. This approach has been incorporated in System Generator since version 3.1.

1 Introduction

Field-programmable gate arrays (FPGAs) have become increasingly important components of digital signal processing (DSP) systems such as digital communications and multimedia applications. FPGA resources most often used for DSP functions include look-up tables (LUTs), flip-flops (FFs), block memories (BRAM), tristate buffers (TBUFs), input/output blocks (IOBs), and dedicated hardware multipliers (18x18 bit multipliers are commonly available) [9]. A top-down design methodology has been recognized to dramatically speed-up the design process without substantially compromising the performance of the hardware implementation [1-4]. In fact, with high level hardware-cost estimation tools, top-down design flows open the possibility of global optimization at the system level, which often leads to more hardware-efficient designs (see, e.g. [5,6]). FPGA resource usage is an important measure of hardware-cost (others include critical path delay and active-power consumption). Minimizing resource usage is particularly important when the goal is to find the best behavioral system performance (such as signal-to-noise ratio) with a device cost constraint, or when the goal is to find the fewest resources meeting a per-

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 424–433, 2004.

formance specification. As another example, it is sometimes necessary to partition a large system to multiple FPGA chips. Resource estimates for sub-systems assists in this partitioning. Optimization processes such as automatic precision adjustment [5-6] further require numerous iterations of resource estimations. This motivates a fast resource estimation tool at high level.

Among available CAD tools for FPGA design [2-4], System Generator for DSP [1] [4,7] is a successful example for modeling and designing Xilinx FPGA-based signal processing systems in Simulink and Matlab[2] [8]. Section 2 begins with a brief description of the System Generator environment. We then describe existing or possible resource estimation methodologies, followed by our proposed method. Our method requires only a Simulink compilation stage to compute an estimate (since data types are propagated during compilation, this step is necessary). It differs from prior work in requiring complete understanding of how the underlying IP-cores are designed, and by its ability to estimate logic that will be trimmed by synthesis and mapping tools. We then describe how the methodology can be integrated with a Simulink GUI and Matlab command line to form a user-friendly infrastructure, which facilitates estimation for selected parts of a system. Section 3 validates the fully implemented resource estimator by studying the estimation results of a couple DSP designs. A few possible future developments are summarized in Section 4.

2 Resource Estimation in System Generator

Our resource estimation is implemented in the System Generator design environment that is described in Section 2.1. Though the methodology is portable to other platforms, the architectural description of a DSP system, as System Generator does naturally, is indeed necessary for accurate estimation.

2.1 System Generator Design Environment

At simulation level, System Generator for DSP maintains an abstraction level very much in keeping with the traditional Simulink blocksets, but at the same time automatically translates designs into hardware implementation [4,7]. The system model and hardware implementation are bit-true and cycle-true. Besides some synthesized blocks, the implementation is also made efficient through the instantiation of high-speed and area-efficient intellectual property (IP) cores that provide a range of functionality from arithmetic operations to complex DSP functions. In System Generator, the capabilities of IP cores have been extended transparently and automatically to fit gracefully into a system level framework. For example, although the underlying IP cores operate on unsigned integers, System Generator provides logic wrappers that allow signed and unsigned fixed-point numbers to be used, including saturation arithmetic and rounding. While providing functional abstraction of IP cores, the System Generator blocks also provide the FPGA-literate designer access to key features in the underlying silicon, which is often necessary to achieve the highest performance

[1] System Generator is a registered trademark of Xilinx Inc.

[2] Simulink and Matlab are registered trademarks of Mathworks Inc.

implementation in an area-efficient manner. For example, the System Generator multiplier block has an option to target embedded high-speed 18x18 multipliers in the Virtex-II family of FPGAs.

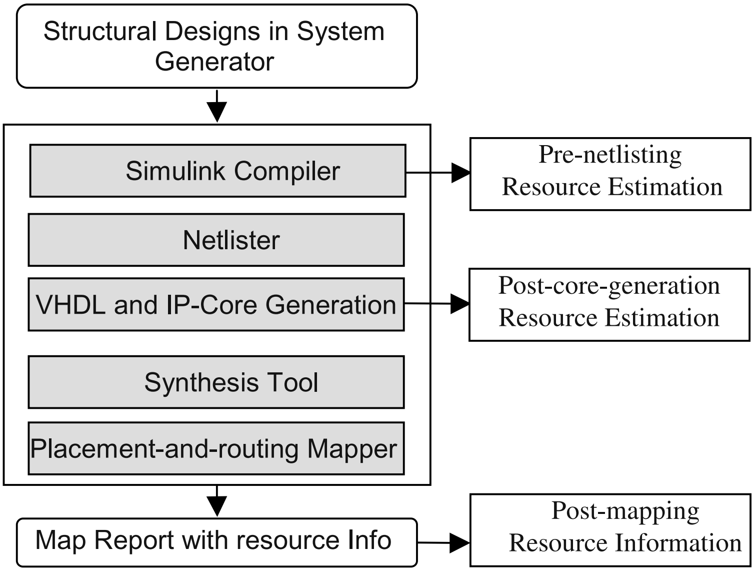

Structural Designs in System Generator
Simulink Compiler
Netlister
VHDL and IP-Core Generation
Synthesis Tool
Placement-and-routing Mapper
Map Report with resource Info
Pre-netlisting Resource Estimation
Post-core-generation Resource Estimation
Post-mapping Resource Information

Fig. 1. Resource estimation methods

2.2 Resource Estimation Methodologies

The hardware resources needed for a design are of course provided in the post technology mapping report. However, as shown in Fig.1, this information becomes available only after Simulink compilation, netlisting, IP-Core generation, synthesis and mapping stages. This entire process can take minutes or even hours, depending on the size of the system. The Simulink compile consumes only a small fraction of the time.

Fig. 1 also shows an estimator can simply sum the resource information available at each core after core-generation. However, because core generation is a significant fraction of the overall time (logic synthesis is the other major contributor), the method is relatively slow—often only a couple times faster than to map-report method. Also, all of the synthesized logic other than the IP cores are not considered, making the results inaccurate as well.

There are several straightforward approaches to pre-netlisting resource estimation. One is to build a database listing the resources given all the possible combination of a particular block. Each entry of in the database is obtained by a complete design experiment. However, an initial implementation[3] of this methodology showed that many blocks involved so many parameter combinations that the tests would have easily

[3] This was tried by C. Shi under Prof. Robert W. Brodersen's advice at University of California, Berkeley.

taken months or more of computing time to complete. Even were this done, the database for some blocks would consume hundreds of Megabytes and no longer be practical. Therefore, for only a few blocks with a small number of parameter combinations was this method is useful (it is indeed used sparsely in our current estimation tool).

An alternate approach is to build database for IP-cores only, while estimating all other synthesized logics by simple functions [3]. This method still suffers from the complexity resulting from highly parameterizable cores that make exhaustive elaboration impractical. One way to reduce the complexity is by ignoring selected block parameters or parameter ranges, but doing so leads to less accurate estimation (e.g. up to 30% or more [2]).

A common problem associated with all these “IP core as black box” estimation methods is the lack of understanding of IP-core design. Our methodology is based on essentially complete reverse-engineering the IP-core designs. Logic trimming caused by synthesis and mapping tools are accounted for by applying detailed knowledge of the synthesis and mapping algorithms, validated by benchmark experiments.

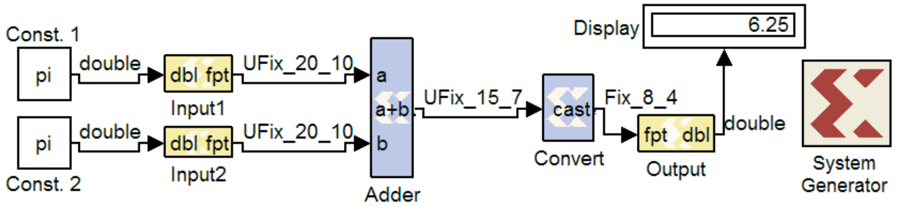


Fig. 2. A simple System Generator design with block output data-type displayed. Here UFix_20_10 means an unsigned signal with 20 bits in total and 10 bits of them are fractional. UFix_15_7 is defined accordingly. Fix_8_4 is a 2's-complement signed signal with 8 bits in total and 4 of them are fractional. This design shows the trimming effects.

2.3 Resource Estimation at the System Level

Fig. 2 shows a simple design in System Generator composed by some basic blocks. To estimate the resources of one type of blocks, such as adders, a MATLAB function in the following framework is written and called:

```
function tarea=get_BlockType_area(system)
% find all blocks in system of a particular masktype
r = find_system(system, ‘masktype',…)
% Initiate a Simulink compilation if needed
for i=1:length(r),
    % Get data-types of block inputs and outputs
    % and other block parameters to pass to subfunction
    get_param(r{i},…);
    % Use a dedicated function to get the resource
    [area(i,:), input_type]=BlockType_area(...);
    % update the resource info for block r{i}
end
```

```
% End the Simulink compilation if it has not been done
% Get the total area of the particular type
tarea=sum(area,1);
```

The functions `find_system()` and `get_param()` are Simulink model construction commands [8]. They allow the control of Simulink system using Matlab scripts, which makes the design automation possible. Simulink compilation is needed to retrieve signal (port) data-types and to compute those formula-based or hierarchically defined mask parameters.

In Fig. 2, a full-precision adder would grow its input data-type to UFix_21_10 (with one more integer bit than the input to accommodate overflow). But as the user defines that only 15 bits of the adder output are needed, some of the adder logic will be trimmed away during synthesis or mapping. This trimming effect is referred as *block-level trimming* and is further studied in section 2.4. Furthermore, if the Convert block uses truncation mode at the LSB side and wrap-around mode on the MSB side, the synthesis tool or the mapper will directly propagate its output word lengths backward to its input, making the true output of the adder block as UFix_8_4 instead of UFix_15_7. As a result, more adder logic will be trimmed away during synthesis. This trimming mechanism is referred as *global trimming*. In the current version of the tool, the global trimming effects are not implemented.

2.4 Block-Level Resource Estimation

A MATLAB function is written for each type of block, initiated as

```
            Function [area,input_type]=
                       BlockType_area(block_params).
```

Normally, each `BlockType_area()` function is written in the following steps,

- case-divide the following steps according to block parameters;
- understand the data-types for output wrapper and all the sub-cores;
- calculate resources for the wrapper and each sub-core accounting for trimming effects;
- sum different resources together;
- get input data-types after backward trimming effects .

Whenever possible, MATLAB vectors are used to speed-up the calculation. Particular attention is paid to take care the block-wise trimming effects. The last step of this procedure is necessary for global trimming. Even a detailed understanding of the algorithm for a block function is usually not sufficient; extensive testing using mapping reports is done to ensure the estimation function gives less than 5% relative error. The following two subsections illustrate these steps using two examples.

2.4.1 Resource Estimation for an Adder/Subtractor Block

The adder/subtractor block (AddSub), is used as the first example. In the `get_addsub_area()` function, the following function is called

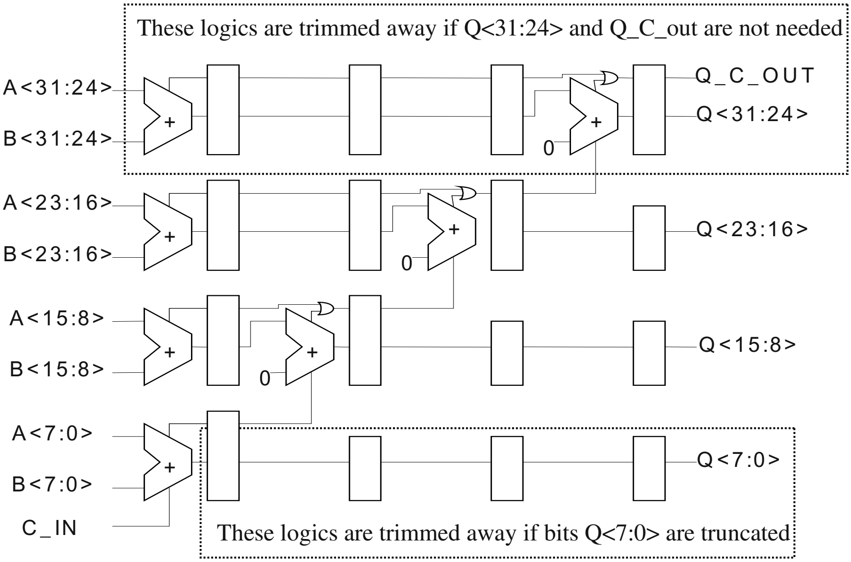

These logics are trimmed away if Q<31:24> and Q_C_out are not needed
A<31:24>
B<31:24>
Q_C_OUT
Q<31:24>
A<23:16>
B<23:16>
Q<23:16>
A<15:8>
B<15:8>
Q<15:8>
A<7:0>
B<7:0>
C_IN
Q<7:0>
+
0
These logics are trimmed away if bits Q<7:0> are truncated

Fig. 3. A possible realization of a 32-bit adder. The blank boxes denote pipeline flip-flops. The logics in the dashed box would be trimmed away if output Q<7:0> is truncated at the output.

```
function [area, input_type]= addsub_area(at_a, wa,wfa,
at_b, wb, wfb, at_o, wo, wfo, prec, q, o, latency,
use_core, use_rpm, pipeline, mode)
% This particular function contains about 200 lines of
% Matlab code that are not shown here.
```

As can be seen, even a simple block has a significant number of parameters that affect block resource usage. The variable *at_a* is the arithmetic type of input *a*; *wa* is the total wordlength of input a; *wfa* is the fractional wordlength of input *a*; *at_b*, *wb*, and *wfb* are similarly defined for input b; *at_o*, *wo*, and *wfo* are similarly defined for the output; *q* and *o* are the output quantization and overflow modes; *latency* is the excess latency at the output; *use_core* indicates whether the add/sub block is generated using IP-core or synthesized from an RTL module; *use_rpm* indicates whether the block uses placement information (a Relationally Placed Macro); *pipeline* indicates whether the block is pipelined internally to the greatest extent; *mode* indicates the block is an adder, subtractor or add/sub combination. All these block parameters are obtained in `get_addsub_area()` function before it calls `addsub_area()`.

Experiments show that using an RPM results slightly higher resources, but usually negligible. The three modes—subtractor, add/sub or adder—usually take similar resources; the difference is negligible except that when two unsigned numbers add each other, some logic LUTs will be replaced by route-through LUTs.

There are three main cases for the AddSub. They will be described in turn, followed by several key observations.

The first case is the pipelined AddSub using an IP-core. In this implementation, the adder is divided into pipelined sessions depending on the latency chosen. The main challenge in this case is to determine the implementation style of the core, based on the choice of latency and output width. This was a surprisingly non-trivial task, accomplished by reverse-engineering the core.

The second case is the non-pipelined AddSub using an IP-core. Here, the challenge was to determine the LUT count including route-though LUTs. The slices containing LUTs absorb one level of latency, and the excess latency is handled by an SRL17 (SRL + output register) implementations. All LUTs on the most-significant-bit side are trimmed away when these bits are not needed, whereas only the flip flops and shift-register LUTs are removed in the least significant parts, as shown in Fig. 3.

The third case is the RTL AddSub, which is similar to the non-pipelined IP core, with an important difference. First, all latency are handled by SRL17s. Second, some of the least-significant-bit logic can be trimmed when one input has trailing zeroes.

Summarizing the approach to estimating a LUT-based block, the area function is designed according to the algorithm used to implement the function, the logic wrapper that provides the fixed-point arithmetic abstraction, and taking into account logic trimming that occurs due to parameters that include input and output data types.

Finally, possible trimming on input bits is described in the `addsub_area()` function, as needed for handling global trimming effects in the future, when the compiler is able to handle backward data-type propagation.

2.4.2 Resource Estimation for the Usage of 18x18 Embedded Multipliers

As another example of writing block level resource estimation function, let’s look at the usage of 18x18 embedded multipliers that are currently available in Virtex-II family FPGAs. When the target multiplier is less than 18x18 size, it can be fit into one embedded multiplier. Otherwise, multiple embedded multipliers are needed, each of which generates a partial product, followed by adder logic to sum the partial products into the final output.

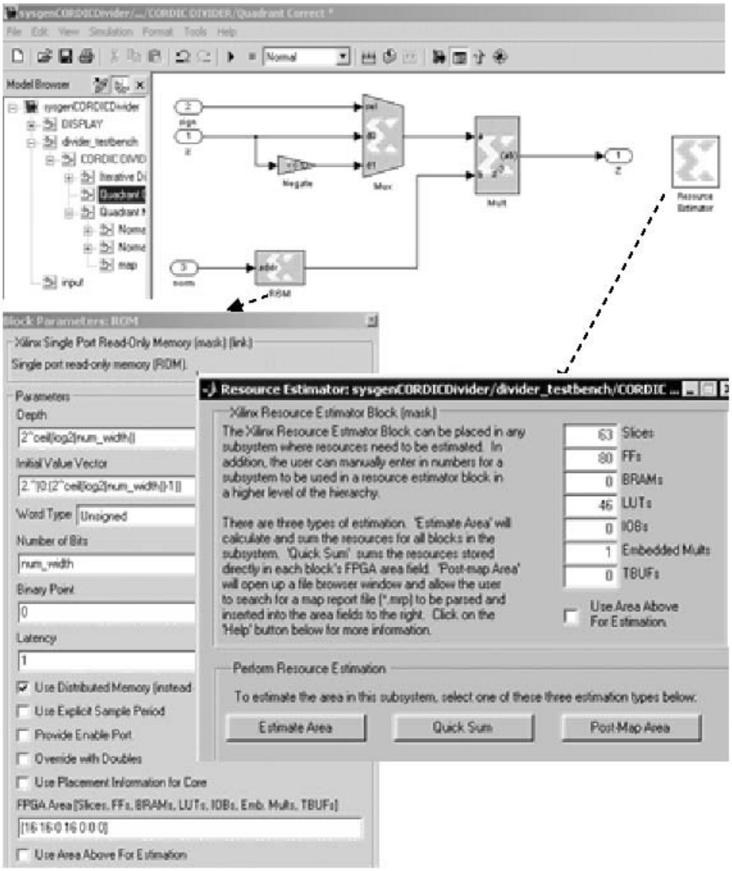
By understanding the way the embedded multiplier is used, the usage of these embedded primitives can be written as a simple function of the parameters of the target multiplier, that is,

$$\begin{aligned}&\text{Number of } 18\times 18 \text{ Embedded Mults in a Multiplier} =\\&\mathrm{Ceil}\left((N_A + \mathrm{Unsigned}_A - 1)\big/ 17\right)\times \mathrm{Ceil}\left((N_B + \mathrm{Unsigned}_B - 1)\big/ 17\right)\end{aligned} \tag{1}$$

where subscripts A and B denote the two inputs of the multiplier, N_A denote the number of bits of input A, Unsigned_A is either 1 or 0 representing signal A is unsigned or signed, similarly for B. and ceil() is the ceiling function as defined in MATLAB. The total number of 18x18 multiplier primitives used in a model is simply the sum of the numbers for each parallel multiplier.

2.5 User Interface and Design Automation

As shown in Fig. 4, every Xilinx block that requires FPGA resources has a mask parameter that stores a vector containing its resource requirements. The Resource Estimator block can invoke underlying functions to populate these vectors (e.g. after parameters or data types have been changed), or aggregate previously computed values



Block Parameters: ROM

Xilinx Single Port Read-Only Memory (mask) (link)

Single port read-only memory (ROM).

Parameters

Depth: 2^ceil(log2(num_width))

Initial Value Vector: 2.^(0:(2^ceil(log2(num_width))-1))

Word Type: Unsigned

Number of Bits: num_width

Binary Point: 0

Latency: 1

☑ Use Distributed Memory (instead

☐ Use Explicit Sample Period

☐ Provide Enable Port

☐ Override with Doubles

☐ Use Placement Information for Core

FPGA Area [Slices, FFs, BRAMs, LUTs, IOBs, Emb. Mults, TBUFs]: [16 16 0 16 0 0 0]

☐ Use Area Above For Estimation

Resource Estimator: sysgenCORDICDivider/divider_testbench/CORDIC ...

Xilinx Resource Estimator Block (mask)

The Xilinx Resource Estimator Block can be placed in any subsystem where resources need to be estimated. In addition, the user can manually enter in numbers for a subsystem to be used in a resource estimator block in a higher level of the hierarchy.

There are three types of estimation. 'Estimate Area' will calculate and sum the resources for all blocks in the subsystem. 'Quick Sum' sums the resources stored directly in each block's FPGA area field. 'Post-map Area' will open up a file browser window and allow the user to search for a map report file (*.mrp) to be parsed and inserted into the area fields to the right. Click on the "Help" button below for more information.

63	Slices
80	FFs
0	BRAMs
46	LUTs
0	IOBs
1	Embedded Mults
0	TBUFs

☐ Use Area Above For Estimation

Perform Resource Estimation

To estimate the area in this subsystem, select one of these three estimation types below.

Fig. 4. System Generator design and the Resource estimation utilities for the Cordic divider demo using distribute memory [4]. The bottom left pane shows the mask parameters of the ROM; the bottom right pane is the block parameter dialog box of the Resource Estimator block that shows the estimation result of the current subsystem.

that have been stored in the vectors. Each block has a checkbox control "Use Area Above for Estimation" that short-circuits invocation of the estimator function and uses the estimates stored in the vector instead.

In Fig. 4, by clicking “Estimate Area” button of the parameter dialog box of the resource estimator block, a Simulink compilation is initiated. When the compilation is done, the `get_BlockType_area()` resource estimation functions are called for every block in the subsystem containing the resource estimator, which in turn calls the `BlockType_area()` core functions to get the estimated area. The results are then displayed in the dialog box of the Estimator. Furthermore, the resource vector of each individual block, such as the ROM in Fig. 4, is also updated and displayed. More detailed information is available in [4].

3 Experimental Results

The MATLAB function `BlockType_area()` for each block type has been tested extensively, sometimes exhaustively, against the mapping report result under various block configurations. In this section, the complete resource estimation tool is further tested against a number of complicated DSP designs, two of which are reported here. One design is an additive-white-Gaussian-noise (AWGN) simulator that generates pseudo-random AWGN noise. The other one is a stage-based Coordinate rotation digital computer (CORDIC) system. Table 1 shows the results on the 7 aspects of the FPGA resources, as well as the time required to get the estimations.

Table 1. Compare the proposed resource estimation tool (new tool) with map-report (previous tool) on a couple designs. AWGN is an additive-white-Gaussian-noise simulator.

	Slices	FFs	BRAMs	LUTs	IOBs	18x18 Mults	TBUFs	Time (min)
AWGN (Prev. tool)	1571	760	0	1595	27	1	0	15
AWGN (New tool)	1606	760	0	1612	27	1	0	.5
11-stages CORDIC (Prev. tool)	453	952	1	773	101	0	0	10
CORDIC (New tool)	471	982	1	794	101	0	0	.3

The results in Table 1 are representative of many other tests. Every aspect of the resources obtained from the new resource estimation tool[4] agrees with the map-report within 10% (usually within 5%). Yet, the estimation time speeds up by 1-2 orders of magnitude comparing with map-report method. This acceleration is accomplished by the elimination of time-consuming netlisting, synthesis, and mapping stages required to get a map-report.

As long as a System Generator design can be compiled by the Simulink compiler, the proposed resource estimation tool is able to estimate. In this way, resource estimation can be obtained for early designs iterations that cannot even pass the rest of the design flow to reach the map-report stage.

4 Conclusion

A novel pre-netlisting FPGA resource estimation tool in System Generator has been developed. The estimation is accurate because the architectural information of a design is available in System Generator. Is also because IP-core implementations and trimming effects are understood. Automation of the tool is realized in MATLAB functions by taking advantage of the Simulink model construction commands. Verifications on real designs show excellent agreement with map-report.

[4] Comparisons of the new estimator were performed against a previous version implemented by the first author as part of his graduate research.

Further developments can be done in several possible areas. First, a dominant portion of the estimation time is spent on Simulink compilation to obtain the data-types at signal nodes, so, more a efficient compiler would speed-up the estimation tool. Secondly, the global trimming effects could be important in some designs, and can be computed by a smarter Simulink compiler that can propagate signal data-types both forward and backward. Thirdly, it would be desireable to develop similar estimation tools for power-consumption and signal path delays.

Acknowledgements. Changchun Shi would like to thank his Ph.D. advisor, Prof. Robert W. Brodersen, for his constant support and advice on this project. He also wants to thank the rest of the System Generator group, IP-core group, XST synthesis group and David Squires at Xilinx for their useful suggestions and support.

References

1. W. R. Davis, *et al*, "A design environment for high-throughput low-power dedicated signal processing systems," *IEEE Journal of Solid State Circuits*, vol. 37, No. 3, pp. 420-431, Mar. 2002.
2. C. Chen, *et. al.*, "Rapid Design and analysis of communication systems using the BEE hardware emulation environment," *RSP*, 2003.
3. A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate Area and Delay Estimators for FPGAs," *Proc. Design Automation and Test in Europe*, Mar. 2002, Paris, France.
4. Xilinx System Generator, Xilinx, Inc., [online]. Available: http://www.xilinx.com. Once there, search on "System Generator" and "System Generator Resource Estimation" for related information.
5. C. Shi, and R. W. Brodersen, "Automated fixed-point data-type optimization tool for DSP and communication systems." Submitted to *Design Automation Conf.*, June 2004
6. C. Shi, and R. W. Brodersen, "An automated floating-point to fixed-point conversion methodology," *Proc. IEEE Int. Conf. on Acoust. Speech, and Signal Processing*, Vol. 2, pp. 529-532, April 2003, Hong Kong, China
7. J. Hwang, B. Milne, N. Shirazi, and J. Stroomer, "System Level Tools for DSP in FPGAs," *Field-Programmable Logic and Applications, FPL 2001 Proceedings*, Aug. 2001, pp. 534-543, Springer- Verlag, 2001.
8. Matlab and Simulink, Mathworks, Inc., [online]. Available: http://www.mathworks.com
9. Xilinx, Inc., Virtex-II Pro: Platform FPGA Handbook, Oct. 2002.

The PowerPC Backend Molen Compiler

Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis

Computer Engineering Lab
Delft University of Technology, The Netherlands
{E.Panainte, K.Bertels, S.Vassiliadis}@et.tudelft.nl

Abstract. In this paper, we report on the backend C compiler developed to target the Virtex II Pro PowerPC processor and to incorporate the Molen architecture programming paradigm. To verify the compiler, we used the multimedia video frame M-JPEG encoder of which the Discrete Cosine Transform (DCT*) function was mapped on the FPGA. We obtained an overall speedup of 2.5 against a maximal theoretical speedup of 2.96. The performance efficiency of 84 % is achieved using automatically generated but non-optimized DCT* hardware implementation.

1 Introduction

Reconfigurable computing (RC) is becoming increasingly popular as it bears the promise of combining the flexibility of software with the performance of hardware. Some concern can be expressed because the current state of the art assumes that the developer has a deep understanding of both software and hardware development before the benefits of this technology can be exploited. This justified concern underlines the necessity to intensify research and development efforts to support the designer in this process. The Delft Workbench is an initiative that investigates the integration and development of tools supporting the different design phases starting at code profiling, synthesis and ending at the generation of binary code. The idea is to automate as much as possible the design exploration and the final development process. This paper addresses an important part of the tool chain, namely the construction of a backend compiler that targets such a hybrid platform. The compiler allows on the basis of function annotations, the automatic modification of applications to generate the appropriate binaries.

The current paper reports on the completed compiler targeting the Molen implementation on the Virtex II Pro platform FPGA. The contributions of the paper can be summarized as follows :

- A compiler backend targeting the PowerPC processor included in the Molen prototype has been developed.
- The theoretical compiler extensions presented in [1] have been implemented and adjusted to the target Field-programmable Custom Computing Machine (FCCM) features.
- Software/hardware development tools have been integrated to automatize the design flow phases.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 434–443, 2004.

The application that was used for experiments is M-JPEG encoder. Measurements show that the resulting code executed on the implementation of the Molen organization on the Virtex II Pro board, allows to obtain overall speedups of 2.5 when compared to the software only execution. We emphasize that the goal of this experiment is not to study multimedia extensions but rather to provide a proof of concept of the compiler toolset targeting FCCMs. We also stress that in contrast to the work discussed in [1], the presented paper bases all experimentation on a real Molen prototype rather than estimations.

The paper is organized as follows. In the next section, we present the Molen organization and discuss related work. In section 3, we present the compiler extensions required for the PowerPC processor and the Molen prototype. We then present the case study where the computation intensive DCT function is mapped on the reconfigurable fabric and show that speedups of 2.5 are achieved.

2 Background and Related Work

In this section, we briefly discuss the Molen programming paradigm [2], describe the Molen machine organization that supports it and discuss related work.

The Molen programming paradigm [2] is a sequential consistency paradigm for programming FCCMs possibly including a general-purpose computational engine(s). The paradigm allows for parallel and concurrent hardware execution and is intended (currently) for single program execution. For a given ISA, a one time architectural extension (based on the co-processor architectural paradigm) comprising 4 instructions (for the minimal πISA as defined in [2]) suffices to provide an almost arbitrary number of operations that can be performed on the reconfigurable hardware. The four basic instructions needed are **set**, **execute**, **movtx** and **movfx**. By implementing the first two instructions (**set**/**execute**) an hardware implementation can be loaded and executed in the reconfigurable processor. The **movtx** and **movfx** instructions are needed to provide the communications between the reconfigurable hardware and the general-purpose processor (GPP). The Molen machine organization [3] that supports the Molen programming paradigm is described in Figure 1. The two main components in the Molen machine organization are the 'Core Processor', which is a GPP and the 'Reconfigurable Processor' (RP). Instructions are issued to either processors by the 'Arbiter' by means of a partial decoding of the instructions received from the instruction fetch unit. The support for the SET/EXEC instructions required in the Molen programming paradigm is based on *reconfigurable microcode*. The reconfigurable microcode is used to emulate both the configuration of the Custom Computing Unit (CCU) and the execution of implementations configured on the CCU. A detailed description of how the Molen organization and programming paradigm compare with other approaches is presented in [1].

An overview of research that aims to combine GPPs and reconfigurable hardware and to provide software support for programming these FCCMs and a discussion of how they relate to research reported in this paper includes the following:

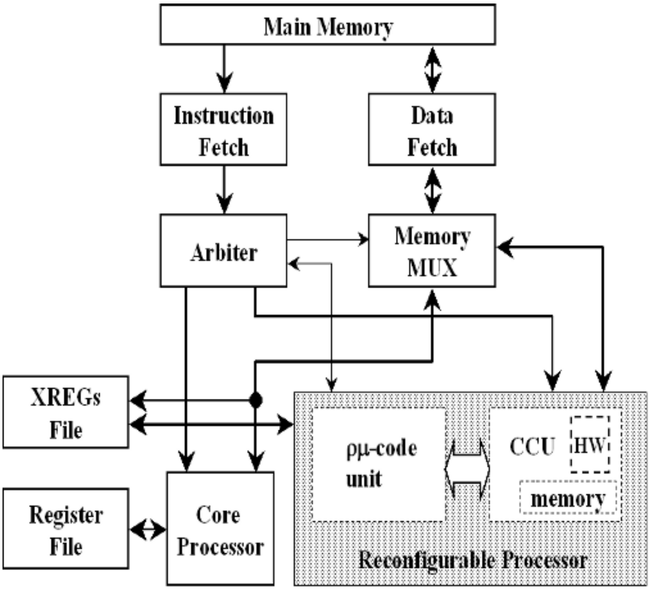


Fig. 1. The Molen machine organization

Reconfigurable Architectures Performance: Several reconfigurable architectures have been proposed in the last decade (see [4] for a classification). The reported performance improvements are mainly based on simulation (see for example [5]) or estimation (e.g. [6] [7]) results. Eventhough some implementations exist [8], in most cases the performance is just estimated. In this paper, we present the validation of the Molen approach based on a real and running implementation of the Molen reconfigurable processor platform.

Compilers for Reconfigurable architectures: When targeting hybrid architectures to improve performance, the applications must be partitioned in such a way that certain computation intensive kernels are mapped on the reconfigurable hardware. Such mapping is not simple as it assumes deep understanding of both software and hardware design. Several approaches (e.g. [5] [7]) use standard compilers to compile the applications to FCCMs. As standard compilers do not target reconfigurable architectures, the kernel computations implemented in hardware are manually replaced by the appropriate instructions for communication with and controlling the reconfigurable hardware. This replacement is done manually and it is a time-consuming [9] and error-prone process. In order to facilitate the design and development process, much effort is put in the development of automated tools (compilers) to perform such tasks [10] [6] [11]. However, the extensions of the cited compilers mainly aim to generate the instructions for the reconfigurable hardware and they are not designed to easily support new optimizations that exploit the possibilities of the reconfigurable hardware. The Molen compiler presented in this paper, is based on a a flexible and extensible infrastructure that allows to add easily new optimization and analysis passes that take into account the new features of the target reconfigurable architecture.

Compiler Support for Hiding the Reconfiguration Latency: One of the major drawbacks of the reconfigurable hardware is the the huge reconfiguration latency [12] [4]. Different techniques such as configuration caching and prefetching (e.g. [3]) have been proposed to reduce the reconfiguration latency. These hardware techniques should be combined with compiler optimizations that provide an efficient instruction scheduling to use the available parallelism between different FCCMs components in the hardware reconfiguration phase. Nevertheless, many FCCMs do not expose a specific instruction for hardware reconfiguration (see [4] for FCCMs classification), thus impeding compiler support for hiding reconfiguration latency. We schedule the SET instruction (which performs the hardware configuration) as early as possible from the hardware execution phase, resulting in exploiting the parallelism between the GPP and the FPGA during the configuration stage.

3 The Molen Compiler

The Molen compiler comprises the Stanford SUIF2[13] (Stanford University Intermediate Format) Compiler Infrastructure for the front-end and the Harvard Machine SUIF framework[14] for developing compiler backends and optimization. In [1], the theoretical compiler extensions target a virtual Molen reconfigurable architecture including an *x86* processor as a GPP. In this section we present the implemented compiler backend and extensions required for the Molen hardware implementation on the Virtex II Pro platform FPGA which includes a PowerPC processor.

PowerPC Backend: The first step is to have a backend C-compiler that generates the appropriate binaries to be executed on the PowerPC processor integrated on the Virtex II Pro board. Current MachineSUIF backends excluded the backend for PowerPC architecture. In consequence, we developed a PowerPC compiler backend and implemented the PowerPC instruction generation, PowerPC register allocation, PowerPC EABI stack frame allocation and software floating-point emulation (not completed). Additionally, in order to exploit the opportunities offered by the reconfigurable hardware, the PowerPC backend has to be extended in several directions, as described in the rest of this section.

Hiding Configuration Latency: Due to the lack of support for dynamic reconfiguration in the current Molen implementation (there was not sufficient information about the Virtex II Pro platform) and taking into account that in our experiments there is only one function (DCT*) executed on the reconfigurable hardware, the Molen compiler generates in advance only one SET instruction for DCT* at the application entry point. The SET instruction does not stall the GPP implying that the compiler can issue this instruction as far ahead as possible from the hardware execution phase. This scheduling allows the GPP to execute in parallel with the FPGA during the configuration stage. This is particularly useful for the cases when the SET instruction is initially included in a loop. Thus, issuing the SET instruction at the application entry point avoids unnecessary repetitive hardware configuration. The cases when multiple operations are sequentially executed on the reconfigurable hardware and do not simultaneously fit on the target FPGA are not covered by this scheduling.

Compiler Extensions for Molen Implementation: First of all, a general purpose reconfigurable architectural scheme (presented in [3]) has been adopted. We implemented the minimal instruction set extension, containing the following:

- SET/EXECUTE instructions are included in the MIR (Medium-level Intermediate Representation) and LIR (Low-level Intermediate Representation) of the Molen compiler. In order not to modify the PowerPC assembler and linker, the compiler generates the instructions in binary form. For example, for the instruction *exec 0x80000C* the generated code is *.long 1A000031* where the encoding format (presented in [15]) is recognized by the arbiter.
- MOVTX/MOVFX: The equivalent PowerPC instructions are *mtdct/mfdcr*. Moreover, the XRs (exchange registers) are not physical registers but they are mapped at fixed memory addresses.

```
la 3, 12016(1)         #load the address of the first param
la 12, 12016(1)        #load the address of the second param
mtdcr 0x00000056,3     #send the address of the first parameter
mtdcr 0x00000057,12    #send the address of the second parameter
sync                   #
nop                    #synchronization
nop                    #
nop                    #
bl main._label0        #instr. required by the arbiter impl.
main._label0:
.long 0x1A000031       #exec 0x8000C
nop                    #synchronization
```

Fig. 2. Code generated by the Molen compiler for the reconfigurable DCT* execution

In Figure 2, we present the code generated by the Molen compiler for the DCT* function call executed on the reconfigurable hardware. In order to correctly generate the instructions for hardware configuration and execution, the compiler needs information about the DCT* hardware implementation. This information is described in an FPGA Description File, which contains for the DCT* operation the fields presented in Figure 3. Line 2 defines the start memory address from where the XRs are mapped. In line 3, the compiler is informed that there is a hardware implementation for the DCT* operation with the microcode addresses for SET/EXECUTE instructions defined in lines 4-5. The *sync* instruction from Figure 2 is a PowerPC instruction that ensures that all instructions preceding *sync* in program order complete before *sync* completes. The sequences of *sync* and *nop* instruction are used to flush the processor pipeline. The SET instruction is not included in the above example because it has been scheduled earlier by the Molen compiler previously presented.

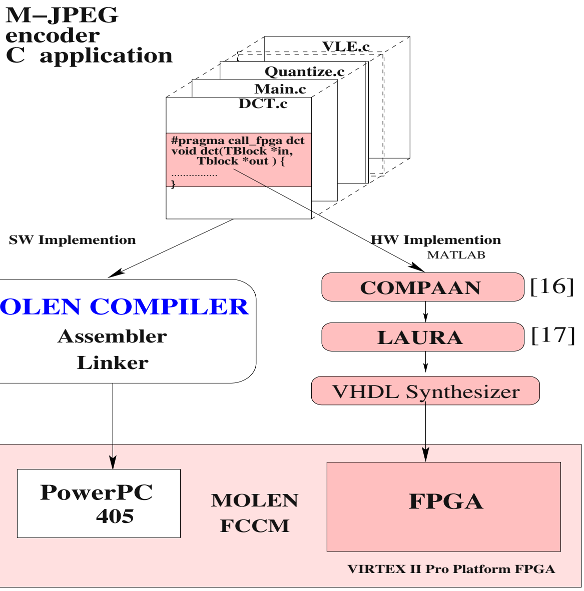
```
NO_XRS       = 512      # number of available XRs
START_XR     = 0x56     # the memory address of the first XR
OP_NAME      = dct      # info about the DCT* operation
SET_ADDR     = 0x39A100 # the address of the DCT* SET microcode
EXEC_ADDR    = 0x80000C # the address of the DCT* EXEC microcode
END_OP                  # end of the info about the DCT* operation
.................................  # info about other operations
```

Fig. 3. Example of an FPGA Description File

4 M-JPEG Case Study

In this case study we report the performance improvements of the Molen implementation on the Virtex II Pro for the multimedia video frame M-JPEG encoder.

Design Flow: The design flow used in our experiments is depicted in Figure 4. In the target application written in C, the software developer introduces pragma annotations for the functions implemented on the reconfigurable hardware. These functions are translated to Matlab and processed by the COMPAAN[16]/LAURA[17] toolchain to automatically generate the VHDL code. The commercial tools can then be used to synthesize and map the VHDL code on the target FPGA. The application is compiled with the Molen compiler and the executable is loaded and executed on the target Molen FCCM.



```
#pragma call_fpga dct
void dct(TBlock *in,
     Tblock *out ) {
...............
}
```

Fig. 4. The design flow

Table 1. M-JPEG video sequences

Name	# frames	Resolution [pixels]	Format	Color/BW
tennis	8	48x48	YUV	color
barbara	1	48x48	YUV	color
artemis	1	48x48	YUV	color

M-JPEG, Software and Hardware Implementations: The application domain of these experiments is the video data compressing. We consider a real-life application namely Motion JPEG (M-JPEG) encoder which compresses sequences of video frames applying JPEG compression for each frame. The input video-frames used in these experiments are presented in Table 1. The M-JPEG implementation is based on the public domain implementation described in ”PVRG-JPEG CODEC 1.1”, Portable Video Research Group, Stanford University. The most demanding function in M-JPEG application is 2D DCT with preshift and bound transforms (named in this paper as DCT*). In consequence, DCT* is the first function candidate for hardware implementation. The only modification of the M-JPEG application that indicates the reconfigurable DCT* execution is the introduction of the pragma annotation as presented in Figure 4. The hardware implementation for execution of the DCT* function on the reconfigurable hardware is described in [9]. The VHDL code is automatically extracted from the DCT* application code using COMPAAN[16]/LAURA[17] tools. The Xilinx IP core for DCT and the ISE Foundation[18] are used to synthesize and map the VHDL code on the FPGA. After the whole application is compiled with the Molen compiler described in the previous section, in the final step we use the GNU assembler and linker and the C libraries included in the Embedded Development Kit (EDK) [19] from Xilinx to generate the application binary files. The target FCCM is the implementation of the Molen reconfigurable architecture on the Virtex II Pro platform FPGA of Xilinx described in [15]. In this implementation, the GPP is the IBM PowerPC 405 processor immersed into the FPGA fabric.

Performance Measurements: The current Molen implementation is a prototype version, which imposes the following constraints:

- the memory size for *text* and *data* sections are limited to maximum 64K. In order for the M-JPEG executable to fulfill these limitations, we rewrote the original application preserving only the essential features for compressing sequences of video frames. Moreover, these limitations also restrict the size of the input video frames to 48x48 pixels (Table 1, column 3).
- dynamic reconfiguration is not supported (yet) on the Molen prototype. In consequence, we could not measured the impact on performance of repetitive hardware configurations.

In addition, the performance measurements have been performed given the following additional conditions:

- the input/output operations are extremely expensive for the current Molen prototype, due to the standard serial connection implemented by UART at

38400 bps between the Molen prototype and the external environment; this limitation can be removed by the implementation of faster I/O system. We therefore did not include the I/O operation impact in our measurements as they are not relevant for RC paradigm

- the DCT* hardware implementation requires a different format for the input data than the software implementation. Consequently, an additional data format conversion is performed in software before and after the DCT* execution on reconfigurable hardware.
- taking into account that the target PowerPC processor included in the Virtex-II Pro platform does not provide hardware floating-point support and that the required floating-point software emulation is extremely expensive, the integer DCT is used for both software and hardware implementation to allow a fair comparison.

The execution cycles for M-JPEG encoder and comparisons are presented in Table 2. As we considered a sequence of 8 video frames for *tennis* input sequence, we present only the minimal and maximal values for each measurement in order to avoid redundant information.

Pure Software Execution: In Table 2(a), we present the results of our measurements performed on the the Virtex II Pro platform, when the M-JPEG application is entirely executed on the PowerPC processor. In row 1, the number of cycles used for executing the whole M-JPEG application is given. In row 2, the cycles consumed by one execution of the DCT* function are given and the next row contains the total number of cycles spent in DCT*. From these numbers, we can conclude that 66% of the total execution time is spent in the DCT* function, given the input set. This 66% represents the maximum (theoretical) improvement that can be obtained by hardware acceleration of the DCT* function. The corresponding theoretical speedup - using *Amdahl's law* - is presented in Table 2(c), row 2.

Execution on the Molen prototype: In Table 2(b), we present the number of cycles for the M-JPEG execution on the Molen prototype. From row 1 we can conclude that an overall speedup of 2.5 (Table 2(c), row 1) is achieved. The DCT* execution on the reconfigurable hardware takes 4125 cycles (row 2) which is around 300 times less than the software based execution on the PowerPC processor (Table 2(a), row 2). However, due to the data format conversion required by the DCT* hardware implementation, the overall number of cycles for one DCT* execution becomes 102,589 (Table 2(b), row 3), resulting in a 10 fold speedup for DCT* and a 2.5 speedup globally. The performance efficiency is about 84% as presented in Table 2(c), last column. It is noted that this efficiency is achieved even though i) the hardware implementation has been automatically but non-optimally obtained (using COMPAAN[16]/LAURA[17] tools) and ii) additional software data conversion diminished the DCT* speedup in hardware. From these measurements, we can conclude that even non-optimized implementation can be used to achieve considerable performance improvements. In addition, taking into account that only one function (DCT*) is executed on the reconfigurable hardware, we consider that an overall M-JPEG speedup of

Table 2. M-JPEG execution cycles and comparisons

		tennis [0-7]		barbara	artemis
		MIN	MAX		
Pure Software Execution (a)	M-JPEG	33,671,821	33,779,813	34,014,157	34,107,893
	DCT*	1,242,017	1,242,017	1,242,017	1,242,017
	DCT* cumulated	22,356,306	22,356,306	22,356,306	22,356,306
	Maximal improvement	66.18%	66.39%	65.73%	65.55%
Execution on Molen prototype (b)	M-JPEG	13,443,269	13,512,981	13,764,509	13,839,757
	DCT* HW	4,125	4,125	4,125	4,125
	DCT* HW + Format conv.	102,589	102,589	102,589	102,589
Comparison (c)	Practical speedup	2.50	2.51	2.47	2.46
	Theoretical speedup	2.96	2.98	2.92	2.90
	Efficiency	84.17%	84.65%	84.70%	84.91%

2.5x from the theoretical speedup of 2.96 x confirm the viability of the presented approach.

5 Conclusions

In this paper, we presented the implemented compiler support for the Molen implementation on the Virtex II platform FPGA. The compiler allows the automatic modification of the application source code using the extensions following the Molen Programming Paradigm. The experiment evaluated the effectively realized speedup of reconfigurable hardware execution of the DCT* function of the M-JPEG application. The generated code was executed on the Molen prototype and showed a 2.5 speedup. This speedup consumed 84% of the total achievable speedup which amounts to 2.9. Taking into account that hardly any optimization was performed and only one function ran on the reconfigurable fabric, a significant performance improvement was nevertheless observed. We emphasize that we do not compare the RC paradigm to other approaches for multimedia applications boosting performance (such as MMX, 3DNow!, SSE). The focus of this paper was rather on the compiler support for the Molen FCCM under the RC paradigm. Further research on the compiler will address optimizations for dynamic configurations and parallel execution on the reconfigurable hardware.

References

1. Moscu Panainte, E., Bertels, K., Vassiliadis, S.: Compiling for the Molen Programming Paradigm. In: 13th International Conference on Field Programmable Logic and Applications (FPL). Volume 2778., Lisbon, Portugal, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2003) 900–910

2. Vassiliadis, S., Gaydadjiev, G., Bertels, K., Moscu Panainte, E.: The Molen Programming Paradigm. In: Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece (2003) 1–7
3. Vassiliadis, S., Wong, S., Cotofana, S.: The MOLEN $\rho\mu$-Coded Processor. In: 11th International Conference on Field Programmable Logic and Applications (FPL). Volume 2147., Belfast, UK, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2001) 275–285
4. Sima, M., Vassiliadis, S., S.Cotofana, van Eijndhoven, J., Vissers, K.: Field-Programmable Custom Computing Machines - A Taxonomy. In: 12th International Conference on Field Programmable Logic and Applications (FPL). Volume 2438., Montpellier, France, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2002) 79–88
5. Campi, F., Cappelli, A., Guerrieri, R., Lodi, A., Toma, M., Rosa, A.L., Lavagno, L., Passerone, C.: A reconfigurable processor architecture and software development environment for embedded systems. In: Proceedings of Parallel and Distributed Processing Symposium, Nice, France (2003) 171–178
6. Kastrup, B., Bink, A., Hoogerbrugge, J.: Concise: A compiler-driven cpld-based instruction set accelerator. In: Proceedings of FCCM'99, Napa Valley CA (1999) 92–100
7. Rosa, A.L., Lavagno, L., Passerone, C.: Hardware/Software Design Space Exploration for a Reconfigurable Processor. In: Proc. of DATE 2003, Munich, Germany (2003) 570–575
8. Lee, M.H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F.J.: Design and Implementation of the MorphoSys Reconfigurable Computing Processor. VLSI Signal Processing Systems **24** (2000) 147–164
9. Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.: System Design using Kahn Process Networks: The Compaan/Laura Approach. In: Proc. of DATE 2004, Paris, France (2004) 340–345
10. Gokhale, M.B., Stone, J.M.: Napa C: Compiling for a Hybrid RISC/FPGA Architecture. In: Proceedings of FCCM'98, Napa Valley, CA (1998) 126–137
11. Ye, Z.A., Shenoy, N., Banerjee, P.: A C Compiler for a Processor with a Reconfigurable Functional Unit. In: ACM/SIGDA Symposium on FPGAs, Monterey, California, USA (2000) 95–100
12. M. Bolotski, A. DeHon, Knight, J.T.F.: Unifying FPGAs and SIMD arrays. In: ACM/SIGDA Symposium on FPGAs, Berkeley, CA (1994) 1–10
13. (http://suif.stanford.edu/suif/suif2)
14. (http://www.eecs.hardvard.edu/hube/research/machsuif.html)
15. Kuzmanov, G., Vassiliadis, S.: Arbitrating Instructions in an $\rho\mu$-coded CCM. In: Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03). Volume 2778., Lisbon, Portugal, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2003) 81–90
16. Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In: Proc. of CODES'2000, San Diego, CA (2000) 13–17
17. Zissulescu, C., Stefanov, T., Kienhuis, B., Deprettere, E.: Laura: Leiden Architecture Research and Exploration Tool. In: 13th International Conference on Field Programmable Logic and Applications (FPL). Volume 2778., Lisbon, Portugal, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2003) 911–920
18. (http://www.xilinx.com/ise_eval/index.htm)
19. (http://www.xilinx.com/ise/embedded/edk.htm)

An Integrated Online Scheduling and Placement Methodology

Manish Handa and Ranga Vemuri

Department of ECECS, University of Cincinnati
Cincinnati, OH 45221-0030, USA
{mhanda, ranga}@ececs.uc.edu

Abstract. Dynamic task scheduling and online placement are two of the main responsibilities of an operating system for reconfigurable platforms. Since these operations are performed during run-time of the applications, these are overheads on the execution time. There is a need to find fast and efficient algorithms for task placement and scheduling. We propose an integrated online scheduling and placement methodology. We maintain empty area as a list of maximal empty rectangles which allows us to explore solution space efficiently. We defer scheduling decisions until it is absolutely necessary to accommodate dynamically changing task priorities. We propose task queue management data-structures for in-order and out-of-order task scheduling strategies. One of our queuing strategies guarantees the shortest execution time for in-order task execution and the other strategy results in better FPGA area utilization for out-of-order task execution. We provide experimental evidence of improvement our methodology yields over the previous approaches.

1 Introduction

FPGA performance has increased manifold in past few years. Underwood analyzed FPGA performance data between 1997 and 2003 to show that FPGA performance improvement trend is better than what is predicted by the Moore's law [1]. With the increase in the FPGA capacity, a multi-user multi-tasking FPGA is not a very distant possibility. In such a reconfigurable system, more than one application can concurrently run on the same FPGA device. A host is responsible for managing the reconfigurable resources and for execution and scheduling of the applications. The system services provided by the host constitute a reconfigurable operating system (ROS).

Users can log on to the host and submit applications represented in a high level language. These applications share the reconfigurable resources with some already executing applications. Each application is synthesized and temporally partitioned into a set of tasks. The tasks are scheduled and placed on the FPGA using partial reconfiguration. Each task needs a finite amount of reconfigurable resources for its execution. The reconfigurable resources can be modeled as a contiguous rectangular area on the FPGA. When a task finishes execution, the area is reclaimed by the ROS and can be reused for execution of subsequent tasks.

Though shape, size and lifetime of the tasks is known after synthesis, the execution flow of an application depends upon input data (e.g. non-manifest loops) and is decided

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 444–453, 2004.

only at the run-time. Further, a number of applications may be competing for the FPGA area resources. So the tasks need to be scheduled and placed at the run-time of the applications. Placement in such an environment is called *online placement*. Time needed for the scheduling and placement is an overhead and it delays execution of the application.

In a typical reconfigurable embedded system, a set of tasks compete for a limited number of resources. Scheduling is the process of assigning the resources to the tasks and to assign a start time and finish time for execution of a task. In this work, we focus on non-preemptive integrated online scheduling and placement methodology. We argue that online scheduling and online placement should be tightly integrated. Unlike previous approaches, our placement algorithm guarantees to find a placement location, if there exists one. We describe our task queue management data-structure for in-order and out-of-order task scheduling. One of our queueing strategies guarantees the shortest execution time for in-order task execution and the other strategy results in better FPGA area utilization for out-of-order task execution. We present detailed performance characteristics for both the scheduling methodologies.

2 Related Work

Scheduling of tasks on a multi-processor system or scheduling of data packets on a network is a well studied research area. A survey of online scheduling algorithms can be found in [2]. FPGAs are fast becoming an integral part of a typical embedded system. So, task scheduling for a reconfigurable system is fast becoming an important research area. Mei et al. [3]presented a genetic algorithm based approach for task assignment and scheduling on multi-FPGA embedded systems. Dick and Jha [4] presented a preemptive multi-rate scheduling algorithm that performs dynamic task reordering based upon FPGA reconfiguration time. Noguera and Badia [5] discussed a dynamic run-time scheduler for the tasks with non-deterministic execution time. Shang and Jha [6] proposed a two-dimensional, multi-rate cyclic scheduling algorithm which calculates the task priorities based upon real time execution constraints and reconfiguration overhead information. Task location is assigned based upon a number of factors that include task prefetch, pattern reutilization and fragmentation. Steiger et al. [7] presented an efficient methodology for online scheduling. We propose and non-preemptive integrated online scheduling and placement methodology. We use an efficient and fast online placement algorithm that improve placement quality as compared to previous approaches. Also, we delay the scheduling decision to accommodate dynamic task priority better then the previous approaches.

3 Motivation

There is a fundamental difference between scheduling of tasks for a FPGA and a multi-processor system. In conventional multi-processor systems, availability of a resource can be determined trivially. If no task is assigned to a resource at a particular time, the resource can be considered un-utilized and can be assigned to a task during that time. On the contrary, in case of an reconfigurable systems, a single FPGA is used to execute a number of tasks concurrently. Due to fragmentation of the area resources, there is no guarantee that resources will be available for execution of a task.

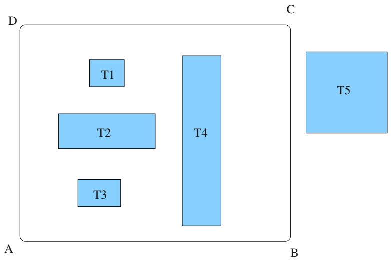


Fig. 1. Modeling the reconfigurable area

In the partially reconfigurable systems, each task uses a set of contiguous area locations for the duration of its execution time. The contiguous reconfigurable resources used by an task can be modeled by a rectangular area on the FPGA surface [8,9,7, 10]. Figure 1 shows four tasks T1-T4 modeled as rectangles placed on the FPGA represented by the rectangle ABCD.

The area on the FPGA may be fragmented due to already placed tasks. Even if the empty area on the FPGA is more than the area of the task, the area may not be contiguous. As an example, task T5 in Figure 1 can not be placed on the FPGA because enough empty contiguous area is not available to accommodate that task. Therefore, in addition to assigning a start time, the scheduler need to assign a placement location to the task. So, the scheduler for reconfigurable systems need an integrated online placement engine. If the scheduler ignores the placement requirements of the task, the task may not find a placement location and may not start execution at the start time assigned to it. Our framework uses an efficient online placement engine for task scheduling.

Speed and efficiency are very important attributes of a good online scheduling methodology. Placement and Scheduling are performed at the run-time of the applications. So, time taken by these operations is an overhead on the execution time of the application. In dynamic reconfigurable systems, no compile-time optimization can be performed. So, quality of online placement engine tends to be inherently poor as compared to the offline approaches like simulated annealing and the force-directed placement. This calls for an efficient task placement strategy that ensures least degradation in the placement quality.

Our dynamic scheduler uses a fast online placement algorithm for assigning placement location to the tasks. Our placement algorithm guarantees to find a placement location if there exists one. This leads to efficient use of the reconfigurable resources.

4 Task and Scheduler Models

In this section, we discuss the task model we use for this work. In addition, we describe our integrated scheduler model and compare it with other contemporary approaches.

A task is assumed to be rectangular is shape. Besides the functionality, the heigh and the width of a task, it is characterized by three time-related integers. Each task has an *arrival time* (a). This is the earliest time a task can start execution. Due to limited amount of resources, a task may not start execution at its arrival time. *Start time* (s) is the time of start of execution of the task. *Execution Time* (e) of a task is the time it takes to finish its execution. In addition, every task has a priority associated with it. Priority of a task denotes execution preference of a task over the other tasks. A task with higher priority is meant to be executed before other tasks with lower priority. Task priority can change dynamically.

In the previous non-preemptive task scheduling methodologies, scheduling and placement are treated as different steps. Tasks are scheduled as and when they arrive. So scheduling is performed according to the arrival time of the tasks. This scheme is shown in Figure 2.

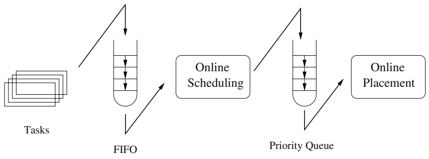


Fig. 2. Separate Scheduling and Placement

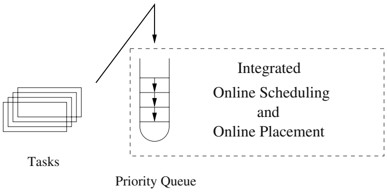


Fig. 3. Integrated Scheduling and Placement

The main problem with this scheme is that this scheme prioritizes the tasks according to task arrival time and Task priority is not respected. If a low priority task T_1 has earlier arrival time, the task is scheduled at a particular time. If a high priority task T_2 arrives later, in spite of its high priority, that task will be scheduled after T_1 because T_1 is already scheduled before the arrival of T_2. Also, once a task is scheduled, change in priority of that task will not make any effect in its execution order. Our methodology defers the scheduling decision until it is absolutely necessary. Our model of scheduler is shown in Figure 3. We perform online scheduling and placement in a single step.

5 Online Placement Algorithm

Finding and maintaining empty area on the FPGA is an important part of an online placement algorithm. We maintain empty area as a list of maximal empty rectangles [1]. Our algorithm [8] can list all the maximal empty rectangles on the FPGA by processing less than about 15% of the total number of cells. Details of our algorithm to find a list of overlapping maximal empty rectangles can be found in [8]. We choose the empty rectangle with least area for placement of a task (best fit placement strategy). In addition to being fast, our algorithm can handle dynamic addition and deletion of tasks efficiently. Number of maximal empty rectangles on the FPGA changes after addition and deletion of every task. So, the list of maximal empty rectangles needs to be updated after every addition and deletion of a task. This is performed by running our algorithm [8]. However, we do not need to run this algorithm on the whole FPGA after each addition and deletion.

The full list of the MERs is not required to be updated after addition or deletion of a new task. Let (m,n) is the top left corner of the added or deleted task and $\mathcal{L}$ be list of all

[1] A maximal empty rectangle is a empty rectangle that cannot be covered fully by any other empty rectangle

maximal empty rectangles. We define the area to the bottom right direction of (m-1,n+1) as the perturb area. perturb area for a task M1 is shown by the dotted line in Figure 4.

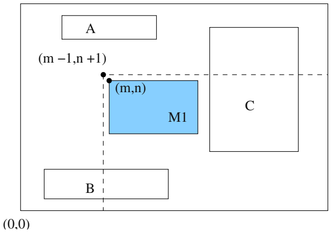


Fig. 4. Perturb Area

Note that all maximal empty rectangles that do not lie in perturb area of a task are not affected by addition or deletion of that task. All the rectangles in $\mathcal{L}$ that have their bottom right virtex in perturb area are deleted. So, in Figure 4, rectangles B and C will be deleted after addition or deletion of M1. A list of maximal empty rectangles is found in the perturb area using our algorithm discussed in [8] and added to $\mathcal{L}$.

6 Scheduling Methodology

Online placement and online scheduling are tightly integrated operations in a partially reconfigurable system. Benefits of the integrated approach is that tasks get executed in order of their dynamic priority and not according the precedence of their arrival time. We assume that the time taken for partial reconfiguration of a task is negligible as compared to the execution time of the tasks.

In our model, all tasks need to be executed on the FPGA. Further, our scheduling scheme is non-preemptive. In between their arrival time(a) and start time(s), tasks are placed inside an input priority queue. Tasks with high dynamic priority are placed at the top of the input priority queue as compared to those with low priority. Tasks with same priority are sorted according to their arrival time. Since all the tasks stay in the same priority queue before their execution starts, dynamic change in their priority is reflected in their execution order. The tasks may or may not be data dependent upon the other tasks. So, there are two possible queues: in-order and out-of-order.

6.1 In Order Processing

Tasks in the input priority queue may be data dependent upon each other. So, the tasks need to be processed in-order. In case of in-order execution, we maintain one other priority queue called the *deletion queue* in addition to the *input queue*. The *deletion queue* contains task deletion events. Events in the *deletion queue* are prioritized according to their time. If a task starts its execution, we place a corresponding task deletion event in the *deletion queue*, with the time stamp of the sum of task start time and the task execution time.

Task processing is performed by execution of task placement and task deletion events. At every time tick, we check status of both *input queue* and the *deletion queue*. If present time is equal to the time of deletion event at the head of the *deletion queue*, we delete that event from the FPGA. If there is an event to be placed in the *input event*, we try to place that event by finding a suitably large empty rectangle by searching through the list of maximal empty rectangles(MERs). The list of MERs is updated after every addition and deletion event as explained in Section 1.

We try to schedule execution of each task at its arrival time, but this is not possible due to limited area resources (Section 3). If a task $\mathcal{T}$ cannot be placed at some time, we have to delay start of execution of all the tasks below $\mathcal{T}$ because the tasks are executed in-order. In such a situation, the *input queue* is said to be locked. If the *input queue* is locked, we cannot place any other task until one of the already placed tasks finishes execution and is removed for the FPGA. Also, if placement of subsequent tasks is delayed because of queue lock, we do not want to delay the time of task deletion event from the FPGA. This goal is achieved because the *deletion queue* is separate from the *input queue*. The queue lock is released after execution of a delete event.

Our in-order processing methodology guarantees minimum delay in execution of the tasks because only the *input queue* is delayed for the time that is absolutely necessary for placement of the tasks and the deletion of tasks is not delayed.

6.2 Out of Order Processing

In out-of-order processing, the tasks are not data-dependent upon each other and can be executed in any order. This relaxation can be used to schedule tasks for better area utilization.

As in the in-order processing case, the incoming tasks are placed in a *input queue* till they are ready to start execution. In the out-of-order processing, both task addition event and task deletion events are maintained in the same queue. If an event at the top of the queue is the delete event, deletion operation is performed. If the event corresponds to an task addition event, the list of MERs is searched to find a suitable empty rectangle for task placement. If no suitable rectangle is found, the task is re-scheduled after the first deletion event in the queue. The algorithm then proceeds to the next event in the queue to see if that is a delete event or an add event. This procedure is repeated at every time tick.

In this strategy, if an task is too large to be placed on the FPGA, it is tried to be placed again after deletion of an task. Note that even second time, there may not be enough space for placement of an event. The event is rescheduled again after the first deletion event in the queue and so on.

As we will see in Section 7, this scheduling methodology results in better area utilization than the in-order processing. Also, in this case, there is no or minimal delay involved in total execution time because this methodology can use the time between delete event and next addition event to place previously unplaced tasks.

7 Results

In this section, we present results of our online scheduling and placement experiments. In first set of results, we show efficiency of our online placement algorithm as compared to a previous approach. Next, we compare both scheduling strategies in terms of FPGA area usage.

In our experiments, we generated tasks with their dimensions (length and width) generated randomly between 1 and a maximum number $\mathcal{L}$. Each task has a finite life-time. Delay between two consecutive tasks is generated randomly between two bounds. Delay-factor is defined as the ratio of maximum delay between two tasks and maximum life-time of tasks. Effectively, delay-factor serves purpose of "normalized" delay between two tasks. A high delay factor means that a set of two tasks have more delay between their start time and a low delay factor value means that a set of two tasks are very near to each other as compared to their life-times. In our experiments, tasks are generated with their delay factor between 0 and a maximum value $\mathcal{D}$. All experiments have been performed on a 96×64 FPGA. Each data point in the graphs is generated by conducting an independent placement run with 1000 macros.

7.1 Online Placement Results

In order to show effectiveness of our online placement algorithm, we compare it with another state-of-the-art online placement algorithm proposed by Bazargan et al. [9].[2] Each task is tried to be placed at its arrival time and if the task can not be placed, it is assumed rejected. We call this no-queue scheduling. Note that the improvement in online placement strategy is reflected directly as the efficiency of the dynamic scheduling methodology. We measure effectiveness of the algorithms in terms of area utilization and acceptance of the tasks. Area utilization is measured in terms of empty area per unit time (EAPT). Empty area is calculated as number of free CLBs. Task acceptance is measured as percentage of task accepted and placed by an algorithm.

Our algorithm maintain empty area as a list of maximal empty rectangles while Bazargan [9] uses heuristics to maintain empty area as non-overlapping empty rectangles. In this section, we compare our algorithm with Large Empty Rectangle (LER) heuristic as discussed in [9]. We chose this heuristic because it produced best placement quality among all the heuristics discussed in [9].

In the Figures 5, 6, 7 and 8, the graphs with legend "Non-Overlapping Empty Rectangle" are generated using Bazargan's methodology.

Figures 5 and 6 show task acceptance and EAPT as a function of maximum size $\mathcal{L}$. Delay factor $\mathcal{D}$ is kept constant at 0.05 for these experiments. As maximum size increases, tasks become larger. As a result it becomes more difficult to place those tasks on the finite FPGA surface. So, percentage of accepted tasks decreases. On the other hand, larger tasks utilize more area. So, the empty area per unit time (EAPT) decreases, showing better device utilization.

Figures 7 and 8 shows acceptance and EAPT as a function of maximum delay factor $\mathcal{D}$. Maximum size of tasks, $\mathcal{L}$ is kept fixed at 25 for these experiments. As delay between two consecutive tasks increases, percentage acceptance of tasks increases. This is because, if next tasks comes at a larger delay, there are more chances of one of the executing tasks to finish execution and release area for the incoming task. Also, the area utilization decreases (increase in EAPT) with increase in delay factor.

[2] We want to thank Dr. Kiarash Bazargan for letting us use their software [9] for comparison purposes.

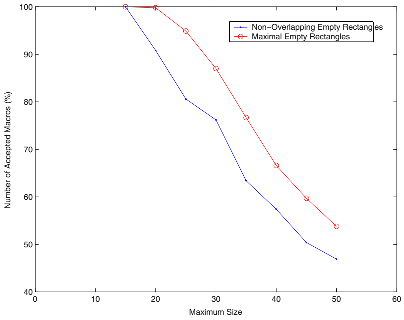


Fig. 5. Percentage Acceptance as a Function of Task Size

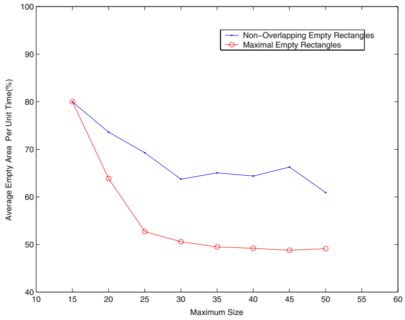


Fig. 6. Empty Area Per Unit Time as a Function of Task Size

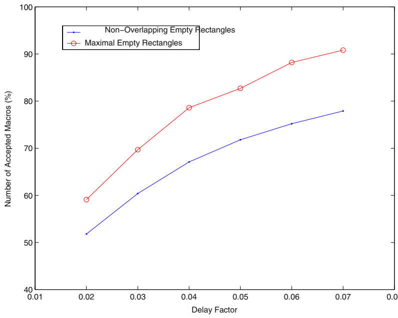


Fig. 7. Percentage Acceptance as a Function of Task Delay

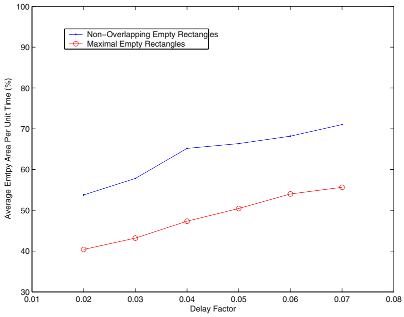


Fig. 8. Empty Area Per Unit Time as a Function of Task Delay

As shown in the Figures, our algorithm performs better than [9] in terms of area utilization and task acceptance. As shown in these figures, our algorithm has about 14% less empty area per unit time (EAPT) as compared to [9]. Due to better area utilization, our algorithm can accept about 10% more tasks than [9]. Our algorithm performs better because we maintain empty area as maximal empty rectangles. This decreases fragmentation of area resources and results in better utilization.

7.2 Scheduling Results

In this section, we present experimental results of our integrated scheduling and placement strategy. We use the same benchmarks that we used in the last section. We compare both in-order processing and out-order processing strategy with no-queue scheduling strategy as described in previous section.

Figure 9 shows empty area per unit time as a percentage of the full FPGA area. Area utilization of the out-of-order scheduling strategy is best among all. Since, in-order scheduling methodology wait for tasks to be removed to place a larger task, the area utilization is poor than the no-queue scheduling case.

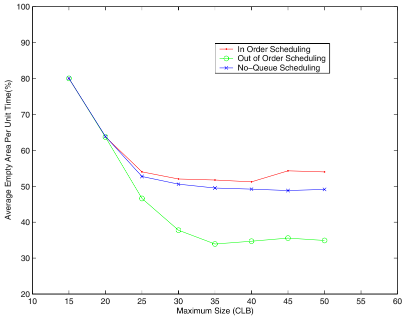


Fig. 9. Area Utilization as a Function of Max. Size

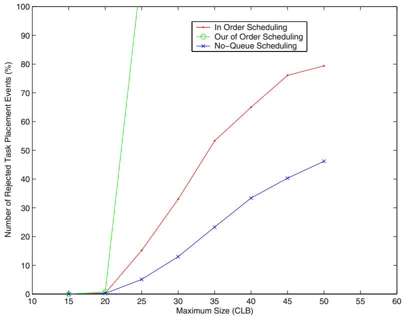


Fig. 10. Task Placement Rejection as a Function of Max. Size

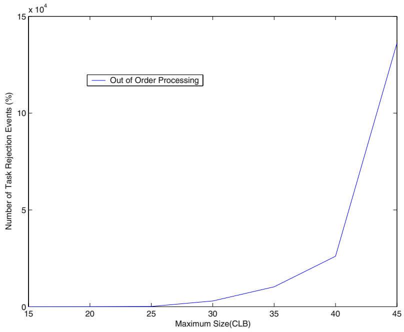


Fig. 11. Task Rejection for Out of Order Processing

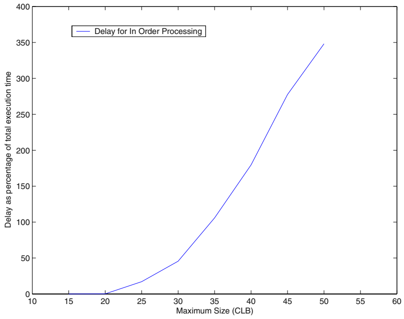


Fig. 12. Delay for In Order Processing

Figure 10 shows rejection of task placement request by the placement engine. no-queue scheduling has minimum task rejection rate because once a task is not placed, no other attempt is made to place that task. The in-order scheduling has more rejections of task placement requests. The difference between the two widens as the maximum task size increases. Rejection of the task placement requests is highest in case of out-of-order scheduling. This is because, in this case, a task may be re-scheduled multiple times due to non-availability of resources. Also, rejection of task placement request increases sharply with increase in maximum task size (as shown in Figure 11) because it is difficult to place larger tasks and these tasks are rejected multiple times.

Figure 12 shows the delay in in-order processing as a percentage of execution time of the 1000 tasks in each benchmark. The delay increase with task size because placement of larger tasks may require removal of a large number of other tasks already executing on the FPGA.

There is no delay in case of out-of-order processing as time between successive tasks is used to place previously rejected tasks. But queue maintenance overheads are very large in case of out-of-order processing as a larger number of task rejections are handled (as evident by the Figure 11).

8 Conclusion

In this paper, we propose an efficient integrated online scheduling and placement strategy. We argue that by designing proper queueing strategy, task scheduling can be simplified in the dynamically reconfigurable systems. We propose two queueing strategies for in-order and out-of-order task processing and discussed their characteristics in details.

Acknowledgments. This work is sponsored in part by the Dayton Area Graduate Studies Institute (DAGSI) and the Air Force Research Laboratory (AFRL) research program under contract number IF-UC-00-07.

References

1. Keith Underwood. FPGAs Vs. CPUs: Trends in Peak Floating-Point Performance. In *Proceedings of the Twelfth ACM International Symposium on Field-Programmable Gate Arrays*, February 2004.
2. J. Sgall. On-Line Scheduling - A Survey. In *Online Algorithms: The State of the Art,Lecture Notes in Computer Science 1442*, 1998.
3. Bingfeng Mei, Patrick Schaumont, and Serge Vernalde. A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems. In *11th ProRISC workshop on Circuits, Systems and Signal Processing Veldhoven, Netherlands*, November 2000.
4. Robert P. Dick and Niraj K. Jha. Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 62–67, 1998.
5. Juanjo Noguera and Rosa M. Badia. Dynamic run-time hw/sw scheduling techniques for reconfigurable architectures. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 205–210, 2002.
6. Li Shang and Niraj K. Jha. Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs. In *Proceedings of the 15th International Conference on VLSI Design (VLSID02)*, January 2002.
7. Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. In *24th IEEE International Real-Time Systems Symposium, Cancun, Mexico*, December 2003.
8. Manish Handa and Ranga Vemuri. An Efficient Algorithm for Finding Empty Space for Online FPGA Placement. In *Proceeding of the 41st Design Automation Conference*, June 2004.
9. Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems . In *IEEE Design and Test - Special Issue on Reconfigurable Computing*, volume 17(1), pages 68–83, Jan.-Mar. 2000.
10. Herbert Walder, Christoph Steiger, and Marco Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing . In *International Parallel and Distributed Processing Symposium (IPDPS'03)* , page 178, April 2003.

On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities

Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker

Universität Karlsruhe (TH), Germany
http://www.itiv.uni-karlsruhe.de/
{ullmann, huebner, bgrimm, becker}@itiv.uni-karlsruhe.de

Abstract. Microcontrollers and ASICs have become a dominant part during the last years in the development of embedded applications like automotive control units. As described in our previous contribution we presented an alternative approach exploiting the possibilities of partial run-time reconfiguration of state-of-the-art Xilinx Virtex FPGAs. Our approach used a run-time system software for controlling reconfiguration and message handling. This paper presents some new extensions introducing dynamic priority measures as a first approach for adaptive reconfiguration decisions.

1 Introduction

As the number of deployed engine control units (ECUs) in the automotive domain grows it will become an increasing problem for many automobile manufacturers since they complain about their handling and the growing costs of development, production and maintenance (e.g. storage of spare parts etc.). 30 years ago an automobile was equipped with only a few (analog) control devices, which could be handled by hobby car mechanics. But today the (now digital) ECU equipment has become very complex and error-prone because of the new functions interdependencies and their local distribution inside the automobile. Additionally as the number of control units and services desired by customers increases, electrical power dissipation will grow to keep all devices working.

During the last years the sector of control devices was dominated by microcontrollers and ASICs so that every function was implemented by a dedicated control unit. Today upper class automobiles contain up to 100 control units and even in inactive mode their total power consumption may discharge the battery which can result in a disabled automobile. Additionally the devices become sooner obsolete and the product life cycle decreased from 5 years down to 2 years [16].

We proposed a different approach by applying reconfigurable hardware devices which could solve partially the problems described above. Our approach is based on the assumption that not all ECU functions must be available at the same time, so it should be possible to identify an adequate subset of functions which can be operated on the same reconfigurable resource by applying a kind of flexible demand-driven time-multiplex [15]. We developed a new flexible FPGA (Xilinx Virtex II) based hardware architecture supporting partial run-time self-reconfiguration and a first

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 454–463, 2004.

version of a run-time system (RTS) which performs reconfiguration, message handling and a first rather simple resource management approach. It's obvious that not all types of ECU-functionalities can be operated on such an architecture because of redundancy and security demands so we restricted at this stage to less critical automotive comfort functions like seat adjustments, compartment lightning or power windows.

The feature of partial run-time reconfiguration might make such new FPGAs attractive for embedded systems because it enables for hardware-based task switching and there exist different academic approaches wherein they try to exploit these novel features [3], [4], [9], [10], [14]. Hardware-based task switching enables for a dynamic function multiplex at run-time where hardware mapped function-blocks can be placed at run-time on different (on-chip / off-chip) execution locations [8], [12]. Run-time task management is not a trivial job so that many approaches adopted solutions from operating system theory [6], [11], [13]. Some researchers implemented complex operating systems by adapting LINUX-based system kernels to their needs so that their systems can do a migration of software-based tasks to hardware and vice versa [7]. Although we conceive a real-time operating system integration of our run-time system at a later stage as well we believe that at present it's a more important task to optimize its run-time behavior. This paper presents some improvements concerning the run-time system's reconfiguration and demand handling of our latest approach as presented in [15]. The paper is structured as follows. In the following we give a short description on the basic hardware architecture supporting run-time reconfiguration, bitstream decompression & placement and on-chip/peripheral communication. Later we describe the most essential parts of the structure of our proposed run-time system and describe the improvements that we added to our run-time system. The paper closes with some preliminary results and a preview on our future work to be done.

2 Target Hardware Architecture

Our implemented target hardware architecture was mapped on a Xilinx XC2V3000 FPGA. As depicted in Fig. 1 it consists of a Xilinx MicroBlaze soft-core processor where the run-time system software is located on. The MicroBlaze processor is connected to an external Controller Area Network (CAN)-Controller device, which handles external bus communication. Additionally MicroBlaze is internally connected to a bus-arbiter device and a decompression/reconfiguration unit. The bus-arbiter controls the internal communication between MicroBlaze and function-modules which are located in predefined reconfiguration-slots on FPGA.

The decompressed automotive function-modules can be reloaded from external Flash-memory separately into the given module slots on FPGA by using the possibilities of internal partial run-time reconfiguration in combination with run-time decompression where the former is initiated via the internal configuration port (ICAP) [3]. The physical separation between single functions on FPGA was realized by a bus macro, which connects the signal lines between the functional blocks and the arbitration/ run-time-system module. More details about used tool-chains, our bus-macro approach and decompression unit can be found in [1], [5], [15]. As mentioned above the run-time system is implemented as static software part on a Xilinx MicroBlaze soft-core processor. Designed with Xilinx Embedded Development Kit (EDK) the run-time system is connected via the on-chip peripheral bus to peripheral functions like the decompressor module and the arbiter. This core handles the incoming and outgoing messages of the complete system. Messages from peripheral devices will be decoded, processed and sent to the bus arbiter which controls the on-chip-bus-lines connecting the currently available functions (see Fig. 1). Messages from the function-modules will be received by the arbiter and redirected via MicroBlaze to the external devices. The implemented structure is also capable of saving states and all necessary parameters if a function-module has to be substituted by another. This context-save mechanism is administrated by the run-time system but the context-data is transmitted via the bus lines and controlled by the arbiter.

The next section will give a short overview on the basic functions of the run-time system and the approach for a function request management. Details on the decision strategy of module substitution can be found in section 4.

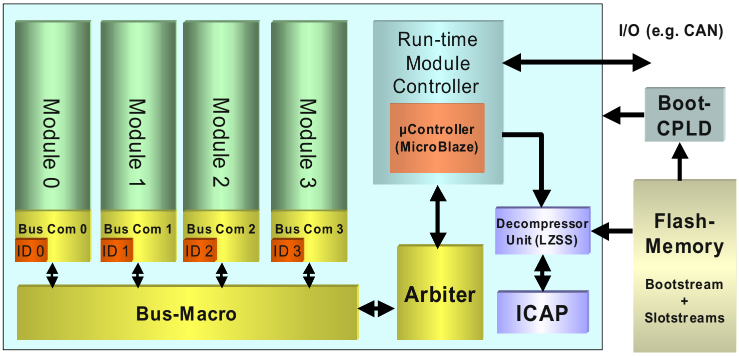


Fig. 1. Run-time system with FPGA partial run-time reconfiguration support and soft-processor

3 On Demand Run-Time System

It is the run-time system's basic job to handle time-constrained communication of the function-modules with the corresponding sensors and actuator components by translating and forwarding the messages. It is relevant to preserve the order of incoming and outgoing messages for every function-module. Furthermore the run-time system manages and controls the reconfiguration process of the function-modules and in this context it performs a resource management on the FPGA as well. The run-time system also stores the state information of the available function-modules which are currently inactive and not physically available on FPGA.

One could conceive a stand-alone conventional FPGA-system as well which offers all needed functions in parallel. But we have found that often not all functions are needed at the same time so we have the possibility for saving hardware resources by applying a reconfiguration-based demand driven multiplex which offers the same set of functions on smaller and cheaper reconfigurable devices which consume less power as well [2]. The presented system may also be reusable for different scenarios and other fields of application like home area (e.g. windows, lights, heating). To take into account the different sizes of function-modules and FPGA types the run-time system must be scalable for a changing number of function-modules and FPGA-slots (resp. FPGA sizes).

3.1 Modular Structure

The run-time system consists of four major modules:

a) *Management of incoming messages:* This part receives the incoming messages, translates them into an internal message representation and forwards them to the function-modules specific input buffers. If the dedicated function is not available on FPGA a reconfiguration request will be generated which is attempted to be processed in the next and following iteration cycles by **b)**.

b) *Reconfiguration process:* It selects a function/reconfiguration request from the reconfiguration-request-buffers for re-activation and one inactive/idle function slot on FPGA. It saves the state information of the depending function-module, initiates the reconfiguration of the FPGA's module slot to be replaced and reloads the state information of the new module into the replaced slot. More details on the reconfiguration mechanism and decompressor hardware unit used here can be found in [15].

c) *Message-buffer management unit*: It stores and forwards after reactivation messages which could not be delivered to their destination function-module-units after these have been deactivated and removed by reconfiguration.

d) *Management of outgoing messages:* This unit receives the messages from the FPGA-slot-modules by checking its corresponding buffer, translates them into external messages and initiates their transmission to the external environment (e.g. via CAN-bus).

As shown in Fig. 2 these four parts are executed sequentially in a pseudo parallel way by defined context switches between the software parts to guarantee a fast message handling. Additionally there exist interrupt triggered handling functions, which buffer all messages incoming from external (CAN-) bus and internal FPGA-bus into the run-time system's message buffers so that the latter may access and process them later when ready. For further details please see at [15].

The allocation and control of FPGA-slots and handling of incoming messages will be done by using tables. The message-module tables are needed for assigning the messages to the corresponding function-modules and for translating them into internal bus-messages. There exist two tables where the first one keeps informations about individual dedicated messages for modules and the second one keeps handling-informations on common/public messages which can involve different function-modules who need input from the same external source(s). The function-module table stores the state information of the inactive function-modules, the start- and end-addresses of the function's bitstream Flash-memory location and further module-specific informations, like the previously sent message. The slot table is a further data structure, which stores information about the currently allocated slots and its history which can be evaluated for slot selection/reconfiguration purposes. Beside these tables there exist several buffers for the communication's management between the run-time system's main parts and for buffering messages and demands of modules. Additional buffers are needed for the interrupt service routines, which store incoming messages for the run-time system arriving from the external Controller Area Network (CAN) -bus and internal FPGA-bus (via bus-arbiter).

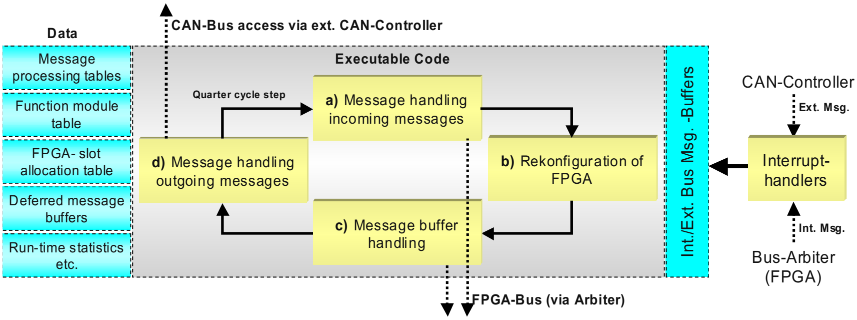

Data
Message processing tables
Function module table
FPGA- slot allocation table
Deferred message buffers
Run-time statistics etc.
CAN-Bus access via ext. CAN-Controller
Executable Code
Quarter cycle step
a) Message handling incoming messages
b) Rekonfiguration of FPGA
c) Message buffer handling
d) Message handling outgoing messages
Int./Ext. Bus Msg. -Buffers
CAN-Controller
Ext. Msg.
Interrupt-handlers
Int. Msg.
Bus-Arbiter (FPGA)
FPGA-Bus (via Arbiter)

Fig. 2. Major parts of the run-time system, showing one iteration cycle

4 On Demand Function Management

The set of targeted automotive functionalities was restricted to less critical passenger compartment functions. Although there are no critical motor-management/security functions involved (e.g. ignition, brakes etc.) other functions like windshield wiper, central locking system or garage door remote control must be immediately at disposal if they are needed because depending on the current situation they might be of importance.

As a consequence priority metrics must be introduced which help selecting the most important function from a set of function-demands to be executed. At this point it should become aware that one has different possibilities to define a priority measure. Incoming and outgoing messages may contain priority information which causes a preferred treatment at the destination side. The developed run-time system does not completely support this feature, since it differs only between common and individual messages, where common messages may have many recipients which might not be seriously affected by the message even if they are inactive and not available. Details on that topic can be found in [15]. If the run-time system would forward a prioritized individual message to a target function on FPGA whereby older queued messages are still waiting, this out-of order forwarding could cause an unpredictable function behavior since the older queued messages might be of importance to get the correct results. Additionally we need different measures for making appropriate reconfiguration decisions.

4.1 Priority Management

We decided to define at design time for each function type a static priority which can be an integer number in the range of [$0_{(low)}$, $k_{(high)}$]. We grouped similar functions or functions of same subjective importance level into a priority hierarchy. As shown in 0 we placed the burglary & anti-theft functionality into the top level where seat adjustment, power windows or sunroof adjustment were placed into the lowest priority level. Additionally it is of importance how far there are any interdependencies between functions. Some similar functions can not be used at the same time (e.g. parking aid vs. cruise control/distance control) so they are candidates to be set at the same priority level. The defined static priorities will help to find some idle/inactive function slots so that the function-slot with lowest assigned static function-priority will be selected for exchange. After defining static priorities the problem of dynamic function assignment has to be solved. One could conceive a static assignment for messages as well, so that messages with same target will receive the same priority value like their target function. As long as there are enough system resources (free available slots) and the number of requested functions is rather small there won't be large problems but static message priorities may cause that unprocessed messages of lower priority won't be served and delivered to their recipients. So as consequence static function priorities can be used for reconfiguration-slot selection but dynamic run-time priority assignments are necessary to select the proper incoming message for further processing.

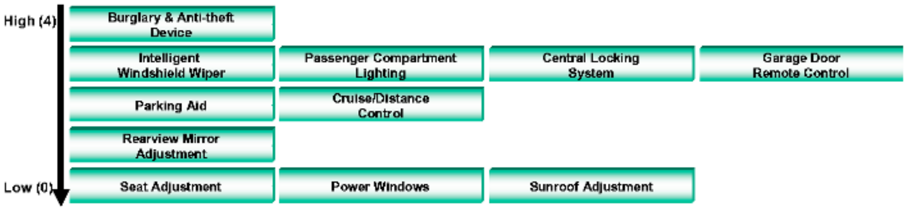

High (4)
Burglary & Anti-theft Device
Intelligent Windshield Wiper
Passenger Compartment Lighting
Central Locking System
Garage Door Remote Control
Parking Aid
Cruise/Distance Control
Rearview Mirror Adjustment
Low (0)
Seat Adjustment
Power Windows
Sunroof Adjustment

Fig. 3. Exemplary possible static priority predetermination at design-time

The order of processing requests for reconfiguration should be free of starvation. Otherwise some functions will never become active. Already active functions will receive at least one pending buffered message each, so that the order of forwarding buffered messages to all active FPGA-functions within on iteration cycle plays no role. But in case that there are several reconfiguration requests pending including some filled message buffers for each requested inactive function one has to consider response deadlines and limited resources (free buffer-memory, allocated buffer elements) as well. So we need additional metrics which give us at run-time information how to select a reconfiguration request for re-activation on FPGA. Most functions have defined response time deadlines until the desired function has to send its reaction on the received message to the external periphery. So we will need to know for every requested inactive function how many unsuccessful reconfiguration request & check cycles were performed since it was requested for activation & reconfiguration on FPGA. The number of request-repetitions can be counted for every requested function each time the run-time system repeats its resource check cycle where all active functions are checked for their idle state. Since that resource check cycle takes a nearly fixed amount of time (depending on system clock) the number of repetitions corresponds approximately to the wait-for-service-time of each function. The targeted automotive comfort functions have in general a demanded response time of about 100 ms where the user won't perceive a delay. This time can be seen as upper limit which corresponds to a certain number of run-time system iteration cycles (see Fig. 2, system timing results can be found in section 6). So one can define a set of dynamic priority measures:

Reconfiguration-request-ratio. We define this ratio as quotient of skipped reconfiguration-requests and maximum RTS -repetition cycles for a certain function-activation-request F_x where the maximum number of cycles can be derived from the function's deadline and cycle-time of the reconfiguration-task-module (see Fig. 2).

This ratio is denoted as $P_{skipped\_rec}(F_x)$ (see also equ. (1)). $P_{skipped\_rec}(F_x)$ will produce values in the range of [0, 1] where a value close to 1 means that it becomes very urgent to activate the function as soon as possible so that corresponding buffered messages can be processed.

$$P_{skipped\_rec}\left(F_x\right) = \frac{\text{Skipped\_Reconfigurations}\left(F_x\right)}{Max\_Cycles} . \qquad \textbf{(1)}$$

Communication / activity rate. Every function receives and transmits messages within a certain time period. So it is possible to define for every active function a counter which is incremented each time a dedicated message was received or sent.

$$P_{act\_rate}\left(F_x\right) = \frac{\text{RX-TX\_Messages}\left(F_x\right)}{Max\_Cycles \cdot k_{\varnothing RX\text{-}TX\,Msg}} . \qquad \textbf{(2)}$$

The number of received/sent messages can be set into relation to a fixed number of iteration cycles (e.g. Max_Cycles from equ. (2)) and the maximum number of messages per function which can be handled within one cycle by the system ($k_{\varnothing \text{RX-TXMsg}}$). We denote this ratio as $P_{act\_rate}(F_x)$ (see equ. (2)). At the current stage of development every active function receives and sends for each RTS-cycle one buffered message ($k_{\varnothing \text{RX-TXMsg}}$=2), so this ratio produces again values in the range of [0, 1]. This ratio can be used as secondary criteria for the selection from FPGA-slots containing functions of same static priority. The larger $P_{act\_rate}(F_x)$ becomes the higher the probability will be that the corresponding function will be needed in future.

Input-buffer-load ratio. Each function offered by the run-time system has its own input message buffer, where all specific incoming messages are buffered and wait for further processing. All function-buffers share the same memory which is organized as dynamic free-buffer-element-list, where allocated buffer elements are concatenated by pointers to their corresponding message buffer chain and moved back later to the free-list buffer-chain after their content was forwarded. So we can define again a dynamic ratio of used buffer-elements to the sum of used and free elements.

$$P_{buff\_load}\left(F_x\right) = \frac{\text{Used\_Message\_Bufferssize}\left(F_x\right)}{\text{Used\_Message\_Bufferssize}\left(F_x\right) + \text{Free\_Buffers}} \ . \qquad \textbf{(3)}$$

The larger $P_{buff\_load}(F_x)$ becomes the more urgent it will be to empty the buffer, otherwise other incoming messages have to be discarded.

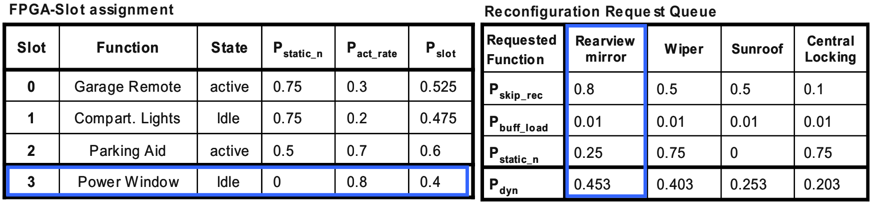
Weighted priority measure. Depending on predefined preferences it is possible to order the dynamic priorities by importance. It should be noted that measure $P_{act\_rate}(F_x)$ correlates in a complementary way with $P_{buff\_load}(F_x)$. Active functions with a rather high activity may have a lower number of buffered messages since they become more frequently processed whereas inactive/waiting functions will have a rather full input buffer. By applying the found measures one can define a dynamic priority measure as heuristic approach for activation-selection which is the weighted sum of $P_{static}(F_x)$, $P_{skipped\_rec}(F_x)$ and $P_{buff\_load}(F_x)$. The weights reflect the importance of each measure and have to be given by the designer.

$$P_{dyn}\left(F_x\right) = \alpha \cdot P_{static}\left(F_x\right) + \beta \cdot P_{skipped\_rec}\left(F_x\right) + \gamma \cdot P_{buff\_load}\left(F_x\right) \ . \qquad \textbf{(4)}$$

5 Application Scenario

This section describes an exemplary scenario which may occur during operation. As depicted in Fig. 3 (left) four functions are configured on FPGA. Where the compartment lightning control and window control are currently inactive and candidates for replacement. We assumed static priorities in the range of [0, 5] which we normalized for easier comparison by dividing through $max\{P_{static}\}$. Additionally we introduced $P_{slot}=½\cdot(P_{static\_n}+P_{act\_rate})$ for selecting the slot with lowest priority. As result slot 3 is selected here for reconfiguration (see Fig. 3).

As next step all queued reconfiguration requests have to be checked. For the calculation of P_{dyn} we chose as weights α=0.2, β=0.5, γ=0.3. Fig. 3 shows the results for a set of four requested function-reconfigurations of different static priority. The results demand to select the Rear-view Mirror function since it produced the highest priority P_{dyn}. Although Wiper and Central-Locking functions have a higher static priority it is more urgent to select the Rear-view Mirror function since its deadline becomes closer because of its larger number of skipped reconfigurations. One should note that depending on the weights which relate to the designer's subjective preferences, others results are possible.



FPGA-Slot assignment

Slot	Function	State	P_{static_n}	P_{act_rate}	P_{slot}
0	Garage Remote	active	0.75	0.3	0.525
1	Compart. Lights	Idle	0.75	0.2	0.475
2	Parking Aid	active	0.5	0.7	0.6
3	Power Window	Idle	0	0.8	0.4

Reconfiguration Request Queue

Requested Function	Rearview mirror	Wiper	Sunroof	Central Locking
P_{skip_rec}	0.8	0.5	0.5	0.1
P_{buff_load}	0.01	0.01	0.01	0.01
P_{static_n}	0.25	0.75	0	0.75
P_{dyn}	0.453	0.403	0.253	0.203

Fig. 3. Application scenario with dynamic selection for reconfiguration (α=0.2, β=0.5, γ=0.3)

6 Conclusions and Outlook

At the current stage the described hardware architecture and run-time system are already operational. Details on synthesis and implementation results (code sizes etc.) can be found in [5], [15]. At the moment we are adapting our run-time system approach concerning the dynamic decision criteria as described in section 4. So detailed results on performance and its real-time-capabilities are not available at this moment. We have tested our system with different function request scenarios and found an average response time (without partial reconfiguration) below 1 ms and reconfiguration time for every slot of about 15 ms (at 66 MHz system clock). The system proved to be able to handle a set of complex functions like seat adjustments, compartment lightning and power windows, whereas a pure microcontroller reference-implementation (25 MHz system clock) was swamped with the load caused by the processing and additional CAN-bus communication handling of only a subset of all functions. We will have to evaluate the system's behavior on real-time constraints and its performance. Our current results are promising and we intend to improve the system's set of features e.g. we think about a self optimization were the system can find the weighting parameters at run-time for given example scenarios.

References

1. Becker, J., Huebner, M., Ullmann, M.: Real-Time Dynamically Run-Time Reconfiguration for Power-/Cost-optimized Virtex FPGA Realizations. Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-SoC), Darmstadt, Germany, December 1-3 2003, 129 – 134.
2. Becker, J., Hübner, M., Ullmann, M., "Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations", 16th Symposium on Integrated Circuit Design and System Design (SBCCI 2003), Sao Paulo, BRAZIL, September 2003.
3. Blodget, B., McMillan, S., Lysaght, P. : A Lightweight Approach for Embedded Reconfiguration of FPGAs. Design, Automation and Test in Europe Conference and Exhibition, Munich, March 3-7, 2003, 399 – 400.
4. Horta, E. L., Lockwood, J. W., Taylor, D. E., Parlour, D.: Applications of reconfigurable computing: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. Proceedings of the 39th conference on Design automation, New Orleans, June 2002, 343 – 348.

5. Hübner, M., Ullmann, M., Weissel, F., Becker, J.: Real-time Configuration Code Decompression for Dynamic FPGA Self-Reconfiguration. Proceedings of the 11th Reconfigurable Architectures Workshop (RAW/IPDPS) 2004, Santa Fé, New Mexico, USA, April 2004.
6. Kearney, D., Wigley, G.: The first real operating system for reconfigurable computers. Proceedings of the 6th Australasian Computer Systems Architecture Conference 2001, ACSAC 2001, Gold Coast, Queensland, Australia, 29-30 Jan. 2001, 130 –137.
7. Kearney, D, Wigley, G..: The management of applications for reconfigurable computing using an operating system. Australian Computer Science Communications, Proceedings of the seventh Asia-Pacific conference on Computer systems architecture - Volume 6, Volume 24 Issue 3, Melbourne, Australia, January 2002, 73 – 81.
8. Levinson, L., Manner, R., Sessler, M., Simmler, H.: Preemptive Multitasking on FPGAs. IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, California, 17-19 April 2000, 301 –302.
9. Marescaux, T., Bartic, A., Verkest, D., Vernalde, S., Lauwereins, R.: Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. Proceedings of the Field-Programmable Logic and Applications (FPL) 2002 Montpellier, France, September 2002, 795 – 805.
10. Mignolet, J-Y., Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. Proceedings of Design, Automation and Test in Europe (DATE) Conference, Munich, Germany, March 2003, 986 – 991.
11. Mignolet, J-Y., Vernalde, S., Verkest, D., Lauwereins, R.: Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. Proceedings of the International Conference Engineering Reconfigurable Systems and Architecture 2002, Las Vegas, USA, June 2002, 116 –122.
12. Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Designing an operating system for a heterogeneous reconfigurable SoC. Proceedings of the International Parallel and Distributed Processing Symposium 2003, Nice-France, April 22-26, 2003, 174 –180.
13. Nollet, V., Mignolet, J-Y., Bartic, T.A., Verkest, D., Vernalde, S., Lauwereins, R.: Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems. Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms 2003, Las Vegas, June 2003, 81 –187.
14. Haug, G., Rosenstiel, W.: Reconfigurable hardware as shared resource for parallel threads. Proceedings IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April 15-17, 1998, 320 –321.
15. Ullmann, M., Hübner, M., Grimm, B., Becker, J. : An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. Proceedings of the 11th Reconfigurable Architectures Workshop (RAW/IPDPS) 2004, Santa Fé, New Mexico, USA, April 2004.
16. VDI report vol. 1547, 9th International Congress on Electronics in Automotives, Baden Baden, Germany, October 2000.

Techniques for Virtual Hardware on a Dynamically Reconfigurable Processor – An Approach to Tough Cases –

Hideharu Amano[1], Takeshi Inuo[2], Hirokazu Kami[2],
Taro Fujii[3], and Masayasu Suzuki[1]

[1] Dept. of Information and Computer Science, Keio University
[2] NEC Silicons Devices Research Laboratories, NEC Corporation
[3] NEC Electronics Corporation

Abstract. Virtual hardware is difficult to implement even on recent dynamically reconfigurable processors when the loop body of the target application cannot be stored in the set of quickly switch-able contexts. Here, techniques for such tough cases are proposed. Differential configuration which changes only different parts of similar contexts can drastically reduce the time for re-configuration. Pairwise context assignment policy can hide the overhead of configuration with double buffering. Out-of-order context switching enables execution of available context in advance. Through an implementation example on NEC's DRP-1, it appears that the virtual hardware can be executed with practical speed by combining the proposed techniques.

1 Introduction

Virtual hardware is a small scale reconfigurable system which executes a large scale hardware by dynamically changing its configuration[1][7]. Since the time for re-configuration of common FPGAs takes a few milliseconds, it has been difficult to build a practical virtual hardware except for the logic emulators used for design verification. Recent dynamically reconfigurable processors with high speed re-configuration enable practical virtual hardware that can replace a certain size of process dynamically during execution[2][4][5][6]. Many parts of the stream processing application consist of a sequence of tasks, and by replacing such tasks on demand, the total job can be executed with a small amount of hardware[9]. However, even with such dynamically reconfigurable processors, the performance is severely degraded by the time for context replacement if (1) the size of the single task is beyond the hardware context or (2) the set of contexts cannot be replaced with a clock.

Here, such a tough case for executing a large single task with a small scale dynamically reconfigurable processor is covered. Three techniques: "differential configuration", "pairwise assignment" and "out-of order context scheduling" are proposed for this purpose, and a simple application is implemented on the NEC's dynamically reconfigurable processor DRP-1 as an example.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 464–473, 2004.

2 Virtual Hardware

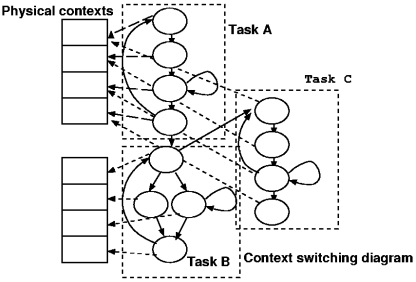
2.1 Task-Level Virtual Hardware

Reconfiguration methods used in recent dynamically reconfigurable processors are classified into two categories: (1) quick configuration delivery from on-chip configuration memory [5][6] and (2) selecting a contexts from the on-chip multiple contexts[1][11][3][2][4].

The former type called "configuration delivery" holds a large number of configuration sets in the on-chip configuration memory, and delivers configuration to processing elements and interconnects with dedicated buses. The time for re-configuration is a few microseconds, faster than common FPGAs which often require a few milliseconds. Such devices are useful for task or loop level virtual hardware. For example, when multiple encryption methods are used in a protocol, the decryption hardware can be replaced dynamically according to the header information. Although the service is suspended for a few microseconds for re-configuration, most of the applications will allow it if the system provides enough amount of packet buffer. This style of virtual hardware is a realistic approach using such devices.

On the contrary, the latter type called "multicontext" switches the contexts in one clock, but the number of physical contexts is limited. For example, NEC's DRP-1[2] provides sixteen contexts, and IPFlex's DNA[4] provides four. In such a device, a single task is usually implemented with multiple contexts for area-efficient implementation by making the best use of quick context switching capability. When the number of contexts is not enough, the configuration data can be loaded onto the unused contexts from on-chip or off-chip configuration memory during execution. Using such multicontext reconfigurable devices, virtual hardware mechanisms have been researched from early 1990's[1][12][14][13]. The results of WASMII project[1] suggested that the configuration overhead of task level or loop level virtual hardware can be hidden by providing two sets of hardware contexts.

Fig. 1 is a context switching diagram which describes this concept. If a task A including a loop with a certain number of iteration can be stored in a set of contexts, the configuration data corresponding to the next task B can be loaded to the unused context set during execution. After finishing the task A, task B can start without any overhead if the execution time of task A is longer than that of configuration loading time for task B. While task B is being executed, the configuration data corresponding to the next task C can be loaded to the area which was used for the task A. This "double-buffer" policy works well if the context number of each task is not beyond each context set, and the next task for execution can be predictable. Fortunately, most stream processing application satisfies these conditions, and the task-level or loop-level virtual hardware without any overhead is quite realistic.



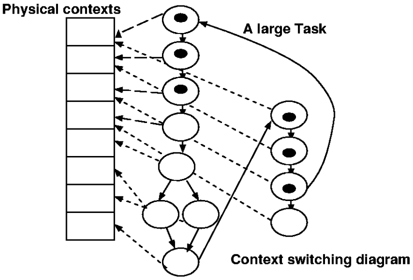


Fig. 1. Task-level virtual hardware

Fig. 2. Inner loop virtual hardware

2.2 Inner Loop Virtual Hardware: The Tough Case

However, the task level virtual hardware is realistic only when the target task or loop can be stored in a certain number of context set which can be changed in a clock. That is, a large task consisting of a long processing sequence and complicated branches between contexts cannot be executed with such a method. Fig. 2 describes such a situation. Since the loading time of configuration data (t_{load}) is much longer than the interval of context switching (t_{ivl}) in the diagram shown in Fig. 2, the execution time becomes almost $t_{load} \times m$, where m is the number of contexts replaced in a loop (marked context in Fig. 2.). t_{load} takes a few microseconds. On the contrary, the context switching interval without loop is usually several clock cycles, and the execution clock of reconfigurable processors is at least 10's MHz. That is, the execution time is more than 100 times slower than the case with enough number of contexts.

In order to reduce the configuration time, compression methods of configuration data [15] have been researched. However, our previous research[16] revealed that the compression cannot drastically reduce the configuration data of the coarse grain architecture.

3 Target Model: NEC's DRP-1

Although the methods proposed here can be applied to most multicontext reconfigurable devices, we introduce NEC's DRP-1 (Dynamically Reconfigurable Processor)[2] for practical discussion. Note that, for inner-loop virtual hardware, some additional mechanisms that are not realized in the current DRP-1 must be assumed.

3.1 DRP Overview

DRP is a coarse-grain reconfigurable processor core which can be integrated into ASICs and SOCs. The primitive unit of DRP Core is called a 'Tile', and DRP Core consists of arbitrary number of Tiles. The number of Tiles can be expandable, horizontally and vertically.

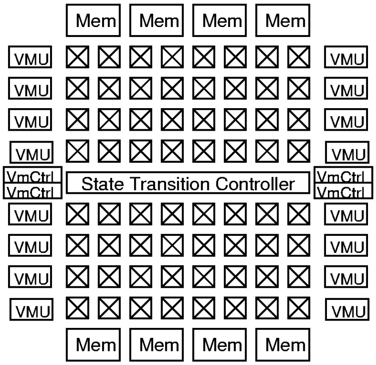


Fig. 3. Structure of a Tile

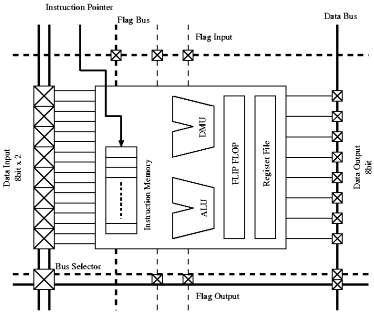


Fig. 4. Structure of a PE

The primitive modules of Tile are processing elements (PEs), State Transition Controller (STC), 2-ported memories (VMEMs: Vertical MEMories), VMEM Controller (VMCtrl) and 1-ported memories (HMEMs: Horizontal MEMories). The structure of a Tile is shown in Fig. 3.

There are 8×8 PEs located in one Tile. The architecture of PE is shown in Fig. 4. It has an 8-bit ALU, an 8-bit DMU, an 8-bit×16-word register file and an 8-bit flip-flop. These units are connected by programmable wires specified by instruction data, and their bit-widths range from 8Bytes to 18Bytes depending on the location. PE has 16-depth instruction memories and supports multiple context operation. Its instruction pointer is delivered from STC.

STC is a programmable sequencer in which certain FSM (Finite State Machine) can be stored. STC has 64 states, and each state is associated with the instruction pointer. FSM of STC operates synchronized with the internal clock, and generates the instruction pointer for each clock cycle according to the state. Also, STC can receive event signals from PEs to branch conditionally. The maximum number of branch is four.

As for the memory units, a Tile has eight 2-ported VMEMs on its right and left sides, and four 1-ported HMEMs on upper and lower boundary. The capacity of a VMEM is 8-bit×256-word, and four VMEMs can be handled as a FIFO, using VMCtrl. HMEM is a single-ported memory and it has a larger capacity than the VMEM. It has 8-bit×8K-word entries. Contents of these memories, flip-flops, register files of PE are shared with the datapath of all the contexts.

DRP Core, consisting of several Tiles, can change its contexts every cycle by instruction pointer distribution from STCs. Also, each STC can run independently, by programming different FSMs.

DRP-1 is the prototype chip, using DRP Core with 4×2 Tiles. It is fabricated with 0.15-um 8-metal layer CMOS process. It consists of 8-Tile DRP Core, eight 32-bit multipliers, an external SRAM controller, a PCI interface and 256-bit I/Os. The maximum operation frequency is 100-MHz. Although DRP-1 can be used as a stand-alone reconfigurable device, Tiles of DRP can be used as an IP (Intellectual Property) on ASIC with an embedded processor. In this case, a number of Tiles can be chosen so as to achieve the required performance with minimum area.

3.2 DRP Configuration

Although DRP-1 can be configured in a number of ways, the most commonly used method is configuration through the PCI interface. Configuration data corresponding to PEs in all Tiles, switches, memory modules and other interfaces is mapped into a single logical address with 20bit address and 32bit data. It can be accessed as a simple byte-addressed 32bit memory from the host, and the configuration data is transferred through PCI bus usually in the burst mode.

According to one evaluation[16], loading configuration data for a context requires at least 1,000 clocks, or 15μsec using the 66MHz PCI bus.

4 Techniques for Inner Loop Virtual Hardware

4.1 Differential Configuration

A large scale application often includes contexts which are similar to each other. For example, in the inverse MDCT implemention on the DRP-1[17], context 4,5 and 11, context 12, 13 and 14, context 9 and 10 are almost the same structure, respectively. In the discrete system simulator shown later, a large number of contexts can be classified into some groups whose member has a similar structure. The difference between such structures comes from the following reasons:

- The location and accessing address of the distributed memory VMEM.
- Parameters or constant numbers used in calculations.
- Structural difference for exceptional operations.

Let us assume that A_1, A_2 and A_3 are similar, and a common part (A) and different parts (a_1, a_2, and a_3) can be divided as $A_1 = A + a_1$, $A_2 = A + a_2$ and $A_3 = A + a_3$. By only replacing the configuration data for different parts (a_1, a_2 and a_3), the structure can be therefore changed. This way of configuration is called "differential configuration". Here, A_1, A_2 and A_3 form a context group.

For differential configuration, the specialized mode is required to the configuration mechanism of the current DRP. That is, in the differential configuration mode, the bit '1' in the loading configuration data reverses the corresponding data bit of the target context memory. That is, the bit '1' in the loading data changes '0' in the current configuration into '1', and '1' into '0'. Using this mechanism, only different part of the configuration data is changed without changing the common part.

4.2 Pairwise Assignment Policy for Differential Configuration

Differential configuration reduces the configuration time drastically if the common part is always in the context memory. Therefore, the double buffer policy for

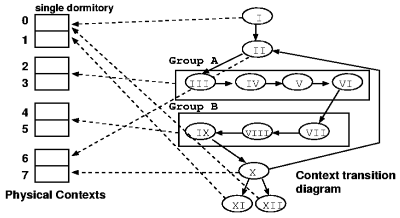


Fig. 5. An example of assignment

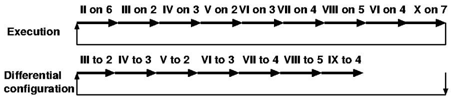


Fig. 6. Timing chart for the example

a task-level virtual hardware must be applied in order to save the common part in the context memory as possible. For this purpose, we propose a context assignment policy called "pairwise assignment" for allocating logical contexts into a physical context for differential configuration. The basic steps are as follows.

1. Assign each context group into a pair of physical contexts.
2. If physical contexts remain, logical contexts in the loop structure are selected and assigned into physical contexts until a pair of physical contexts remain. Here, the logical context with longer loading time should be selected with higher priority.
3. The last pair of physical contexts called *single dormitory* are used for all remaining unallocated logical contexts.

Fig. 5 illustrates an assignment example. Contexts in group A and B are assigned into each pair of contexts. Then, context II and X are assigned into their own physical contexts because they are iteratively executed. Other contexts I, XI and XII are assigned into *single dormitory.* The context diagram shown in Fig. 5 works as shown in Fig. 6 after the initialization. In the figure, each context is asumed to be executed several clocks and differential configuration can be also done with the similar number of clocks. In this case, the loop can be iterated without overhead by using context pair as the double buffer.

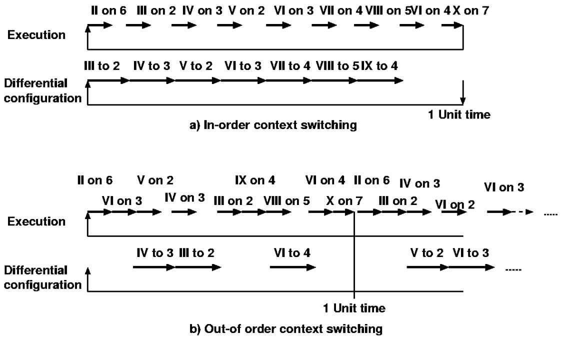
4.3 Out-of-Order Context Switching

In the virtual hardware proposed in WASMII[1], the context switching is dynamically controlled in a data-driven manner. However, this method is difficult to implement[18] because of the hardware overhead for detecting the executable context. In most applications, the state transition control mechanism used in DRP is advantageous compared with data driven control.

However, for the inner-loop virtual hardware, out-of-order context switching is sometimes advantageous for executing contexts in the current context memory. Assume that the differential configuration time is longer than the execution time of a context in group A (Fig. 5) and contexts in group A can be executed in arbitrary order. If the execution order of contexts in group A is fixed, every state transition causes a stall to wait for the differential configuration loading (Fig. 7a). On the contrary, if contexts in the group can be executed in the reverse order, the stall can be reduced by executing the current available contexts first (Fig. 7b). In this case, context VI and V in group A, and context IX and VIII in group B are loaded by the previous iteration, and they do not have to be loaded in this iteration. That is, this technique is useful for reducing the number of re-configuration.

Unlike the heavy overhead for complete data driven operation[18], this mechanism can be implemented with a small additional hardware by limiting the out-of-order context switching in the following case. That is, the execution order can be changed only:

- between contexts in a group, and
- when current available contexts can be executed in advance.



a) In-order context switching

b) Out-of order context switching

Fig. 7. Timing chart when differential configuration takes longer time than execution

5 An Implementation Example: Discrete System Simulator

5.1 Discrete System Simulator on the DRP-1

As an example of large scale applications, a discrete system simulator (DSS) is implemented on the DRP-1. DSS represents a queuing model consisting of several queues, call generators and servers linked together. In such a model, a call is generated by a source according to some distribution probability, and distinguished according to its type. Calls move around the network, and branch or join at junctions. The simulator is equipped with a component library corresponding to calls, servers, branch junctions and joining junctions which can form a queuing network. Direct mapping with a single clock execution policy proposed in [19] is used, and the target system represented with the library is executed directly on a reconfigurable system.

As a target of the simulator, a switch connected parallel machine shown in Fig. 8 is adopted. Four call generators (CGENs) which represent processing units are connected with four servers which correspond to memory modules (SRVs)

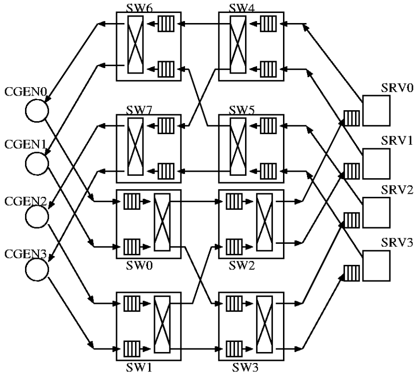


Fig. 8. Target parallel machine of DSS

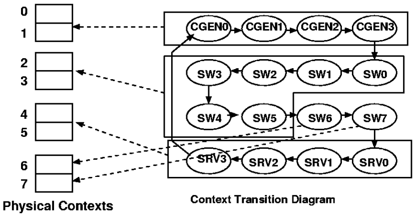


Fig. 9. Context assignment

with 2-stage interconnection networks (SWs). CGEN provides a random number generator, state controller, and an output queue with four entries. It changes its state according to the parameter and the generated random number, and issues write or read requests to SRVs. SW provides input queues and a switch consisting of branch/join junctions. It receives requests from CGENs, and transfers them to SWs in the next stage or SRVs. If conflicts occur, a request is selected, and the other is enqueued. SRVs receives requests through the input queue, and returns the data for read request after a fixed interval.

In this implementation, each component of CGEN, SW and SRV is implemented in its own context. Thus, a system shown in Fig. 8 consists of sixteen contexts. The queues and state of each component is allocated into VMEMs on DRP-1. Each component reads VMEM for checking the queue entry, obtains the random number, manages the queue, and decides the next state. For the operations, 4 or 8 clocks are required. CGENs, SWs and SRVs form their context group, thus, the DSS consists of three context groups. Table 1 shows the maximum delay (Delay: nsec) and required clock cycles for the operation (That is, the context switching interval: t_{itv}) of each component. This table shows that the maximum operating frequency of the DSS is 44MHz.

The difference of structure in a context group is mainly caused by the position of VMEMs and parameters. That is, the configuration data for interconnect between VMEMs and logics in the common part is the main source of the difference. The difference of words corresponding to required clocks for differential reconfiguration(D-Reconf) is also shown in Table 1.

Table 1. Specification of each group

Component	Delay (nsec)	t_{itv} (clocks)	D-Reconf (clocks)
CGEN	22.5	4	6
SW	19.7	8	45
SVR	19.7	8	35

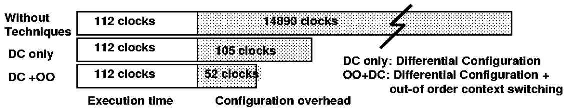


Fig. 10. Clock cycles for 1 DSS unit time

5.2 Inner-Loop Virtual Hardware for the Discrete System Simulator

Although the simulator corresponding to Fig. 8 with sixteen contexts is available on DRP-1, the inner-loop virtual hardware requires extended mechanisms that are not provided in the current DRP-1. So, the model shown in Fig. 8 is assumed to be executed on an extended DRP model with only eight contexts but one that provides the required functions for inner-loop virtual hardware. The execution clock cycles are also calculated.

The state transition diagram for context control of the discrete system simulator is shown in Fig. 9. According to the pairwise context assignment, three pairs of physical contexts are assigned into the context group for CGENs, SWs and SRVs, respectively; six contexts are thus used for this purpose. In this case, the remaining two are assigned into SW0 and SW1 as shown in Fig. 9, since SWs require the longest time for the differential reconfiguration.

Fig. 10 shows required clock cycles for a DSS unit time of DSS. Since an enormous number of iteration is required for this kind of DSS, the influence of the initialization is omitted. Here, the DSS is assumed to work at its maximum speed (44MHz) while the configuration data is loaded with double frequency, or 88MHz. A dedicated external memory is assumed for storing reconfiguration data, and the overhead by the PCI protocol is omitted. Fig. 10 shows that the overhead of reconfiguration is not negligible only with differential configuration (DC only), since the configuration time is larger than the execution time even when double frequency loading clock is used. The out-of-order context switching (DC+OO) reduces the overhead by executing current available contexts in advance, and improves performance by 50%. However, if the DSS is executed by only eight contexts without the virtual hardware support mechanisms proposed here, a large overhead (14,090 clocks) is suffered (Without Techniques). Therefore, the proposed mechanism can drastically reduce the overhead so that the inner-loop virtual hardware can be implemented to work at a practical execution time.

6 Conclusion

Techniques for inner-loop virtual hardware, a tough case of virtual hardware implementation, are proposed and discussed. Through the implementation example on NEC's DRP-1, it appears that the virtual hardware can be executed with practical speed by combining the proposed techniques. The DSS implemented here is not a typical application of dynamic reconfigurable processors. We will implement practical stream processing application for demonstrating the efficiency of the techniques proposed here in the near future.

References

1. X.-P. Ling, H. Amano. “WASMII: A Data Driven Computer on a Virtual Hardware,” Proc. of FCCM, pp. 33–42, 1993.
2. M.Motomura. “A Dynamically Reconfigurable Processor Architecture,” Microprocessor Forum, Oct. 2002.
3. T. Fujii, et al. “A Dynamically Reconfigurable Logic Engine with a Multi-Context/ Multi-Mode Unified-Cell Architecture,” Proc. of Intl. Solid-State Circuits Conf., pp.360–361, 1999.
4. http://www.ipflex.co.jp
5. http://www.pactcorp.com
6. http://www.elixent.com
7. E.Caspi, et al. “Stream Computations Organized for Reconfigurable Execution (SCORE),” Proc. of FPL’00, pp. 605–614, 2000.
8. P.Master. “The Age of Adaptive Computing Is Here,” Proc. of FPL, pp.1-3, 2002.
9. G.J.M.Smit, P.J.M.Havinga, L.T.Smit, P.M.Heysters. “Dynamic Reconfiguration in Mobile Systems,” Proc. of FPL’02, pp.162-170, 2002.
10. E.L.Horta, J.W.Lockwood, D.Partour. “Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration,” Proc. of DAC’02, 2002.
11. S. Trimberger, D. Carberry, A. Johnson, J. Wong. “A Time-Multiplexed FPGA,” Proc. of FCCM, pp.22-28, 1997.
12. N.Kaneko, H.Amano. “A General Hardware Design Model for Multicontext FPGAs,” Proc. of FPL, pp.1037–1047, 2002.
13. R.Enzler, C.Plessl, M.Platzner. “Virtualzing Hardware with Multi-context Reconfigurable Arrays,” Proc. of FPL’03, pp.151-160, 2003.
14. D.Lehn, et al. “Evaluation of Rapid Context Switching on a CSRC Device,” Proc. on International Conference on Engineering of Reconfigurable Ssytems and Algorithms, 2002.
15. Z.Li, S.Hauck. “Configuration Compression for Virtex FPGA,” Proc. of FCCM, pp.143-154, 2001.
16. T.Kitaoka, H.Amano, K.Anjo. “Reducing the Configuration Loading Time of a Coarse Grain Multicontext Reconfigurable Device,” Proc. of FPL’03, 2003.
17. Y.Yamada, et al. “Core Processor/Multicontext Device Co-design,” Proc. on Cool Chips VI, pp.82, 2003.
18. Y.Shibata, et al. “A Virtual Hardware System on a Dynamically Reconfigurable Logic Device,” Proc. of FCCM, pp.295-296, 2000.
19. O.Yamamoto, et al. “A Reconfigurable Stochastic Model Simulator for Analysis of Parallel Systems,” Proc. of FCCM, pp.291-292, 2000.

Throughput and Reconfiguration Time Trade-Offs: From Static to Dynamic Reconfiguration in Dedicated Image Filters

Marcos R. Boschetti[1], Sergio Bampi[1], and Ivan S. Silva[2]

[1] Informatics Institute, Federal University of Rio Grande do Sul, C.P. 15064, 91501-970, Porto Alegre, Brazil

[2] Informatics and Applied Mathematics Department, Federal University of Rio Grande do Norte, Natal, Brazil

Abstract. This paper analyzes the evolution of DRIP image processor from a statically reconfigurable design to a fast dynamic approach. The focus is in the overhead introduced by new programmable modules needed for RTR support and in the methodology used to redefine the basic processor elements. DRIP can perform a huge set of digital image processing algorithms with real-time performance, attending the requirements of contemporary complex applications.

1 Introduction

Recently, several implementations of complex CSoCs (Configurable Systems on Chip) have been proposed and realized. This approach presents an attractive solution to the exponential increase of CMOS mask costs and to the fast changes in algorithms and standards of contemporary applications. Flexibility is now an essential issue for the commercial success of a digital system. New mobile telecommunication systems (3G), as an example, will have to be prepared to provide the processing rates demanded by complex algorithms and to offer the required adaptability to support new services emerged after product release [1].

Run-Time Reconfigurable (RTR) systems, in special, play an important role in this scene. Dynamic reconfiguration allows the implementation of virtual hardware strategies, making it possible even the development of circuits with more logic resources than are physically available. Moreover, RTR architectures can adapt to the instantaneous needs of an application. This means more efficiency, once we have an extra opportunity to obtain specialization during the execution of the task. In the dynamic reconfiguration approach the designer can achieve outstanding benefits from the custom computing paradigm by creating mechanisms to hide the reconfiguration latency with the overlap of computation and configuration.

However, all this power comes with a cost. It's necessary to build control logic capable to manage all the necessary data transfers and the entire reconfiguration process. It's important to analyze the design parameters and understand the effects of the overhead introduced by the run-time reconfiguration.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 474–483, 2004.

In this paper we analyze the impacts of dynamic reconfiguration in the reconfigurable neighborhood processor DRIP (Dynamically Reconfigurable Image Processor) [2], presenting a first statically reconfigurable design and the steps taken to offer the hardware support for run-time reconfiguration (DRIP-RTR). DRIP can efficiently implement low-level image processing algorithms. It's application domain includes filtering tasks for multimedia operations that require heavy real-time processing. Image analysis and machine vision solutions such as robotic, security, medical imaging, and scene inspection. All these computing problems have regular characteristics and an inherent level of parallelism that can be exploited by the reconfigurable system. Complete infrastructure for relocating tasks in hardware in a run-time environment for a complete SoC is presented in [3], specifically targeting image processing applications as a first demonstrator

This work is organized as follows. Section 2 discusses important concepts about dynamic reconfiguration presenting possible reconfiguration models. Section 3 presents the general processor structure and the processing elements of the statically reconfigurable DRIP. It also describes the enhancements made in the basic cells for the implementation of the dynamic reconfiguration mechanisms, presenting the methodology used to generate the RTR design. Section 4 give performance results achieved for important image processing algorithms, showing the overheads, advantages and disadvantages between the statically and dynamically reconfigurable approaches. Conclusion and future work are drawn in Section 5.

2 Dynamic Reconfiguration Strategies

A clean and unified taxonomy for reconfigurable architectures (RA) classification has not emerged from the literature. The active and fast-moving applications field for RA have led to some confusion as to what dynamic reconfiguration really means.

A possible division introduces three different classes. A static design exists when the circuit has only one configuration and it never changes. In a statically reconfigurable design the circuit has several configurations, but the reconfiguration process occurs only at the end of each processing task. Finally, in a dynamically (run-time) reconfigurable design the reconfiguration takes place during the algorithm execution [2].

Run-time reconfiguration allows the designer to better exploit the flexibility of reconfigurable architectures. Besides minimizing the reconfiguration overheads of statically reconfigurable architectures, dynamic reconfiguration brings the possibility to a better exploration of the design space with virtual hardware strategies (as mentioned previously) and stream computations. The SCORE (Stream Computations Organized for Reconfigurable Execution) [4] employs this approach, a program can be seen as a graph of computation nodes and memory blocks linked together by streams, new operators are loaded in the hardware as demanded by the execution flow.

2.1 Reconfiguration Models

To efficiently implement dynamic reconfiguration designs, different reconfiguration models have been proposed and can be found in the literature [5]. Dynamic partial reconfiguration, multicontext and pipelined reconfiguration are possible approaches.

Partial dynamic reconfiguration is related to the redefinition of parts of a circuit mapped to a reconfigurable fabric through modifications in the configuration bitstream. Static parts of the array may continue execution, what leads to the overlap of computation with reconfiguration, hiding some of the reconfiguration latency.

Xilinx Virtex FPGAs family [6] is among the programmable devices that allow the implementation of this model. Its internal structure is divided in columns that correspond to an entire vertical slice of the chip. It is possible to perform transformations in the logic of a dynamic module mapped to one of these columns through modifications in the configuration file. As an example in [7] the Recats architecture is presented. Recats is a run-time reconfigurable ATM switch that uses partial reconfiguration to dynamically download pre-compiled hardware modules to a Virtex FPGA.

A potential disadvantage of the partial model is that address information must be supplied with configuration data, increasing the total amount of bytes transferred. To avoid considerable overheads, compression techniques can be used. On of these techniques is wildcarding hardware [6], which provides a method to program multiple logic cells with a single address and data value.

In the multicontext dynamic reconfiguration model, a context represents a possible configuration of the device. In this environment, there are multiple programming bits for a given configurable resource and these memory bits can be seen as different configuration planes. The reconfiguration controller can switch between the different contexts according to the application needs.

In the Chameleon reconfigurable communication processor [8] there are two configuration planes. The active configuration controls the fabric, and the background plane is used to hold another configuration. The controller can load a new configuration into the background plane while the fabric is running with the active plan. The entire system can be reconfigured in one clock cycle. Other systems exploring similar styles can be found in [9] and [10]. A drawback of multicontext reconfiguration is the amount of bytes necessary to store each configuration, what can make prohibitive a large number of different planes in hardware reducing the overall flexibility. Multicontext dynamic reconfiguration is the model followed by DRIP-RTR as will be seen in section 3.

Finally, we have the pipelined dynamic reconfiguration model. Through dynamic reconfiguration a pipeline structure implemented in hardware can support more stages than are physically available. In a first step the pipeline stage is adapted to compute part of the application, in the sequence, the processing effectively occurs. This happens in the PipeRench [11] and makes performing the computation possible even if the entire configuration is never present in the fabric at the same time.

3 DRIP Architecture

The DRIP structure is inspired in the functional programming (FP) paradigm [12]. A great advantage of this approach is the ability to express complex problems with simple basic functions, allowing the efficient implementation of a large number of image processing algorithms.

3.1 Reconfigurable Datapath

The datapath is composed by a bidimensional (9x9) matrix of processing elements (PEs). The structure of the pipeline follows a data flow graph represented by a class of non-linear filters widely used in digital image processing [13]. The hardware implementation of this filter is based on a parallel sorting network, more specifically on the odd-even transposition sort algorithm [14], which achieves a good trade-off between parallelism, regularity and execution time. Figure 1 presents a high level view of the datapath.

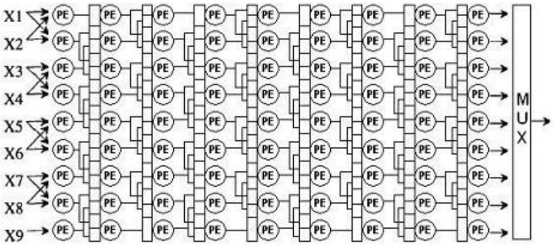


Fig. 1. Dapath overview

3.2 Processor Elements

The processor element is the basic programming block of the processor. In spite of being a simple cell, provides all the flexibility needed to implement entire classes of digital image processing algorithms. A processor element can execute only two basic operations: MAX representing the class of non-linear operations and ADD representing the class of linear algorithms. Each PE receives two input pixels and to increase its logical capabilities a restrict integer weight (-1, 0 or 1) is associated to each input.

Datapath (re)configuration consists in the customization of the PE network. According to the mentioned parameters we are allowed to apply 18 different configurations to a single PE. However, many of them are symmetrical, for example MAX(X1*1,X2*0) is the same as MAX(X1*0,X2*1) what defines a set of 11 really distinct functions, each one corresponding to one row of table 1.

In the statically reconfigurable DRIP, the PE is a super-specialized cell. Each PE is mapped to only one of the functions of Table 1 and can not be reprogrammed. The reconfiguration strategy demands the end of the processing task

Table 1. Possible PE configurations

Configuration	Function
Add(0,0); Max(0,0)	0
Add(0,X); Add(X,0)	X
Add(-X,0); Add(0,-X)	-X
Add(X1,X2)	addition
Add(-X1,X2); Add(X1,-X2)	subtraction
Add(-X1,-X2)	-X1 - X2
Max(0,X2); Max(X1,0)	If X1(2) > 0 then X1(2) else 0
Max(0,-X2); Max(-X1,0)	If X < 0 then X else 0
Max(X1,X2)	Max(X1,X2)
Max(-X1,X2);Max(X1,-X2)	If X1(2) > X2(1) then X1(2)
Max(-X1,-X2)	-Min(X1,X2)

in order to rebuild the processor elements in the bit level. The loading of the new configuration introduces considerable reconfiguration overhead.

The complete design flow is managed by our CAD system called VDR (Visual Interface for Dynamic Reconfiguration [15]). It receives the algorithm definition provided by the designer and optimizes it generating a VHDL model as the final result. As this process is repeated, a configuration library is formed and these different algorithms can be mapped to the reconfigurable fabric as needed, but always after the end of the processing task.

3.3 Datapath Support for RTR

Considering this design flow, a new level of programmability for RTR support is required. Therefore, DRIP processor element received new hardware elements and control signals increasing the PE grain size and including the necessary flexibility for dynamic algorithm switching. Besides programmability, larger grain size can bring other benefits such as less routing complexity [16]. An important point is that the term granularity, in this work, is being used to define the complexity of a cell. Thus, an increase in the granularity of a PE means that it has more hardware resources than before, however, it still processes 8-bit pixels. Figure 2 shows a new RTR PE. Some extra circuit blocks (hardware overhead) were introduced like the necessary multiplexers.

Analyzing successive algorithms mapped on the datapath it is important to recognize that the less modifications needed from one configuration to another the less reconfiguration time overhead. Besides, all possible algorithms are composed by only 11 different basic PE functions, what results in a considerable PE reuse.

In reconfigurable architectures based on partially reconfigurable FPGAs great part of the design effort is concentrated in reducing the amount of hardware to be reconfigured. In this context, three different types of blocks can be defined: static, dynamic and semi-static. Static blocks remain unchanged between

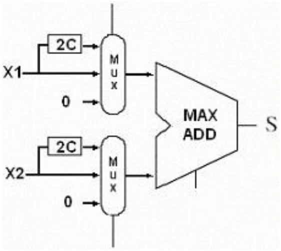


Fig. 2. DRIP-RTR PEs

configurations. Dynamic cells are the ones that change completely or almost completely, while semi-static cells present very few structural differences.

These concepts can be extended to the processor element network. Analyzing the different digital image processing algorithms mapped on DRIP datapath, it is readily seen that a significant similarity level exists, what reflects in a huge number of static and semi-static PEs. Therefore, we can map only the really needed elements to the reconfigurable fabric reducing resources usage significatively and, at the same time, minimizing the logic depth of the cells what leads to higher operational frequencies. DRIP specific CAD tool performs similarity analysis through a topological comparison between target algorithms. As result an optimized VHDL model for the application is generated in design time. To minimize the overhead for RTR, only the necessary resources are generated during synthesis stage.

3.4 Complete DRIP-RTR Architecture

A neighborhood processor simulates an array processor. It processes an input image, generating a new image, where each output pixel is an image function of its correspondent in the input image and the nearest neighbors. Using a standard neighborhood (e.g.: 3x3, 5x5 or 7x7 pixels), it scans the image line by line. The general architecture of the RTR processor can be seen in figure 3.

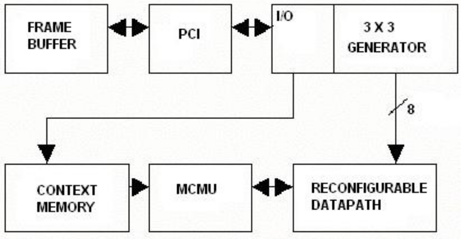


Fig. 3. DRIP-RTR architecture

The I/O processor is responsible for the communication with the host system. It comprises the neighborhood generator that provides 3x3 pixel windows to the datapath. Moreover, the I/O processor receives new configurations to be stored in the MCMU. Each configuration is a very compact (405 bits only) representation of a whole program that can be loaded in the reconfigurable fabric.

The multicontext management unit (MCMU) is the responsible for the reconfiguration process. It includes the context buffers where complete datapath configurations are stored. An entire program requires 405 bits, divided in 9 slices of 45 bits to configure each of the 9 pipeline stages of DRIP-RTR. Each slice contains the configuration of an individual column. The replacement of a configuration can be performed in parallel or in a pipelined fashion. In parallel mode the configuration data is transferred from a context buffer to the datapath in one clock period. In pipelined mode the reconfiguration simulates the data flow, changing at each clock cycle the configuration of a single column. The MCMU is prepared to receive new configurations from the I/O processor. This allows the inclusion of new algorithms extending the processor functionalities. In this situation the context buffers can be seen as a background configuration plane that can store new algorithms while the current active program is still running.

4 Statically Reconfigurable Versus RTR

As explained previously, DRIP can perform a large number of image processing operations. Some examples include: linear (convolution), non-linear and hybrid filters, binary and gray-level morphological operations (dilation, erosion, thinning and thickening algorithms, morphological edge detectors, gradient), binary and gray-level geodesic operations, etc. Table 2 shows the maximum datapath frequency achieved for five of these algorithms using two different Altera FPGA families.

Table 2. Maximum pipeline frequency

Algorithm	**FLEX10K** (MHz)	**APEX20K** (MHz)
Median Filter	32.89	77.05
Morph. Edge Detector	48.78	110.45
Erosion	39.84	78.74
Dilation	44.12	84.18
Separable Median Filter	46.30	105.9

These numbers assure real-time performance even for significant image resolutions. The parallelism and regularity of the datapath and the characteristics of image processing applications make DRIP well suited for low-level multimedia tasks.

Figure 4 shows the frame throughput (in frames/sec) for 3 different situations. The morphological edge detector and the median filter represent, respectively, the fastest and the slowest configurations when considering only one algorithm mapped to the datapath at a given time, as in the statically reconfigurable approach. The inferior line in figure 4 (triangle style line) represents DRIP-RTR, in a configuration that supports faster reconfiguration, in order to run up to 5 different image processing algorithms. All implementations have considered the DRIP hardware processing in VGA (640 x 480), SVGA (800 x 600 and 1024 x 768), and SXVGA (2048 x 1536) image resolutions.

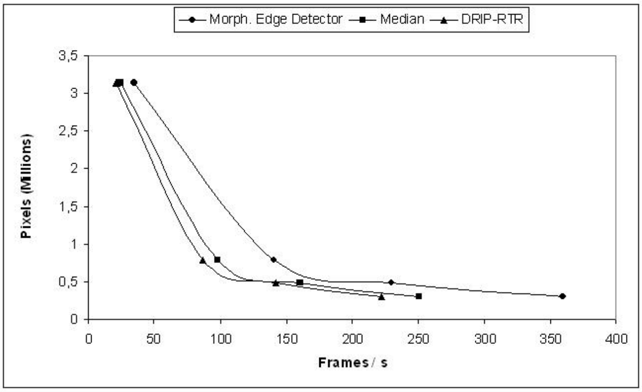


Fig. 4. Pixels versus frames/s for some image resolutions

As said before, the values in table 2 were achieved considering a datapath implementation that can support only the synthesized algorithm (statically reconfigurable), it is a maximum performance table. Due to the overheads naturally added when allowing run-time reconfiguration, a datapath implementation where it is possible to perform RTR reconfiguration between the five algorithms of table 2 needs 1830 Altera logic cells. For this DRIP-RTR pipeline the maximum frequency is 68,2 MHz.

Table 3. DRIP and DRIP-RTR speedup over a general purpose processor

Algorithm	Speed-up SR DRIP	Speed-up DRIP-RTR	Image Size
Dilation	0,94	0,76	256 x 256
Dilation	3,06	2,4	512 x 512
Dilation	17,4	14,15	1024 x 1024
Erosion	11,55	9,4	256 x 256
Erosion	45,9	37,2	512 x 512
Erosion	206,4	167,3	1024 x 1024

The area overhead of DRIP-RTR with respect to the statically reconfigurable version is about 65%. This increase is the price paid for the extra programmability introduced by control logic and on-chip configuration contexts, but it is not a serious penalty once the total RTR implementation still fits well in modern programmable devices.

Table 3 presents a comparison between two digital image processing algorithms running in three different architectures. The base platform for this comparison is a Sun Blade station with 512Mb memory and running at 650 MHz (UltraSparc II processor). The algorithms were developed in C language, using a scientific image processing library called DipLib [17]. As the image size increases, the larger is the speedup achieved. The statically reconfigurable DRIP, again, is faster than the RTR version.

4.1 Reconfiguration Time

As can be seen in Figure 4 and tables 2 and 3, DRIP-RTR is slower than the datapath implementations that can support only one algorithm. However, the great advantage of the RTR approach is based on the reconfiguration time. FPGA devices, even in parallel configuration mode, take dozens and even hundreds of milliseconds for a complete reconfiguration. Partially reconfigurable FPGAs can provide faster reconfiguration. However, the support for a great number of dynamic modules is still restricted and the configuration latency keeps on being prohibitive for high-performance architectures.

Thanks to the modifications introduced in the DRIP-RTR, it can be completely reconfigured in one clock cycle (15 nanoseconds). This is a very large speedup of about 10^7 (million times) with respect to the reconfiguration time of the entire device in the statically reconfigurable design.

5 Conclusions

Reconfigurable architectures are extending products life cycles through real time in-system debugging, field-maintenance and remote upgrades. This flexibility comes together with characteristics of special purpose designs such as high performance and low power consumption. In this context, we presented the evolution of the DRIP image processor. From a statically reconfigurable design a new flexible run-time reconfigurable system was developed. Increasing the processor basic building block grain size, we added the extra hardware to support a new level of programmability. Our CAD tool is used to determine the right amount of hardware needed by each PE in the reconfigurable datapath. DRIP-RTR achieves a significant performance with very little reconfiguration latency. The MCMU can efficiently handle the reconfiguration process and is also suited to support the extension to new algorithms. This can be done for data-flow oriented tasks that allow for highly parallel and highly reconfigurable datapaths. The results showed that the datapath and control overheads introduced in the development of the run-time reconfigurable strategy is a price worth paying. There is a 11% difference in the performance in favor of the statically reconfigurable approach (considering its worst case), however, due to the RTR version programmability and the on-chip context buffers, it can be reconfigured in one clock cycle. Hence, million times faster than reprogramming the entire FPGA. Considering the development of configurable systems-on-chip, DRIP-RTR can be an efficient core specialized in general image filtering applications.

References

1. Becker, J., Pionteck, T., Glesner, M.: An application-tailored dynamically reconfigurable hardware architecture for digital baseband processing. In: XII Symposium on Integrated Circuits and System Design (SBCCI), Manaus, Brazil (2000)
2. Adário, A.M.S., Roehe, E.L., Bampi, S.: Dynamically reconfigurable architecture for image processor applications. In: Proceedings of the 1999 Design Automation Conference (DAC), New Orleans, USA (1999) 623–628
3. Mignolet, J.Y., Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In: Design, Automation and Test in Europe (DATE 2003), Proceedings, Munique, Alemanha (2003) 986–991
4. Dehon, A., Caspi, E., Chu, M., Huang, R., Yeh, J., Markovsky, Y., Wawrzynek, J.: Stream computations organized for reconfigurable execution (score): Introduction and tutorial. In: Proceedings of Field-Programmable Logic and Applications (FPL). (2000)
5. Comptom, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. In: ACM Computing Surveys, Vol. 34. NO. 2. (2002) 171–210
6. Xilinx, I.: Xilinx virtex datasheets (2003) <http://www.xilinx.com>.
7. Horta, E., Kofugi, S.: A run-time reconfigurable atm switch. In: RAW2002 9th - Reconfigurable Architectures Workshop Proceeding, Fort Lauderdale, USA (2002)
8. Salefski, B., Caglar, L.: Re-configurable computing in wireless. In: Design Automation Conference (DAC), Proceeding, Las Vegas, USA (2001)
9. Cardoso, J.M.P., Weinhardt, M.: From c programs to the configure execute model. In: Design, Automation and Test in Europe (Date 2003), Proceedings, Munique, Alemanha (2003) 576–581
10. Dehon, A.: Dpga-coupled mocroprocessors: Commodity ics for the early 21st century. In: FPGAs for Custom Computing Machines, Proceedings. (1994) 31–33
11. Schmit, H.e.a.: Piperench: A virtualized programmable datapath in 0.18 micron technology. In: IEEE Custom Integrated Circuits Conference (CICC). (2002)
12. Backus, J.: Can programming be liberated from the von neumann style? a functional style and its algebra of programs. In: Comm. Of the ACM, Vol. 21, NO. 8. (1978) 613–641
13. Leite, N.J., Barros, M.A.: A highly reconfigurable neighborhood processor based on functional programming. In: IEEE International Conference on Image Processing, Proceedings. (1994) 659–663
14. Knuth, D.E. In: The Art of Computer Programming. Addison-Wesley (1973)
15. Boschetti, M.R., Adario, A.M.S., Silva, I.S., Bampi, S.: Techniques and mechanisms for dynamic reconfiguration in an image processor. In: Integrated Circuits and Systems Design Symposium, SBCCI2002, Porto Alegre, Brazil (2002)
16. Hartenstein, R.: A decade of reconfigurable computing: a visionary retrospective. In: Proc. International Conference on Design Automation and Testing in Europe 2001 (DATE2001), Munique, Alemanha (2001)
17. Van Vliet, L.J.: Diplib - the delft image processing library (2002) <http://www.ph.tn.tudelft.nl/DIPlib/>.

Over 10Gbps String Matching Mechanism for Multi-stream Packet Scanning Systems

Yutaka Sugawara, Mary Inaba, and Kei Hiraki

Department of Computer Science,
Graduate School of Information Science and Technology,
University of Tokyo, Tokyo, Japan
{sugawara, mary, hiraki}@is.s.u-tokyo.ac.jp

Abstract. In this paper, we propose a string matching method for high-speed multi-stream packet scanning on FPGA. Our algorithm is capable of lightweight switching between streams, and enables easy implementation of multi-stream scanners. Furthermore, our method also enables high throughput. Using Xilinx XC2V6000-6 FPGA, we achieved 32Gbps for a 1000 characters rule set, and 14Gbps for a 2000 characters one. Rules can be updated by reconfiguration, and we implemented a converter that from given rules automatically generates the matching unit.

1 Introduction

Recent technology has realized fast networks such as 10Gbit Ethernet and OC-768. To take advantage of such high-speed networks, it is necessary to accelerate network applications. Part of the applications requires intensive packet payload scanning. For example, in network intrusion detection systems (NIDS), payload scanning to find suspicious patterns occupies the major part of the computation time[1]. Other examples include Content-based billing, SPAM filtering, and HTTP request distribution. To accelerate them, fast string matching is necessary. In the matching, a set of byte strings (*rules*) is statically given, and occurrences of the strings in input packet streams are checked.

When matching speed alone is important, hardware implementation is better than software implementation. However, it is also necessary to change pattern rules. For example, in NIDS, rules are frequently updated to cope with the patterns found in new intrusion methods. Therefore, we cannot hard-wire rules because the rule update is impossible without replacing the hardware. FPGA based implementation is a solution for string matching in hardware while allowing the rules to be changed. When FPGAs are used, the rules can be changed by reconfiguration. There are studies of string matching using FPGAs such as comparator based methods[2][3], CAM based methods[4], non-deterministic automaton(NFA) based methods[5][6][7], deterministic finite automaton(DFA) based methods[8], Bloom Filter based methods[9], and Knuth-Moris-Pratt(KMP) algorithm based methods[10]. They enabled high-speed string matching while allowing the rules to be changed. For a high throughput under a limited clock speed, it is important to process multiple input characters at once[2][3][7].

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 484–493, 2004.

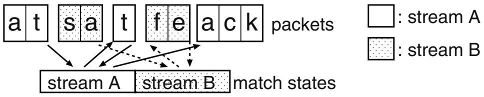


Fig. 1. An example of match state switching

In TCP, a communication data stream is split into packets. Therefore, a target pattern may span multiple packets. Such fragmented pattern cannot be discovered by a per-packet scan. This is a serious problem in applications such as NIDS that require a complete scan. To avoid the problem, it is necessary to scan TCP streams. However, existing FPGA based methods cannot be used for scanning multiple streams because of the difficulty in switching between streams.

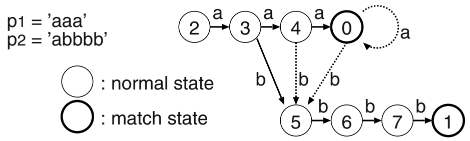
In this paper, we propose a string matching method called suffix based traversing (SBT). SBT is an extension of the Aho-Corasick algorithm[11], that uses table lookup in state transition. Since a small number of state bits are used in the SBT method, lightweight stream switching is enabled. The main points of extension for SBT are (1)processing multiple input bytes at once, and (2)table size reduction. When the Aho-Corasick algorithm is naively extended to process multiple bytes, a large lookup table is necessary. We solved this problem by reducing the number of table input patterns. As a result, the large table is converted into smaller tables and over 10Gbps scanning is enabled. In addition, we implemented a converter that automatically generates VHDL files of SBT matchers from given rules. Therefore, the rules can be updated easily.

In Section 2, we explain requirements for string matchers. Section 3 describes the SBT method. In Section 4, we present an implementation of SBT on FPGAs. In Section 5, we evaluate the SBT method using a Xilinx XC2V6000 FPGA. Section 6 presents related works, and we conclude this paper in Section 7.

2 Requirement for String Matchers

Per-stream String Matching. When a per-packet scanner is used, a target pattern is overlooked when it is split into multiple packets as shown in Figure 1. In the figure, a per-packet scanner cannot discover the pattern "attack" because it is split into three packets. This problem is serious especially in security applications because crackers intentionally split packets to evade scanners[12]. To discover such fragmented patterns, TCP-stream-level matching is necessary. To scan each stream separately using one matching unit, the match states must be swapped appropriately as shown in Figure 1. Note that if multiple scan units are used to support multiple streams, either the size of rules or the bandwidth per stream is reduced.

With existing pattern matching methods, $O(n)$ bits have to be saved at each switching[3][6][8], or the last k characters of each stream must be saved and re-scanned when it is restored[4]; where n is the total length of rules and k is the length of the longest rule. This reduces the performance and complicates the implementation for practical rules because their n and k are



state	\0	.....	a	b	c	.....	\xff
0('aaa')			0	5			
1('abbbb')							
2('')							
3('a')			4	5			
4('aa')			0	5			
5('ab')				6			
6('abb')				7			
7('abbb')				1			
default	2		3	2	2		2

Fig. 2. Example of trie and lookup table used in the Aho-Corasick Algorithm

large. For example, Snort[13] uses rules of $n \geq 1000$ and $k \geq 100$. Therefore, an alternative method is necessary to enable lightweight switching.

Processing Multiple Bytes in One Clock. To match strings at high speed, it is necessary to increase the number of input bytes processed in one clock. For example, when a string matcher processes one byte in one clock, it must operate at 1.25GHz to realize 10Gbps, which is impossible for current FPGAs. Furthermore, in future, the number of parallel bytes must be increased because network speeds are growing faster than clock speeds of FPGAs. Note that using stream-level parallelism is not a substantial solution because bandwidth depends on the number of active TCP streams.

3 SBT – Trie Based String Matching

3.1 The Aho-Corasick Algorithm

In the Aho-Corasick algorithm [11], a trie is built that contains all the rule byte strings $p_1, \cdots, p_n$. Then, an automaton is generated whose state is the node position of the trie that corresponds to the longest pattern matched to the input byte sequence up to this time. Each time a byte is received, the next state is calculated using a lookup table that receives the current state and the byte as input. When a match state is reached, the algorithm reports that the match has occurred. Figure 2 shows an example of the trie and the lookup table. In the example, match states are numbered first because it is necessary in the SBT algorithm.

3.2 Processing Multiple Bytes at Once

In a naive implementation of Aho-Corasick, the lookup table tells the state transition when one character is received. Therefore, k sequential lookups are necessary to calculate state after k characters. As a result, it is impossible to process multiple input characters at once. On the other hand, in our method, we use lookup tables which tells the state transition when k *characters* are received. Therefore, we can calculate state change after multiple characters by single table lookup. In our method, w bytes are processed at once, where w is a power of 2.

Specifically, we use lookup tables $NS_1, NS_2, NS_4, \cdots, NS_w$ for state calculations. A table NS_k tells the state transition when k byte(s) are received. NS_1 receives a byte as an input. However, $NS_k (k \geq 2)$ receives an *identification*

NS2

state \ suffix	aa	ab	bb	*a	**
0('aaa')	0		6		
1('abbbb')					
2('')					
3('a')	0		6		
4('aa')	0		6		
5('ab')			7		
6('abb')			1		
7('abbb')					
default	4	5	2	3	2

NS4

state \ suffix	abbb	bbbb	*aaa	*abb	**aa	**ab	***a	****
0('aaa')		1						
1('abbbb')								
2('')								
3('a')		1						
4('aa')		1						
5('ab')								
6('abb')								
7('abbb')								
default	7	2	0	6	4	5	3	2

Fig. 3. NS_k table examples when $w = 4$

number (IN) of the suffix which characterizes the input pattern rather than the original bytes. This is because the table becomes too large when raw byte patterns are use for the lookups. For example, NS_4 tables must hold the elements for all the 256^4 raw byte patterns.

In the static preparation, for each NS_k, a set of the necessary suffixes are identified from the rules, and INs are assigned to them. Each time a k–bytes pattern is received, the IN of its longest suffix in the set is calculated, and used for the lookup. Figure 3 shows NS_2 and NS_4 for the rules in Figure 2. In the NS_4 table of Figure 3, when a pattern $'\mathtt{bbab}'$ is received, its longest suffix is $' * *\mathtt{ab}'$. Therefore, the new state is $5('\mathtt{ab}')$ regardless of the old state.

Using the tables, state is calculated by a circuit similar to the right-hand side (state manager) of Figure 5. The signal S holds $r(0)$, the state at a 4-bytes boundary of the input stream. Each time a 4 bytes chunk is input, S is updated using the NS_4. At the same time, $r(1)$–$r(3)$, the states at $(4n+1)$th–$(4n+3)$th bytes, are calculated from S using NS_1 and NS_2. Since we number the match states of the trie first, the match flags $mt(*)$ can be calculated using comparators. Delay registers are necessary to output all $r(*)$ and $mt(*)$ in the same cycle.

We denote the input suffix set for NS_k as I_k, which is calculated statically. It is a set that consists of all the k bytes substrings of $p_i (\in P)$, and all the prefixes of $p_i (\in P)$ that are shorter than k bytes. In the example in Figure 2, the suffix sets and an IN assignment example are $I_2 = \{0\colon '', 1\colon '\mathtt{a}', 2\colon '\mathtt{aa}', 3\colon '\mathtt{ab}', 4\colon '\mathtt{bb}'\}$, and $I_4 = \{0\colon '', 1\colon '\mathtt{a}', 2\colon '\mathtt{aa}', 3\colon '\mathtt{ab}', 4\colon '\mathtt{aaa}', 5\colon '\mathtt{abb}', 6\colon '\mathtt{abbb}', 7\colon '\mathtt{bbbb}'\}$.

At run time, lookup tables $C_2, C_4, \cdots, C_w$ are used for longest suffix calculation. C_2 returns the IN of the longest suffix of a 2-bytes string. Other C_k return the IN of the longest suffix of a concatenation of two suffixes of up to $k/2$ bytes. Figure 4 shows C_2 and C_4 for the rules in Figure 2. The IN of the longest suffix of a w-bytes input is calculated from those of its sub-patterns as shown in left-hand side (suffix calculator) of Figure 5. In the Figure, $m(2,0)$ and $m(2,2)$ are the INs of the longest suffixes of the 2 bytes sub-patterns, and $m(4,0)$ is the IN of the longest suffix of the 4 bytes input.

3.3 Formal Algorithm Description

We denote the w input bytes as a w bytes string p_{in}. Let P be the set consisting of all the rule strings p_i. The character indices of strings start from 0. The length of the string p is written as $|p|$. We write the concatenation of strings p and p' as $p :: p'$. $\mathtt{substr}(p, i, j)$ is the j characters substring of p starting from position i, C2

1st \ 2nd	\0	...	a	b	...	\xff
\0						
...						
a			2(’aa’)	3(’ab’)		
b				4(’bb’)		
...						
\xff						
default	0(”)		1(’a’)	0(”)		0(”)

C4

1st \ 2nd	0(”)	1(’a’)	2(’aa’)	3(’ab’)	4(’bb’)
0(”)					
1(’a’)			4(’aaa’)		5(’abb’)
2(’aa’)			4(’aaa’)		5(’abb’)
3(’ab’)					6(’abbb’)
4(’bb’)					7(’bbbb’)
default	0(”)	1(’a’)	2(’aa’)	3(’ab’)	0(”)

Fig. 4. C_k table examples when $w = 4$

and $\texttt{prefix}(p, k) = \texttt{substr}(p, 0, k)$. We write $p \preceq p'$ when the string p is p' itself or a suffix of p'. $\texttt{IN}(X, p)$ is the IN of the longest string p' in X which satisfies $p' \preceq p$. The correctness of the algorithm is proved in [14].

Variables and Tables. $T = \{p \,|\, p \preceq p_i (p_i \in P)\}$.

$$I_k = \begin{cases} \text{all the byte characters} & \text{when } k = 1 \\ \bigcup_{i,j,l(l<k)} (\texttt{substr}(p_i, j, k) \cup \texttt{prefix}(p_i, l)) & \text{when } k = 2, 4, 8, \cdots, w \end{cases}$$

for $k = 2, 4, 8, \cdots, w$ and $p, p' \in I_{k/2}$,

$$C_k(\texttt{IN}(I_{k/2}, p), \texttt{IN}(I_{k/2}, p')) = \begin{cases} \texttt{IN}(I_k, p') & \text{when } |p'| < k/2 \\ \texttt{IN}(I_k, p :: p') & \text{when } |p'| = k/2 \end{cases}$$

for $k = 1, 2, 4 \cdots, w$ and $i = 0, k, 2k, \cdots, w - k$ and $p \in I_k$ and $s \in T$,

$$m(k, i) = \begin{cases} \texttt{IN}(I_1, \texttt{substr}(p_{in}, i, 1)) & \text{when } k = 1 \\ C_k(m(k/2, i), m(k/2, i + k/2)) & \text{when } k \geq 2 \end{cases}$$

$$NS_k(\texttt{IN}(T, s), \texttt{IN}(I_k, p)) = \begin{cases} \texttt{IN}(T, p) & \text{when } |p| < k \\ \texttt{IN}(T, s :: p)) & \text{when } |p| = k \end{cases}$$

T is the nodes of the rule string trie. I_k is the input suffixes for NS_k table. Table C_k tells the IN of the longest suffix of given 2 suffixes' concatenation. $m(k, i)$ is $\texttt{IN}(I_k, \texttt{substr}(p_{in}, i, k))$, the IN of the longest suffix of a portion of p_{in}. Table NS_k tells the state change after k characters.

Static Preparation. Make T, I_k from given P and assign INs. The INs of the elements in I_1 are their character codes. In the IN assignment of T, the elements which match some p_i (i.e. $\{p \mid p \in T \text{ and } p_i \preceq p \text{ for some } p_i\}$) are numbered first, and then the other elements are numbered, in increasing order. Let u be the maximum IN of the elements in T which match some p_i. Then, calculate lookup tables C_k and NS_k.

Input. In each cycle, w *characters of stream data* are input. Let $p_{in}(t)$ be the w input characters of the t-th clock cycle.

Output. Let $q_i = p_{in}(0) :: \cdots :: p_{in}(t-1) :: \texttt{prefix}(p_{in}(t), i)$. In each cycle, for each q_i, the following are output: (1) $r(i)$: $\texttt{IN}(T, q_i)$, i.e. *the IN of the longest string in T that matches q_i*, and (2) $mt(i)$: A *match flag* which indicates whether q_i matches some p_j.

Algorithm Procedure. Let S be a state variable which holds $\texttt{IN}(T, p_{in}(0) :: \cdots :: p_{in}(t-1))$. In each clock, do following using lookup tables:

1. Calculate each $m(k, i)$ from the input $p_{in}(t)$.
2. Calculate $r(i)$ $(i = 0, ..., w-1)$ recursively: $r(0) = S$,
   $r(k(2i + 1)) = NS_k(r(2ki), m(k, 2ki)) \qquad k = 1, 2, 4, \cdots, w$
3. $mt(i) = \mathit{true}$ if $r(i) \leq u$. Otherwise, $mt(i) = \mathit{false}$.
4. Assign the next state $NS_w(S, m(w, 0))$ to S

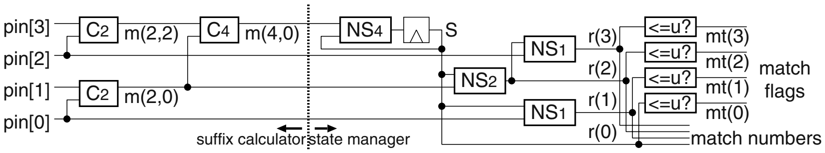
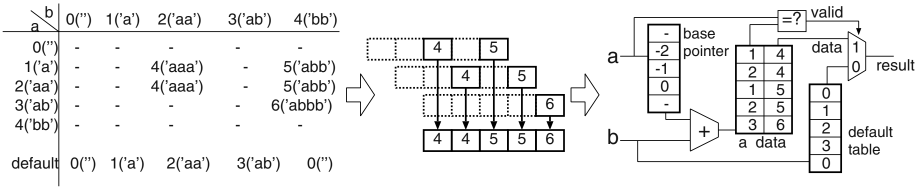


Fig. 5. Structure of the pattern matcher when $w = 4$



a \ b	0('')	1('a')	2('aa')	3('ab')	4('bb')
0('')	-	-	-	-	-
1('a')	-	-	4('aaa')	-	5('abb')
2('aa')	-	-	4('aaa')	-	5('abb')
3('ab')	-	-	-	-	6('abbb')
4('bb')	-	-	-	-	-
default	0('')	1('a')	2('aa')	3('ab')	0('')

Fig. 6. Reducing table size using indirect pointers

3.4 Memory Size Reduction

When the total length of rules is n, T and I_k contain $O(n)$ strings. Therefore, $O(n^2)$ size of memory is necessary for 2D-tables C_k and NS_k. However, the number of valid elements in each 2D-table is at most $O(nk)$. Therefore, table size can be reduced using indirect pointers as shown in Figure 6. The 2D-table is decomposed into rows, and then packed into the linear array. Each entry of the linear array has a row number to identify the owner row of the entry. When the 2D-table lookup fails, the default value of each column is returned.

Note that unused elements may exist in the linear array because of fragmentation of the free area. We define *packing density* as the value (# of valid elements in the linear array)/(total linear array length). When the packing density is 100% (i.e. rows are packed without a gap), the maximum size of each table is $O(nk)$. The number of tables for each k is w/k for $k = 1, 2, 4, \cdots, w$. Therefore, total table size is $O(\sum_k nk \cdot w/k) = O(nw \log w)$. However, as we show in Section 5, typical memory usage is about $O(nw)$. This is because the typical size of each table is $O(n)$ rather than $O(nk)$.

Actual table size depends on the packing density, which can be improved by converting the indices of the 2D-table using appropriate functions. The worst-case usage of the linear array and the best conversion function are not yet established; with complex analyses being necessary to identify them. This problem constitutes a future work focus. In the evaluation here, we used a simple conversion that consists of 3-input binary functions selected by heuristics.

4 Implementation on FPGA

The left diagram in Figure 7 shows an outline of our implementation of C_k and NS_k functions. These circuits are pipelined and the latency is 3 clocks. The ID1

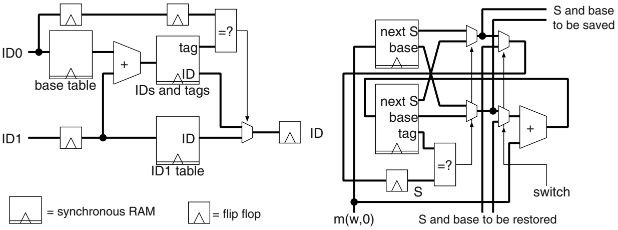


Fig. 7. Simplified diagram of C_k and NS_k (left) and S (right)

table holds the default output for each ID1. The base table holds the indirect pointers used in the 2D-table lookup. When the 2D-table lookup fails, the output of the ID1 table is selected. These three tables are implemented using on-chip RAMs of generic FPGAs. The number of RAMs used for each table is minimized based on the table size. As mentioned in Section 3.4, in C_k and NS_k functions, ID0 and ID1 are actually converted before the 2D-table lookup.

The bandwidth of SBT depends on the calculation latency of the next state S. When the latency is l cycles, the resulting bandwidth is w/l. The latency of the naive implementation of NS_k is 3 clocks. So we used a special circuit to update S in 1 clock latency as shown in the right side of Figure 7. In the circuit, state S and its indirect pointer (base) are calculated simultaneously. The latency from $m(w, 0)$ input to S output is 2 cycles. The circuit has selectors and signals to switch states. The matcher can switch between streams with no dead cycle. In fact, we used an optimized version of the circuit of S to improve clock speed.

The critical path consists of C_1, C_2, C_4, $\cdots$, C_w, S, $NS_{w/2}$, $NS_{w/4}$, $\cdots$, NS_1. Therefore, the calculation latency is $3 \cdot \log w + 2 + 3 \cdot \log w = 6 \log w + 2$ cycles.

5 Evaluation

We made a converter that generates VHDL files of SBT matchers for Xilinx XC2V FPGAs from given rules. The tables in C_k, NS_k, and S are implemented using on-chip 18Kbit 2-port Block RAMs. Each Block RAM is shared by two instances of the same I_k or NS_k function if possible to save Block RAMs.

The converter generated SBT units for an XC2V6000-6 FPGA, and they are evaluated using timing analyzer of Xilinx ISE6.1i. We use the three rule sets shown in Table 1 which are generated by randomly choosing patterns from the rules of Snort[13]. Because of a restriction in our tool, the total size of the rules is limited, and w must be greater than or equal to 4. Therefore, we used rule sets of less than about 2000 characters, and evaluated configurations of $w \geq 4$.

5.1 Size of Tables

Figure 8 shows the number of bits and Block RAMs used in each configuration. Total of 144 Block RAMs are available on an XC2V6000. Set3 did not fit in the

Table 1. Pattern sets used in the evaluation

	set1	set2	set3
# of patterns	40	72	180
total size(bytes)	523	1003	2178

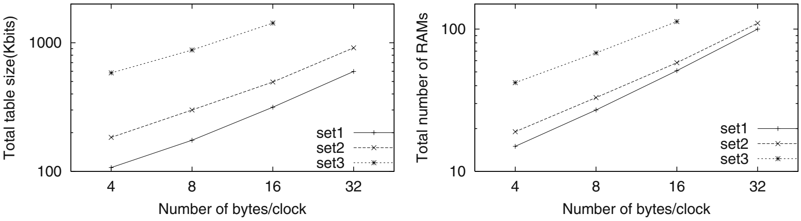


Fig. 8. Table size and number of Block RAMs

XC2V6000 when $w = 32$ because it ran out of Block RAMs. We can see that the total table size is proportional to w. This result is better than the theoretical worst case value $O(w \log w)$. The packing density of each 2D-table was always 100% because most of the rows of 2D-tables contained at most 1 valid element. The maximum slice usage was 7%.

Theoretically, the table size is proportional to the total rule size. However, the actual table size grows irregularly as the rule size increases. This is because the size of each table grows discretely since its depth is limited to a power of 2.

5.2 Impact of Multiple Streams

At each stream switching, S and its indirect pointer have to be saved and restored. Their total bit width is theoretically $O(\log n)$. In the experimental result, bit width was 24 bits at maximum and about 20 bits in typical cases. In this case, 512 streams can be supported using the 2K-bytes on-chip RAM, and 256K streams using a 1M-bytes SRAM. Therefore, in terms of state switching, multiple streams can be supported with small switching overhead and memory usage.

5.3 Bandwidth

The throughput of the SBT method is $8wf$ bps, where f is the operating frequency. Since the total table size is $O(nw)$ in the experimental result, maximum value of w is $O(M/n)$, where M is the available memory size. Therefore, when f is a constant, the maximum bandwidth is proportional to M.

Actually, the clock frequency f is not a constant. Figure 9 shows the evaluation result of maximum clock speed and bandwidth using the post place&route timing analyzer of Xilinx ISE6.1i. The bandwidth reduction of set3 is due to the clock speed degradation caused by a high fan-out of address lines shared by many Block RAMs.

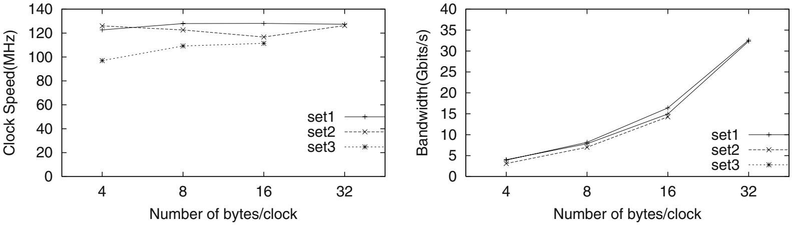


Fig. 9. Maximum clock speed and bandwidth

6 Related Works

Cho et al.[2] realized comparator based string matchers, and Sourdis et al.[3] achieved 10Gbps using a similar method. In CAM based methods [4], the patterns can be updated dynamically. Since these methods have difficulty in stream switching, they are not suitable for scanning TCP streams. On the other hand, our SBT enables simple and lightweight stream switching.

Sidhu et al.[5] proposed a regular expression matching method based on NFA, and Hutchings et al.[6] applied a similar method to NIDS. Clark et al. [7] extended the NFA method to process multiple characters in one clock. They showed that 100Gbps bandwidth is possible if clock frequency is 200MHz and number of parallel bytes is 64. A DFA based method is used in [8]. Those methods have an advantage that rules can be described in regular expressions. However, they have difficulty in stream switching because $O(n)$ state bits have to be saved at each state switching. On the other hand, our method enables fairly high throughput while allowing lightweight state switching.

Baker et al.[10] proposed an area efficient string matching mechanism based on the KMP algorithm. However, it cannot process multiple characters at once, limiting the throughput. The Bloom Filter method[9] uses a hash to eliminate redundant searches but has problems in worst case performance and scalability.

7 Concluding Remarks

In this paper, we have proposed SBT, a trie-based string matching method. We implemented a converter that automatically generates VHDL files of SBT units from given rules. The SBT method enables lightweight stream switching that was impossible in existing methods because of higher switching overhead and larger memory size. In addition, SBT can achieve 10Gbps for practical rules.

Using the SBT method, high-speed TCP scanners can be realized, and more reliable information can be gathered. The method is especially useful for security applications. We are developing a multi-port NIC with FPGAs. Using this NIC, we will realize a high-speed TCP scanner based on the SBT method. However, to realize a complete TCP packet scanner, a stream reassembler is necessary. For TCP stream reassembly, other difficult problems exist; these must be resolved. We intend to work on such problems in our future work.

Acknowledgements. This research is partially supported by the Special Coordination Fund for Promoting Science and Technology, CREST project of Japan Science and Technology Corporation, and 21st century COE project of Japan Society for the Promotion of Science.

References

1. C. J. Coit, S. Staniford, J. McAlerney: Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. In: DISCEXII, DARPA Information Survivability conference and Exposition. (2001)
2. Y. H. Cho, S. Navab, W. H. Mangione-Smith: Specialized Hardware for Deep Network Packet Filtering. In: Proc. of 12th Intl. Conf. on Field Programmable Logic and Applications(FPL '02). (2002)
3. I. Sourdis, D. Pnevmatikatos: Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In: Proc. of 13th Intl. Conf. on Field Programmable Logic and Applications(FPL '03). (2003)
4. M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, V. Hogsett: Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. In: Proc. of 12th Intl. Conf. on Field Programmable Logic and Applications(FPL '02). (2002)
5. R. Sidhu, V. K. Prasanna: Fast regular expression matching using fpgas. In: Proc. of 9th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01). (2001)
6. B. L. Hutchings, R. Franklin, D. Carver: Assisting network intrusion detection with reconfigurable hardware. In: Proc. of 10 th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'02). (2002) 111–120
7. C. Clark, D. Schimmel: Scalable pattern matching for high speed networks. In: Proc. of 12th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'04). (2004)
8. J. Moscola, J. Lockwood, R. P. Loui, M. Pachos: Implementation of a content-scanning module for an internet firewall. In: Proc. of 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03). (2003) 31 – 38
9. S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. Lockwood: Deep packet inspection using parallel bloom filters. In: Proc. of 11th IEEE Symp. on High Performance Interconnects (HotI '03). (2003) 44 – 51
10. Z. K. Baker, V. K. Prasanna: Time and Area Efficient Pattern Matching on FPGAs. In: Proc. of the 2004 ACM/SIGDA 12th Intl. Symp. on Field programmable gate arrays(FPGA'04). (2004) 223–232
11. A. V. Aho, M. J. Corasick: Efficient String Matching : An Aid to Bibliographic Search. Communications of the ACM **Vol. 18** (1975) 333 – 340
12. M. Handley, V. Paxson: Network Intrusion Detection : Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In: Proc. of 10th USENIX Security Symposium. (2001)
13. M. Roesch: Snort - Lightweight Intrusion Detection for Networks. In: Proc. of Lisa'99: 13th Administration Conference. (1999)
14. Y. Sugawara: Correctness Proof of the SBT method. Technical report, Dept. of Computer, Science, Univ. of Tokyo (2004)

Hardware Design of a FPGA-Based Synchronizer for Hiperlan/2

Mª José Canet[1], Felip Vicedo[2], Vicenç Almenar[3],
Javier Valls[1], and Eduardo R. de Lima[3]

[1] Dpto. Ingeniería Electrónica, Universidad Politécnica de Valencia, Gandia, Spain
[2] Dpto. Física y Arquitectura Computadores, Universidad Miguel Hernández, Elche, Spain
[3] Dpto. Comunicaciones, Universidad Politécnica de Valencia, Gandia, Spain

Abstract. This paper deals with the design and implementation of a frame, time and frequency synchronizer for Hiperlan/2 WLAN standard. In a packet oriented system, to perform a quick and correct synchronization it is critical to avoid severe bit error rate degradation. So, the design of this subsystem is one of the most challenging tasks to be done in the implementation of a transceiver. In this paper we give practical solutions to the hardware design problems that arise when the synchronization algorithm is turned into a digital circuit. We evaluate the fixed-point realization of the synchronization algorithm and introduce some simplifications to reduce, as much as possible, the cost in area of the circuit without losing its performance.

1 Introduction

Hiperlan/2 is a Wireless LAN (WLAN) standard from ETSI BRAN [1,2] which works in the 5 GHz band and achieves data rates up to 54 Mbps. For the physical layer, Orthogonal Frequency Division Multiplexing (OFDM) is used, since it allows getting high bit rates in highly dispersive fading environments. To prevent intersymbol interference (ISI) a cyclic prefix is added to the OFDM symbol, this prefix can also be employed to have some tolerance for symbol timing errors.

This paper presents part of the work carried out at the *Universidad Politécnica de Valencia* for the design and implementation on Xilinx FPGAs of an OFDM based WLAN transceiver. This paper deals with the frame, time and frequency synchronization stage for Hiperlan/2 standard.

In Hiperlan/2 systems data is transmitted in bursts, always preceded by a preamble. So, the acquisition stage at the receiver must be quick enough to get all the data in the burst.

Most of the practical solutions for frame and time synchronization based on a pilot preamble that can be found in the literature are based on the autocorrelation of the received signal, as it was proposed by Schmild and Cox [3]. For example, in [4] a hardware architecture of a time OFDM synchronizer is presented. This is similar to [3] but some simplifications have been carried out on the algorithm. First, in [4] the

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 494–504, 2004.

branch that calculates the energy of the received signal is removed. We have checked that if the signal amplitude varies and the input signal energy is not calculated, the synchronizer performance is drastically reduced. Besides the autocorrelation scheme, it is possible to perform the time and frame synchronization using the cross-correlation between the received and the known preamble, but this solution has smaller performance [5] and bigger hardware cost [6].

This paper describes in detail the hardware implementation of a frame, time and frequency synchronizer for Hiperlan /2. This paper is organized as follows. Section 2 presents an overview of the physical layer. Section 3 deals with frame detection, Section 4 with symbol time estimation and Section 5 with CFO estimation and CFO compensation. In Section 6 complete synchronizer is commented and in Section 7 hardware results are summarized. Lastly, in Section 8 some conclusions are presented.

2 Physical Layer

In Hiperlan/2 the baseband signal is built using a 64-IFFT. Then a cyclic prefix of 16 samples is added to make the system robust to multipath. Each symbol is 80 samples long, that gives a duration of 4μs with a sampling frequency of 20 MHz.

During the broadcast phase, the access point transmits a preamble that is used by mobile terminals (MT) to perform frame synchronization, time synchronization, automatic gain control (AGC), frequency synchronization and channel estimation. Therefore, the synchronization phase in an MT can be divided in three parts: detection of the broadcast preamble; time synchronization, that consists in estimating the sample when the OFDM symbol starts; and carrier frequency offset (CFO) estimation. These tasks should be done using the broadcast preamble structure (Fig. 1). In this structure we can distinguish 3 sections: A, B, and C. A and B sections are intended for frame detection, time synchronization, AGC and coarse CFO estimation; section C can be used for fine CFO estimation and channel estimation.

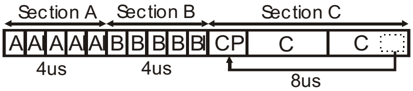


Fig. 1. Broadcast preamble

3 Frame Detection

Frame detection is the first phase of the whole synchronization process, and the most important, since a false detection could cause that an MT transmission begins at a wrong instant [5].

3.1 Detection Algorithm

We will work with one of the algorithms proposed in [5], this makes use of the autocorrelation of the received signal in a similar way as in [3]. In Fig. 2 it is shown the block diagram of this algorithm. The received signal is correlated with a delayed (16 samples) version of itself. Then, the obtained signal is averaged over 48 samples giving the autocorrelation signal *R* and scaled by the mean power (signal *P*) of the input signal.

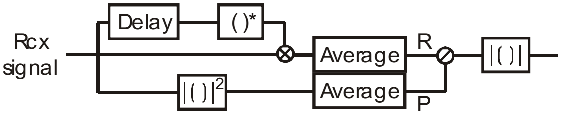


Fig. 2. Block diagram of the autocorrelation scheme

Fig. 3 shows the output, without any channel distortion, of the autocorrelation scheme (magnitude, and phase scaled by π) for Hiperlan/2 broadcast preamble. This result is compared with the output obtained when an uplink long preamble is introduced, since the frame detection algorithm must distinguish between both kinds of preambles. On the one hand, the broadcast preamble has two peaks in the magnitude plot, one in section A (sample 64) and another in section B (sample 144). Also, the phase is ±π during section A and 0 during section B. On the other hand, the uplink preamble has a magnitude of 1 between samples 64 and 144 and has a phase of 0.

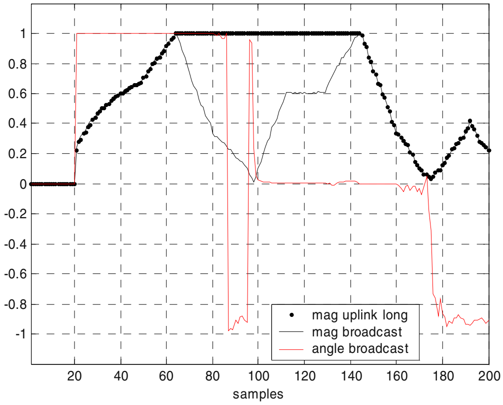


Fig. 3. Output of the autocorrelation scheme for Hiperlan/2 broadcast and uplink preambles.

The detection algorithm sets a magnitude threshold THR_{MAG} and a phase checking. After threshold is exceeded, the phase of the 30 previous samples is checked. If they are grater (in magnitude) than π/2, the algorithm assumes that a broadcast preamble is present. In order to detect the phase condition it is enough to look at the sign of the real part of the autocorrelation scheme output (it should be negative).

To evaluate the performance of this algorithm we have measured the probability of detection failure (DF) and of false alarm (FA) under these conditions:

- Channel model A (*delay spread* of 50 ns), since it represents a typical indoor office environment [7].
- Number of test frames: 10.000
- Signal clipping (preamble): 5 dB below the rms, 5 dB above the rms and no clipping (10 dB above the rms).
- SNR: 10 dB. This is a worst case since to achieve an acceptable packet error rate (10^{-2}), the SNR should be at least 12 dB, for a 6 Mbit/s bit rate [8].

The implementation of the scheme presented above would require one complex multiplier (3 real multipliers and 5 adders) to calculate the autocorrelation, two real multipliers to obtain the input signal energy, two dividers to scale the real an imaginary autocorrelation output, and a CORDIC [9] to estimate output magnitude. All this operators will require a lot of hardware resources in the FPGA. In this paper we propose to modify the synchronization algorithm, insofar as possible, in order to reduce the hardware cost, but without reducing the performance (without increasing probability of DF or FA).

Some simplifications of the initial algorithm have been studied. For this algorithm, the threshold magnitude condition [3] is modified as follows:

$$\left|\frac{R}{P}\right| > THR_{MAG} \rightarrow |R|^2 > THR_{MAG}^2 \cdot |P|^2 \tag{1}$$

The main purpose of this modification is to eliminate the dividers, since their hardware design is complicated and their cost is high. We have replaced two divisors by two multipliers, one to calculate the squared energy and another to multiply it by the squared threshold. Also we obtain the squared modulus of the averaged and scaled signal instead of the modulus: the use of a CORDIC, which has a high hardware cost, is avoided. This is replaced by two multipliers and one adder.

The performance of this modification has been compared with the one of the original algorithm. Fig. 4 illustrates the obtained results in Hiperlan/2 for a signal clipping of 5 dB below the rms. Floating-point data has been considered, and magnitude thresholds between 0.1 and 0.9 have been evaluated. In original algorithm, which works with THR_{MAG}, DF probability is very small for thresholds below 0.75. Simplified algorithm works with THR^2_{MAG}, for this reason the magnitude threshold is lower than before. The DF probability is lower than 10^{-3} for thresholds under 0.55 and FA probability is lower than 10^{-3} for thresholds over 0.4. Finally, we have decided to use a threshold of 0.5 because it provides a good performance and avoids the use of one multiplier (multiply by 0.5 is performed as a hardwired shift).

3.2 Finite Precision Analysis and Hardware Implementation

A Matlab finite precision model of the whole synchronizer has been performed. The quantization has been applied to each block of the system, meanwhile the others blocks keep the ideal behavior. The aim has been to reduce the precision keeping the same performance as in the floating-point solution. This study has been carried out with a signal clipping of 5 dB under the preamble rms and without signal clipping (10 dB above the preamble rms).

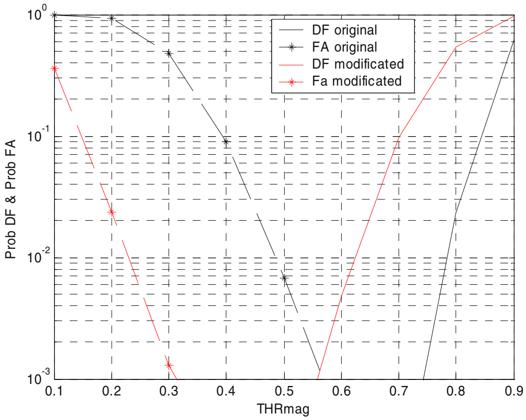


Fig. 4. DF and FA probability in Hiperlan/2

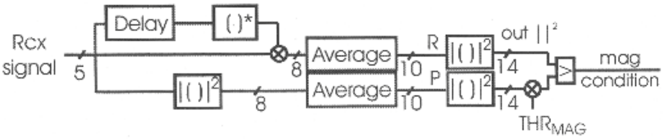


Fig. 5. Implemented synchronizer

The average circuit has been implemented with an accumulator and a subtractor. In each cycle, the accumulator adds a new sample and the subtractor eliminates the sample 48th. Phase condition is fulfilled if the real part of the autocorrelation output is negative during 30 cycles. Their most significant bit (MSB) is registered with 29 flip-flops, and then multiplied (with an AND gate) with the MSB of the last sample. Only one comparator is needed to check the magnitude condition. It is assumed that a broadcast preamble is present if phase and magnitude conditions are met (another AND gate is used). Finally, a low cost circuit has been implemented to reduce FA probability: magnitude and phase condition must meet during more than 8 cycles. With these circuits we have obtained DF and FA probability (in %) of the implemented synchronizer, for 10000 test frames, 10dB SNR and channel model A. Table 1 summarizes the obtained results.

Table 1. Implemented synchronizer performance

Clipping	**10dB rms**	**5dB rms**	**-5dB rms**
DF	0.05%	0.03%	0.64%
FA	0.20%	0.14%	0.08%

4 Time Synchronization

After frame detection is accomplished, it is necessary to estimate the first sample of the broadcast preamble. This estimation must be as accurate as possible to avoid ISI with previous or later symbols when the FFT window is taken. Multipath channel, analog filters required in transmitter and receiver, and interpolate and decimate filters distort 6 or 7 samples of the cyclic prefix [10]. So, there is a window of 9 samples where the FFT can begin without introducing ISI.

In Hiperlan/2, after detection of the maximum magnitude value en section A, section B peak is searched in order to perform time synchronization. An average of the autocorrelation scheme output has been made and a low cost maxima detection is used: actual and previous samples are compared. If actual sample is smaller than previous sample during 2 cycles, we assume that there is a maximum.

Fig. 6 illustrates the deviation between the symbol timing estimator and the ideal initial sample using the implemented synchronizer when there is not signal clipping (we assume that AGC has finished during A phase). It can be seen that 95% of the estimated initial samples fall in the range between 0 and 4, and that the maximum deviation is lower than 9, which was the maximum width of the valid window. So, this solution is valid and would not degrade the performance of the receiver due to ISI.

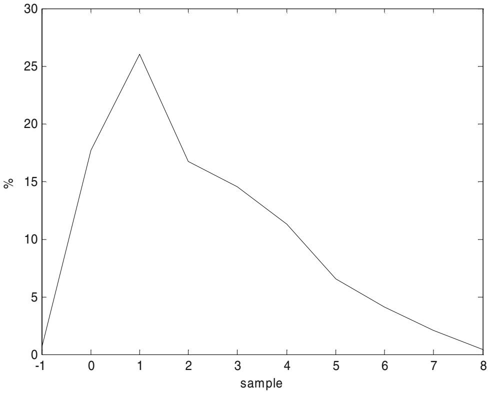


Fig. 6. Deviation from initial sample

5 CFO Estimation and CFO Compensation

One of the main drawbacks of OFDM is its sensitivity to carrier frequency offset (CFO). In Hiperlan 2, section B peak is used for coarse CFO estimation. The angle of correlator output ($\angle(R)$) is calculated with a circular vectoring mode CORDIC and the estimated frequency offset ($\hat{f}_o$) can be calculated by [3]:

$$\hat{f}_o = \frac{\angle(R)}{2 \cdot \pi \cdot N_c \cdot T}, \tag{3}$$

where T is the sampling period and N_c is 16. Next, the estimated frequency will be used to remove this CFO from de signal, by using the same CORDIC processor, configured as circular rotation mode.

The maximum frequency error that can be estimated depends on the delay (D) in the autocorrelation scheme [11]:

$$\hat{f}_{o\max} = \frac{1}{2 \cdot D \cdot T}, \tag{4}$$

The autocorrelation scheme used to detect B peak has D=16, so $\hat{f}_{o\max}$ =625kHz. This frequency offset is higher than 212kHz (20ppm) frequency offset allowed in [1].

The effect of carrier frequency error on the performance can be measured by the SNR loss. For relatively small frequency errors, can be calculated by [11]:

$$SNR_{Loss} = \frac{10}{3\ln 10}(\pi T f_\Delta)^2 \frac{E_S}{N_0} dB, \tag{5}$$

where f_Δ is the frequency error as a fraction of the subcarrier spacing (SSP) and T is the sampling period. The performance varies strongly with the modulation used. For example, 64-QAM can not tolerate more than 1% error in the carrier frequency for a SNR loss of 0.5dB, while QPSK can tolerate up to 5% for the same SNR loss. On the other hand, 64-QAM needs at least 30dB SNR to achieve an acceptable packet error rate (10^{-2}), whereas BPSK only needs 12dB [8]. A higher SNR improves the performance of the CFO estimator [11].

First, a floating point analysis has been made. It can be supposed that in section B there is not clipping. Multipath channel model A is used. It has been obtained that the worst case is 64-QAM (30 dB SNR). An error less than 1.6% can not be reached. For this error we have an SNR loss of 1.6dB. If less SNR loss is desired, a fine frequency offset technique must be also used. In BPSK (10dB SNR) there are not problems: the error is less than 5% and, therefore, the SNR loss is less then 0.5dB.

In order to reduce errors in CFO estimation, section C of the broadcast preamble is used. First, the estimated coarse CFO must be compensated from samples of Section C. To do this we reuse the same CORDIC that was employed for coarse CFO estimation. Next, corrected Section C is autocorrelated using a delay of 64 samples and an average of 96 samples. Then, the obtained peak is used to calculate fine CFO with the same CORDIC as before. The autocorrelation can be performed with the same structure used in A, B sections. Only some multiplexers in the delay block and in the average block must be included.

The autocorrelation scheme used to detect C peak has D=64, so $\hat{f}_{o\max}$ =156,25kHz. This frequency offset is lower than the worst case of 212kHz. For this reason, first it is necessary to perform a coarse CFO estimation and, after compensation a fine CFO estimation. In this, standard deviation for 30dB SNR (64-QAM) is less than 0.3% in a floating point analysis.

Now, a fixed point analysis has been made. Our frame and time synchronizer only needs 5 bits in the input. We have checked that this input precision it is not sufficient for the coarse and fine CFO estimator. At least 6 input bits are needed. Moreover, the output of the multipliers and the output of the average must have complete precision. The inputs of the CORDIC need 8 bits precision and the output must have also 8 bits. In Fig. 7 final synchronizer is shown. The CORDIC used in the CFO estimation can be reused in CFO correction, so the required hardware is reduced. A standard deviation of 0.35% is reached with the implemented fine CFO estimator, so SNR loss is less than 0.5dB, as shown in Fig. 8.

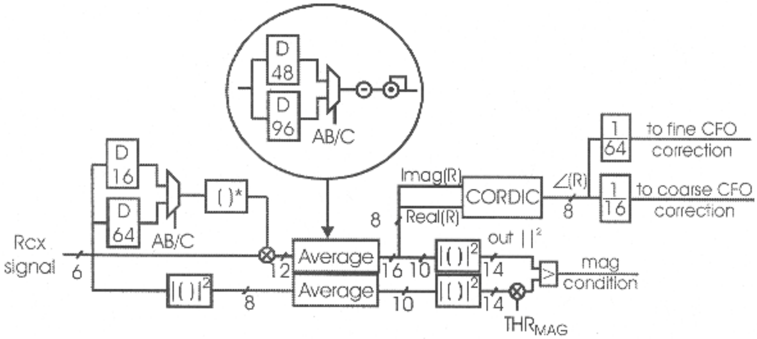


Fig. 7. Final synchronizer

Once fine CFO is calculated, Section C must be corrected before being used for channel estimation. The received OFDM symbols must be corrected with a combination of the coarse and fine CFO estimation.

6 Complete Synchronizer

Figure 9 shows complete synchronizer. It is composed of three blocks: frame detection, time synchronization and CFO estimation & compensation. Frame detection block is based on an autocorrelator and a circuit that finds Section A peak of the broadcast preamble. With this information, frame detection block generates an output

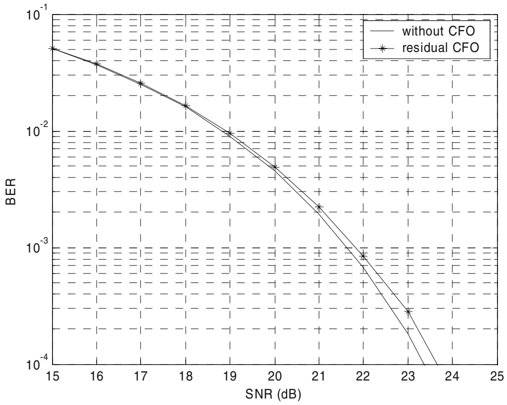


Fig. 8. Measurement of SNR loss due to residual CFO

(final_cond) that indicates where Section B peak can be found (±20 samples). Time synchronization block searches Section B peak of the autocorrelator output when final condition is accomplished. This peak is used to enable CFO estimation. Once coarse CFO is estimated, CFO in Section C is compensated using this information and, next, fine CFO is estimated (autocorrelation of Section C is done with frame detection block, and then Section C peak is found with time synchronization block). Finally, input OFDM symbols are compensated with coarse and fine CFO estimation.

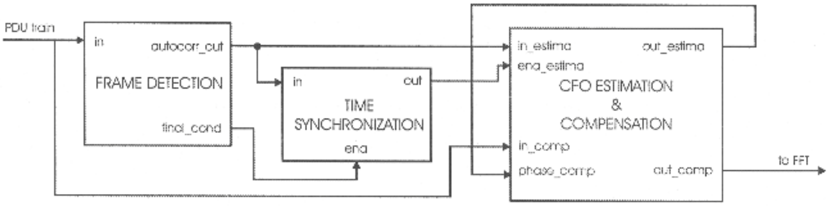


Fig. 9. Complete synchronizer

7 Hardware Results

The design of all necessary circuits has been done using the tool System Generator from Xilinx [12], this tool is integrated in the Matlab-Simulink environment and allows rapid prototyping. The synchronizer has been implemented on a Spartan-3 Xilinx FPGA. These devices have slices composed by look-up tables to implement logic and arithmetic resources to propagate carries, and embedded multipliers which simplify our design. In table 2 are summarized the resources that are required by each block.

So, the whole synchronizer requires 827 slices and 10 embedded multipliers and runs at 20MHz. The design fits in a XC3S400-4 Spartan-3 device (4775 slices, 16 BRAM and 16 embedded multipliers).

Table 2. Hardware resources

	Slices	**Mults**
Frame Detection	411	8
Time Synchronization	64	0
CFO Estimation & Compensation	310	2
Control Logic	42	0

8 Conclusions

In this paper we have presented a hardware design of a frame, time and frequency synchronizer for Hiperlan/2. We have taken one of the algorithms proposed in [5] and have introduced some simplifications to reduce the area cost of the final circuit. The objective has been that these simplifications and the fixed point design maintain the performance achieved by the original floating-point algorithm. Finally we have implemented all the necessary circuits in a Xilinx FPGA to validate the process.

Acknowledgements. This work was supported by the *Ministerio de Cicencia y Tecnología* under Research Project TIC2001-2688-C03, by *Generalitat Valenciana* research grant CTBPRB/2002/225, and in part by the *Universitat Politècnica de València*.

References

1. ETSI TS 101 475 v1.2.2 BRAN ; HIPERLAN Type 2 ; Physical (PHY)layer.
2. R. Van Nee and R. Prasad. *OFDM for Wireless Multimedia communications.*Archech House, 2000.
3. T. Schmidl, and D. Cox. "*Robust Frequency and Timing Synchronization for OFDM"*. IEEE Trans. On Comm. Vol 45, No. 12, December 1997.
4. S. Johansson, M. Nilsson, P. Nilsson. "*An OFDM Timing Synchronization ASIC"*.ICECS2000. Jounieh,Lebanon, 2000.
5. V. Almenar, S. Abedi, Rahim Tafazolli. "*Synchronization Techniques for HIPERLAN/2"*. VTC 2000 (Fall), Atlantic City, USA, 2002.
6. A. Fort, J. Weijers, V. Derudder,W. Eberle, A. Bourdoux. "*A performance and complexity comparison of auto-correlation and cross-correlation for OFDM burst synchronization"*. ICASSP 2003. Hong Kong, 2003.
7. BRAN WG3 PHY Subgroup. *Criteria for Comparison*. ETSI/BRAN document no. 30701F, 1998.
8. J. Khun-Jush,G. Malogren, P.Schramm and J. Torsner. *HIPERLAN type 2 for broadband wireless communication.* Ericsson Review No.2, P108-120, 2000.
9. F. Cardells, and J. Valls, "*High Performance Quadrature Digital Mixers for FPGA*", *FPL2002*, Monpelier, France, 2002.

10. M.J. Canet, F. Vicedo, J. Valls, V. Almenar. “*Design of a digital front-end transmitter for OFDM-WLAN systems using FPGA”*. ISCCSP 2004, Hammamet, Tunisia, 2004.
11. J. Heiskala, J. Terry. *OFDM Wireless LANs: A theoretical and practical guide.* SAMS Publishing, 2001
12. Xilinx System Generator for DSP v2.2 Reference Guide

Three-Dimensional Dynamic Programming for Homology Search

Yoshiki Yamaguchi[1], Tsutomu Maruyama[2], and Akihiko Konagaya[1]

[1] RIKEN Genomic Sciences Center,
1-7-22 Suehiro-cho Tsurumi-ku Yokohama Kanagawa, 230-0045, Japan
[2] Institute of Engineering Systems and Mechanics, University of Tsukuba,
1-1-1 Ten-ou-dai Tsukuba Ibaraki, 305-8573, Japan

Abstract. Alignment problems in computational biology have been focused recently because of the rapid growth of sequence databases. Many systems for alignment have been proposed to date, but most of them are designed for two-dimensional alignment (alignment between two sequences), because huge amount of memory and very long computational time are required by alignment among three or more sequences. In this paper, we describe a compact system with an off-the-shelf FPGA board and a host computer for three-dimensional alignment using Dynamic Programming. Through our approach, high performance are attained by "two phase search" with reconfigurations of an FPGA and co-processing the FPGA and software. Furthermore, in order to achieve higher parallelism in the FPGA, we use a payoff matrix for matching elements in sequences and the matrix is divided into sub-matrices which are minimized. In comparison to a single Intel Pentium4 2.53GHz processor, our system with a single XC2V6000 enables more than 250-fold speedup.

1 Introduction

Alignment problems in computational biology, namely homology search, have been focused recently because of the rapid growth of sequence databases[1,2,3]. By computing alignment, we can investigate true similarity among the sequences. Dynamic Programming (DP)[4,5] is a technique which is applied to finding the optimal alignment among sequences. In DP, the search space is expressed as d-dimensional lattice, where d stands for the number of sequences. The method requires traceback pointers which indicate connections to its neighboring $2^d - 1$ sites. We need to store all the traceback pointers so that the optimal alignment is obtained. Hence, its computational complexity is very high (O(N^d), to make comparisons among d sequences with a length N), so it is not realistic to use algorithms based on DP even for alignment between two sequences on desk-top computers. In order to reduce the computational time, many heuristic algorithms[6,7,8] and hardware systems [9,10,11,12,13,14,15,16] have been proposed. Most of them, nevertheless, are designed for two-dimensional alignment (alignment between two sequences) because huge amount of memory and very long computational time are required by alignment among three or more sequences.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 505–515, 2004.

In this paper, we describe a compact system which consists of an off-the-shelf FPGA board and a host computer for three-dimensional alignment using DP. In our approach, high performance is attained by "two phase search" with reconfigurations of an FPGA, and co-processing the FPGA and software. In the first phase, only the final result (score of the optimal alignment) is calculated by the FPGA. In this phase, no traceback pointers are output. Here, the system achieves maximum performance which depends only on the size of the FPGA. Then, if the first phase gives good scores, the circuit for the second phase is reconfigured on the FPGA, and the optimal alignment is obtained by the co-processing the FPGA and software. The total number of traceback pointers could be very large. Thus, it is not realistic to output all of them under the limited memory bandwidth of current FPGAs. To address this program, a part of scores obtained during the search are output by the FPGA, and sent to the host computer. These outputs are utilized in our technique for a deduction of the search space in three-dimensional DP, thereby the optimal alignment is obtained on the host computer efficiently.

Furthermore, in order to achieve higher parallelism in the FPGA, three-dimensional payoff matrix used for matching elements in sequences is divided into several sub-matrices, and each of them is minimized.

This paper is organized as follows. Section 2 describes the overview of three-dimensional DP. Then, the details of the approach are described in Section 3. In Section 4, processing units implemented on an FPGA are shown. Current status and future works are given in Section 5 and Section 6.

2 Three-Dimensional Dynamic Programming

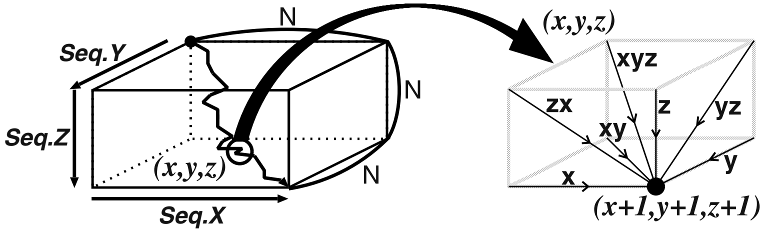
2.1 Overview

Dynamic Programming (DP) takes account of all possibility in search space and always finds the optimal result. In this principle, the complexity in finding the optimal result is minimized when we have all the traceback pointers. However, computational time required for obtaining all the traceback pointers is long. As an attempt to reduce the computational time, stochastic algorithms are extensively used[6,7,8], but they may miss the optimal result.

Fig.1 shows search space of three-dimensional DP, and how alignment is searched in the space. The length of three sequences is N so that the size of the space is $N \times N \times N$. For each lattice site in the space, a score is calculated by a procedure shown in Fig.2, which is illustrated in Fig.1(right). A single site has its 7 subsequent sites connected by the traceback pointers (the arrows with *xyz, xy, yz, zx, x, y, z*). Then, each score through the seven connections is calculated using a payoff matrix.

Suppose that the address of a lattice site is (x, y, z), the payoff matrix is accessed using three elements (*seq_X[x], seq_Y[y]* and *seq_Z[z]*) as an address. For all sites in the search space, their scores and their traceback pointers which give the scores, are stored in memory. These traceback pointers are traced back from the final site to the start site by dereferencing connections. A path given by this dereferencing provides the optimal alignment.

The computational order is N^3 for calculation of all scores in the search space. Also, memory size amounts to $O(N^3)$. On the other hand, only $O(N)$ is required to trace back.



Search Domain (cube, N on a side) New payoff is computed from 7 directions.

Fig. 1. The overview of three-dimensional DP

```
char Sequence_X[Length_of_Sequence_X];
char Sequence_Y[Length_of_Sequence_Y];
char Sequence_Z[Length_of_Sequence_Z];

int Score[Length_of_Sequence_X][Length_of_Sequence_Y][Length_of_Sequence_Z];

int 3D_Payoff_matrix[Num_of_elements][Num_of_elements][Num_of_elements];
int 2D_Payoff_matrix[Num_of_elements][Num_of_elements];

Calculate_Score(x, y, z){
    xyz = Score[x][y][z]+3D_Payoff_matrix[Sequence_X[x]][Sequence_Y[y]][Sequence_Z[z]];

    zx = Score[x][y+1][z]+2D_Payoff_matrix[Sequence_Z[z]][Sequence_X[x]]+gapcost_y();
    yz = Score[x+1][y][z]+2D_Payoff_matrix[Sequence_Y[y]][Sequence_Z[z]]+gapcost_x();
    xy = Score[x][y][z+1]+2D_Payoff_matrix[Sequence_X[x]][Sequence_Y[y]]+gapcost_z();

    x = Score[x][y+1][z+1]+gapcost_y()+gapcost_z();
    y = Score[x+1][y][z+1]+gapcost_z()+gapcost_x();
    z = Score[x+1][y+1][z]+gapcost_x()+gapcost_y();

    Score[x+1][y+1][z+1]=selecting_maximum_score(xyz, xy, yz, zx, x, y, z);
}
```

Fig. 2. Computation of the score on a node (processing unit)

2.2 Parallel Processing of Three-Dimensional DP

Fig.3 illustrates how three-dimensional DP can be processed in parallel. In Fig.3, the length of sequence(X), sequence(Y), and sequence(Z) are N, w, and w, respectively, where $w \ll N$. Here, w is a number which is determined from the size of FPGA. The computation starts from the top left corner in the far side of the $N \times w \times w$ rectangular parallelepiped (we call it *bar object*), and progresses along X axis. In this case, $w \times w$ lattice sites in the bar object can be processed in parallel at maximum. By this parallel processing, the computational time of the bar object can be reduced from $O(w^2N)$ to $O(N)$. In our current implementation, 64 processing units can be implemented on a single XC2V6000. Thus, the maximum number of w becomes 8. In general, the length of sequences (N) are much longer than 8.

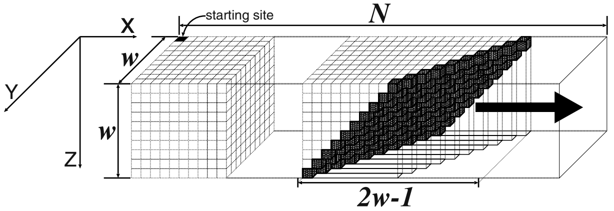


Fig. 3. Parallel processing of three-dimensional DP (small size)

In order to compute alignment among long sequences with a limited number of processing units, we have extended the technique. To carry out this technique, the search space ($N \times N \times N$) is divided into sub-spaces ($w \times w \times N$) to which the above procedure is sequentially applied. Fig.4 illustrates how the search space is divided into sub-spaces, and how the bar objects are processed in parallel with $w \times w$ processing units. The computational procedure of this search in Fig.4 are described below.

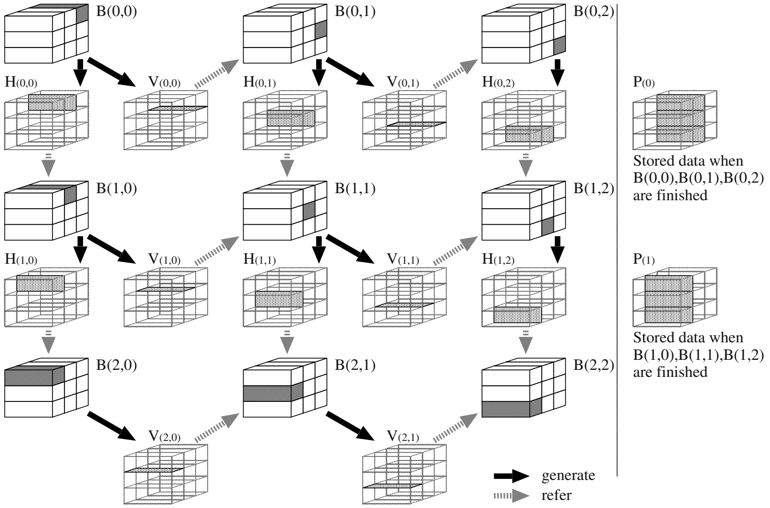


Fig. 4. Parallel processing of three-dimensional DP (large size)

B(0,0): The shaded bar object B(0,0) is processed first. Scores on the two rectangles (H(0,0) and V(0,0)) are stored.

B(0,1): B(0,1) is processed subsequently. Here, V(0,0) is used as boundary conditions to B(0,1). Then, H(0,1) and V(0,1) are stored.

B(0,2): In the same way, B(0,2) is processed using V(0,1). In this case, only H(0,2) is stored. At this point, P(0) (namely H(0,0), H(0,1) and H(0,2)) are stored for the following computation (namely B(1,0), B(1,1) and B(1,2)).

B(1,0): B(1,0) is processed using H(0,0) as boundary conditions. Then, H(1,0) and V(1,0) are stored.

B(1,1): B(1,1) is processed using H(0,1) and V(1,0) as boundary conditions. H(1,1) and V(1,1) are stored.

B(1,2): B(1,2) is processed using H(0,2) and V(1,1). Then, only H(1,2) is stored. At this point, P(1) (namely H(1,0), H(1,1) and H(1,2)) are stored for the following computation (namely B(2,i)).

B(2,i): The B(2,i) is computed in the same way as B(1,i).

3 High Speed Computation with Two-Phase Search

Suppose that N is 1000, and w is 8. In order to obtain the optimal alignment by tracing back, we need to store

R_c**:** all traceback pointers to the final result, which amounts to $1000^3 \times 3b$ (3-bit is required for directions), and

R_s**:** V(i,j) and H(i,j) in Fig.4.

Thus, the factors that arise serious bottle-necks of time to be output both data (R_c and R_s) from an FPGA to external memory banks on a FPGA board, and time to transfer the data for the traceback pointers (R_c) from the external memory banks to the host computer.

In order to avoid these bottle-necks, two phase search is used in our approach.

1. In the first phase, data for H(i,j) are output to external memory banks (data for V(i,j) are cached on an FPGA). The performance of the system depends only on the size of the FPGA.
2. In the second phase, if the first phase gives scores which exceed user-defined threshold, the circuit for the second phase is reconfigured on the FPGA. In this phase, only H(i,j) and V(i,j) are sent to the host computer, and the optimal alignment is obtained by the co-processing the FPGA and software. Then, they are used in a reduction of the search space on the host computer.

3.1 The First Phase

Taking into account calculations of the optimal score, we need to output scores of the lattice sites on V(i,j) and H(i,j) in Fig.4 for each bar object. The size of V(i,j) and H(i,j) are both $1000 \times 8 \times 16b$ (no output for the connections). As shown in Fig.4, V(i,j) is used only for the calculation in the next bar object. It means that V(i,j) can be overwritten by V(i,j+1), thus, V(i,j) can be reduced to a simple plane V. Likewise, H(i,j) is used for the computation in the next row. It means that H(i,j) can be overwritten by H(i+1,j), thus, H(i,j) can be reduced to N/w planes. Therefore, we need $1 + N/w$ memory planes that are

1. one memory plane (called V) for V(i,j), plus
2. N/w memory planes (called H_j) for H(i,j).

As shown in Fig.5, V can be overwritten during the computation of the next bar object, while H_j can be overwritten during the computation of every N/w bar objects. One memory plane (V) can be configured with BRAMs in an FPGA. However, N/w memory planes (H_j) are too large to be implemented in the FPGA. Therefore, BRAMs can be used for V, while we need to use external memory banks for H_j.

The processing time for one bar object by 8×8 processing units is approximately 1000 ($8 \ll 1000$) clock cycles. Therefore, in order to achieve maximum performance, we need to output all scores in H_j within 1000 clock cycles. That is, we need to output eight 16-bit data at every clock cycle. Our FPGA board (ADM-XRC-II by Alpha Data) has eight external memory banks (including two memory banks on additional SRAM board). The data width of each memory bank on the FPGA board is 32 bit. In four memory banks, we can store eight 16-bit data at every clock cycle. Therefore, eight banks can be used to store and load the data for H_j (data for H(i,j) are stored to the four banks, while data for H(i-1,j) are read from another four banks).

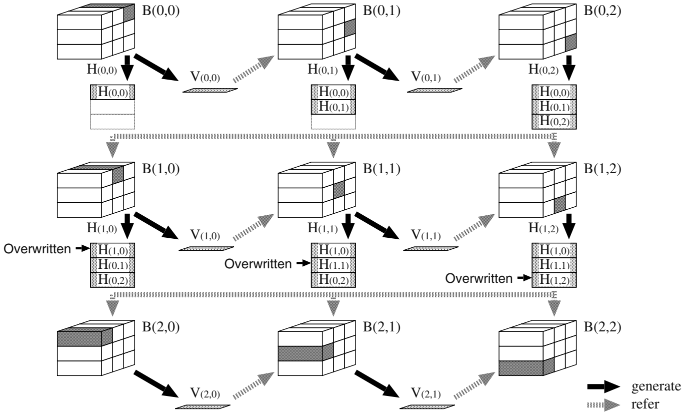


Fig. 5. Procedure in the first phase

3.2 The Second Phase

In order to store all traceback pointers which become necessary to obtain the optimal alignment, we need to reduce the speed of the processing units, because all of the eight memory banks are already used to store only H(i,j) at the speed of the processing units. In order to store all traceback pointers, the slow down is estimated to be no less than 1/3. Data transfer rate becomes another serious bottle-neck, because the data of the traceback pointers are too large to store in the FPGA board, and have to be sent to the host computer during the computation.

In the second phase, instead of storing the connections, data for V(i,j) as well as H(i,j) are stored in the external memory banks on the FPGA board. Fig.6

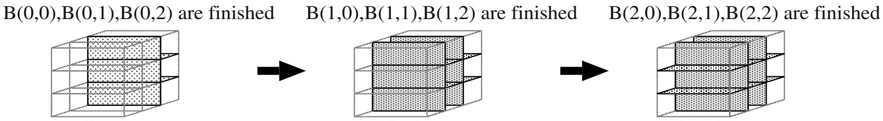

B(0,0),B(0,1),B(0,2) are finished
B(1,0),B(1,1),B(1,2) are finished
B(2,0),B(2,1),B(2,2) are finished

Fig. 6. Storing lattice shaped output data for software computation

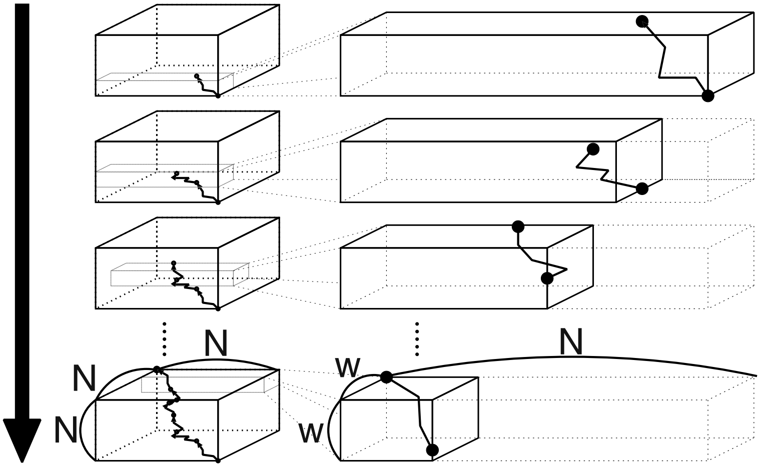


Fig. 7. Back tracing for the specification of the optimal alignment

illustrates H(i,j) and V(i,j) stored in the memory. Each of H(i,j) and V(i,j) is stored to different positions on the memory banks. The speed of the processing units has to be reduced to 1/2 in order to store H(i,j) and V(i,j).

Then, the data of H(i,j) and V(i,j) are sent to the host computer, and three dimensional DP is executed on the host computer. Here, we have a search space which was reduced by the procedure in this phase. Due to this reduction, the optimal alignment can be found efficiently. As shown in Fig.7, three dimensional DP is executed from the bar object containing the final site using the data as boundary conditions. Then, by tracing back the connections in the bar object, we can find a point of intersection on a surface of the bar object and the path of the optimal alignment. The next bar object which shares a plane including the intersection is chosen as next target, and three-dimensional DP is executed. In the host computer, this procedure is performed repeatedly until the path arrives at the starting site. This path corresponds to the optimal alignment.

By storing only a part of V(i,j) and H(i,j) (i mod k = 0 and j mod k = 0), we can reduce the amount of memory which have to be stored (and sent to the host computer), and achieve higher performance on the FPGA. However, this makes three-dimensional DP on the host computer slower. We need to decide value of k, taking account of the balance of the performance of the FPGA and the host computer.

4 Processing Unit

4.1 Optimization of the Payoff Matrix

Payoff matrix used in DP can be divided into two kinds of sub-matrices (one three-dimensional matrix for *xyz* in Fig.1(right) and Fig.2, and three two-dimensional matrices for *xy, yz* and *zx*). In order to achieve higher parallelism it is very important to minimize these payoff matrices.

We show a technique to reduce the size of the three-dimensional matrix as follows. Concerning the two-dimensional matrix, we have already discussed in [16]. The range of values in the matrices are -128 to $+127$ (8-bit). Therefore, a table of 32KBytes ($(2^5)^3 \times$ 8-bit) is required for a three-dimensional matrix, because three 5-bit characters are used to access the matrix of 8-bit width. Only 18 matrices can be implemented on a single XC2V6000 with this naive implementation.

The payoff, however, is not affected by the permutation of the three 5-bit characters. For example, each (A, L, C), (A, C, L), (C, A, L), (C, L, A), (L, A, C), and (L, C, A) gives the same payoff. In addition, only 24 characters are used in the sequences. Hence, the size of the matrix can be reduced from $(2^5)^3$ to ${}_{24}H_3$ if we can find a method to generate addresses to the minimized matrix from three 5-bit characters. That is why only 4KBytes are required for each table (on a single XC2V6000, we can implement up to 144 matrices).

The circuit for address generation is shown in Fig.8(upper left corner). In the circuit, two conversion tables are used to generate the optimal addresses from three 5-bit characters. We could not find a general algorithm to make the conversion tables at present, and the tables are designed by hand-coding. In the current implementation, 64 matrices are implemented on the FPGA, because we need to use some of BRAMs for other circuits.

4.2 Pipelined Processing

Fig.8 shows the details of the processing unit. As described in Section 2.1, there are seven traceback pointers toward each lattice site in the search space. In each processing unit, seven scores through those connections are calculated, and the maximum of them is chosen as the score of the site. The processing unit consists of 3 pipeline stages.

First Stage. A score is calculated using the three-dimensional payoff matrix and the score of previous node (x, y, z).

Second Stage. Three scores are calculated using two-dimensional payoff matrices, i.e. (seq.X, seq.Y, gap), (seq.X, gap, seq.Z), and (gap, seq.Y, seq.Z), and the maximum of the three scores and one score from the first stage is chosen. Each payoff matrix is also minimized[16]. The following equation shows Gap-cost in the second stage. In this equation, $S_{openingcost}$ and $S_{consecutivecost}$ are constants, and k is the number of consecutive gaps.

$$gapcost() = S_{opening} + S_{consecutive} \times k \tag{1}$$

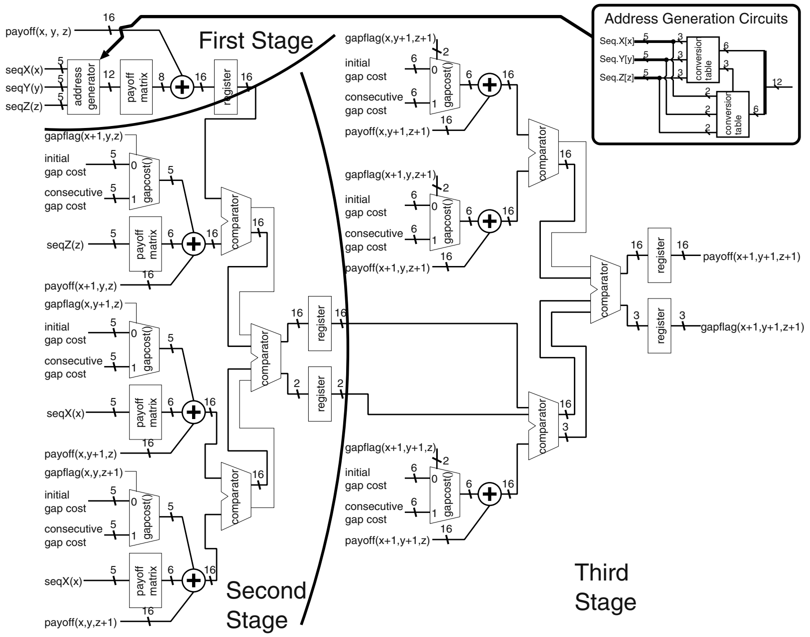


Fig. 8. Processing unit

Third Stage. Three scores are calculated, i.e. (seq.X, gap, gap), (gap, seq.Y, gap), and (gap, gap, seq.Z), and the maximum of the three scores and one score from the second stage is chosen as the output. The following equation shows Gap-cost in the third stage.

$$\begin{aligned} gapcost() = {}_0S_{opening} + {}_0S_{consecutive} \times {}_0Num_{consecutive_0} \\ + {}_1S_{opening} + {}_1S_{consecutive} \times {}_1Num_{consecutive_1} \end{aligned} \qquad (2)$$

5 Results

We implemented our method on an off-the-shelf FPGA board (ADM-XRC-II by Alpha Data with a single XC2V6000). Table1 shows computational time and speedup gain in the first phase as compared with a single Pentium4 2.53GHz. The number of processing units is 8 × 8, and they run at 66MHz. Almost 90% of hardware resources in the FPGA is used for the implementation. As shown in Table1, the performance gain in the first phase is 250 times higher.

Table 1. The comparison of performance between hardware and software (first phase)

N^3	P4 2.53GHz	XC2V6000	speedup gain
256^3	1.280s	0.005s	256
512^3	10.340s	0.034s	304
1024^3	83.720s	0.259s	327

In the second phase, the performance of the system depends on the balance of w, N, I/O data width, operating frequency, size of FPGA, data bandwidth between FPGA, and host computer, the ability of the host computer, and so on. These parameters have been tuned up for higher performance.

6 Conclusions

In this paper, we presented an approach for high speed computation of three-dimensional DP. The novelty of our approach is in the use of two-phase search with reconfigurations of an FPGA, co-processing the FPGA and the host computer, and optimization of all payoff matrices. It allows user to obtain the optimal result in reasonable time.

In the result, 64 units, which operate at 66MHz, can be implemented on a single XC2V6000. It achieves more than 200-fold speedup as compared with a desktop computer (Pentium4 2.53GHz, 2GB DDR-SDRAM).

The performance of the present system is limited by memory band-width of the FPGA board. Using an FPGA board with DDR memory banks and a single larger FPGA (those types of FPGA board are already on the market), the performance in the first phase can be dramatically improved. We estimate that the system with the FPGA board can achieve more than 1000-fold speedup.

This high performance makes it possible that the search time by three-dimensional DP is comparable with stochastic algorithms for three-dimensional alignment.

References

1. National Center for Biotechnology Information (NCBI), "NCBI-GenBank Flat File Release 137.0",
   http://www.ncbi.nlm.nih.gov/, Aug 2003.
2. European Molecular Biology Laboratory (EMBL), *http://www.ebi.ac.uk/embl/.*
3. DNA Data Bank of Japan (DDBJ), *http://www.ddbj.nig.ac.jp/.*
4. Saul B. Needleman, and Christian D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *Journal of Molecular Biology*, Vol.48, Issue 3, pp.443-453, 1970.
5. Temple F. Smith, and Michael S. Waterman, "Identification of common molecular subsequences", *Journal of Molecular Biology*, Vol.147, Issue 1, pp.195-197, Mar 1981.
6. Stephen F. Altschula, Warren Gisha, Webb Millerb, Eugene W. Meyersc, and David J. Lipman, "Basic Local Alignment Search Tool", *Journal of Molecular Biology*, Vol.215, Issue 3, pp.403-410, 1990.
7. Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller and David J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", *Nucleic Acids Research*, Vol.25, No.17, pp.3389-3402, 1997.
8. William R. Pearson and David J. Lipman "Improved tools for biological sequence comparison", *Proceedings of the National Academy of Sciences of the USA*, Vol.85, pp.2444-2448, 1988.

9. PARACEL, “GeneMatcher2”, *http://www.paracel.com/products/genematcher2.php.*
10. Dominique Lavenier, “SAMBA Systolic Accelerators For Molecular Biological Applications”, *Technical Report RR-2845*, 1996.
11. C. Thomas White, Raj K. Singh, Peter B. Reintjes, Jordan Lampe, Bruce W. Erickson, Wayne D. Dettloff, Vernon L. Chi, Stephen F. Altschul, “BioSCAN: A VLSI-Based System for Biosequence Analysis”, *IEEE International Conference on Computer Design: VLSI in Computer & Processors*, Vol.147, pp.504-509, (1991).
12. Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder, “Splash 2: FPGAs in a Custom Computing Machine”, Wiley-IEEE Press, 1996.
13. TimeLogic Corporation, “Decypher bioinformatics acceleration solution”, *http://www.timelogic.com/products.html*, 2002.
14. Kiran Puttegowda, William Worek, Nicholas Pappas, Anusha Dandapani, and Peter Athanas, “A Run-Time Reconfigurable System for Gene-Sequence Searching”, *International VLSI Design Conference*, pp.(to appear), 2003.
15. Steven A. Guccione and Eric Keller, “Gene Matching Using JBits”, *International Conferenece on Field-Programmable Logic and Applications*, pp.1168-1171, 2002.
16. Yoshiki Yamaguchi, Yosuke Miyajima, Tsutomu Maruyama, Akihiko Konagaya, “High Speed Homology Search with Run-time Reconfiguration”, *International Conferenece on Field-Programmable Logic and Applications*, pp.281-291, 2002.

An Instance-Specific Hardware Algorithm for Finding a Maximum Clique

Shin'ichi Wakabayashi and Kenji Kikuchi

Faculty of Information Sciences, Hiroshima City University
3-4-1, Ozuka-higashi, Asaminami-ku, Hiroshima 731-3194, Japan
wakaba@computer.org

Abstract. This paper presents a hardware algorithm for finding a maximum clique of a given graph, and shows experimental results of the proposed algorithm running on an FPGA. The proposed algorithm is constructed according to a given instance of graph, and can find a maximum clique efficiently based on branch and bound search. The proposed algorithm is designed to be implemented on FPGAs, and realizes an efficient branch and bound search with parallel and pipeline processing. Experimental results showed that, compared with the software solver, the proposed algorithm produced a maximum clique in a very shorter running time even if the time for circuit synthesis and configuration of FPGA was taken into account.

1 Introduction

Reconfigurable computing with Field Programmable Gate Arrays (FPGAs) has become popular as a new approach to combinatorial problems[4]. In particular, problem solving by instance-specific accelerators has been widely noticed such as the Boolean satisfiability problem, the minimum cover problem, etc[3,6,8]. In this approach, instance specific accelerators are generated on the fly, depending on problem instances rather than problems. It has been shown that they outperform software solvers, provided compilation time of the circuit description is kept short and the computation time is large.

In this paper, we focus on the problem of finding a maximum clique of an undirected graph. This problem is one of the important fundamental problems in graph theory, and it has various practical applications[1]. Since the maximum clique problem is an NP-hard problem, no efficient algorithms could be devised. For this reason, many approaches to solve the problem have been proposed, including branch & bound methods [5,7] and various meta-heuristics methods such as Simulated Annealing (SA) and Genetic Algorithm (GA). However, none of meta-heuristics can guarantee to generate an optimal solution. In addition, when exact methods are realized by software, an optimal solution cannot be obtained for a large problem within a practical computation time.

In this paper, we present a novel approach to the maximum clique problem based on reconfigurable computing. In the proposed method, for a given instance

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 516–525, 2004.

of the problem, an HDL description of an instance-specific accelerator is generated to produce an optimum solution of the problem. The proposed instance-specific accelerator is based on branch & bound search with various pruning techniques. Furthermore, pipeline and parallel processing are introduced to speed up the computation time. As far as we know, this is the first approach based on instance-specific reconfigurable computing to the maximum clique problem. Moreover, we develop a system, which generates the Verilog HDL description of the accelerator automatically for a given problem instance. The generated Verilog description is compiled and downloaded to an FPGA as configuration data to solve the maximum clique problem by hardware. Experimental results showed that, compared with the software solver `dfmax` [2], the proposed algorithm produced a maximum clique in a very shorter running time even if the time for circuit synthesis and configuration of FPGA was taken into account.

The remainder of this paper is organized as follows. After formulating the maximum clique problem in Section 2, in Section 3, we present a new hardware oriented algorithm to solve the maximum clique problem using FPGAs with instance-specific information. Section 4 explains the hardware implementation of the proposed method. Section 5 shows the experimental results, followed by a conclusion in Section 6.

2 Preliminaries

The maximum clique problem is defined as follows. Let $G = (V, E)$ be an arbitrary undirected graph, where V is a non-empty vertex set of G and $E \subseteq V \times V$ is an edge set of G. Let $n = |V|$ and $m = |E|$. For $G = (V, E)$, let $\Gamma(v) = \{w \in V | (v, w) \in E\}$ denote a set of vertices adjacent to vertex v in G. The number of vertices adjacent to vertex v, denoted $|\Gamma(v)|$, is called the (vertex) degree of v. In addition, we assume that for the vertex set V of G, the ordering of vertices is given. In the following, without loss of generality, we represent the i-th vertex in the set V as v_i $(1 \leq i \leq n)$. Given a vertex v, we denote the order of v as $ord(v)$. That is, if v is the i-th vertex in V, then $ord(v) = i$. Furthermore, for any subset of V, its ordering of vertices is also specified, that is inherited from the ordering of the original vertex set V.

For a subgraph $G(Q) = (Q, E(Q))$ of graph G is called an induced subgraph of G on $Q \subseteq V$, if $E(Q) = \{(v, w) \in E | v, w \in Q\}$. $G(Q)$ is called a *clique* of G if every two vertices are adjacent. A clique with the size being maximal in G or maximum is called a maximal or maximum clique. The number of vertices of a maximum clique of G is expressed with $\omega(G)$. This problem is an NP-hard problem, and no polynomial time algorithms could be devised.

3 The Algorithm

3.1 Overview

The proposed algorithm is based on branch & bound search. Overview of the algorithm is given below. As already noted, for an undirected graph $G = (V, E)$

with n vertices, we assume that the vertex set V is given a total order in advance, and the order of a vertex v is represented as $ord(v)$, $1 \leq ord(v) \leq n$. Let V_i be $\{v_i, v_{i+1}, v_{i+2}, \ldots, v_n\}$.

In the algorithm, the candidate vertex set, in which a maximum clique is currently searched, is stored in the array $R(*)$. For an induced subgraph $G(V_i)$, the algorithm of finding a maximum clique of $G(V_i)$ is given below.

Algorithm `mq` for finding a maximum clique in $G(V_i)$

```
Step0 (Initialize) $R(0) = V_i$, $Q = \emptyset$, $M_i = \emptyset$, $sp = 0$.
Step1 If $R(sp) = \emptyset$, then goto Step8. Otherwise, select a vertex $p \in R(sp)$ such
      that $ord(p)$ is minimum.
Step2 $R_p = R(sp) \cap \Gamma(p)$. Add $p$ to $Q$, and delete $p$ from $R(sp)$.
Step3 (Judgment) If $R_p = \emptyset$, or $R_p$ is a clique of $G(V_i)$, then goto Step6.
Step4 (Bound) If $R_p \cup Q$ contains no clique with size larger than $|M_i|$, goto
      Step7.
Step5 (Branch) Let $R(sp + 1) = R_p$, $sp = sp + 1$. Goto Step1.
Step6 (Update) If $|R_p \cup Q| > |M_i|$, then $M_i = R_p \cup Q$.
Step7 Eliminate a vertex $p$ such that $ord(p)$ is maximum.
Step8 (Terminate) If $sp = 0$, then terminate. Otherwise, $sp = sp - 1$, goto
      Step1.                                                                □
```

In the proposed algorithm, Q always contains a clique (not necessarily maximal or maximum) of $G(V_i)$, and any vertex in R_p is adjacent to all vertices in Q. Thus, when R_p becomes empty, or R_p becomes a clique (i.e., $G(R_p)$ is a complete graph), then $R_p \cup Q$ becomes a maximal clique of $G(V_i)$. The algorithm proceeds the search until a maximal clique is found in Step3, or it becomes clear that no maximum clique of $G(V_i)$ will be found in the current candidate vertex set. $R(sp)$ can be implemented as a stack, and sp is a stack pointer.

The total order of the vertex set affects the efficiency of the proposed algorithm, although any total order can be specified. In this paper, we use the total order defined as follows. First, we approximately solve the vertex coloring problem on the given graph. Then, the vertex set is partitioned into a set of classes so that each class consists of vertices with the same color. Finally, for each class, vertices are sorted according to their vertex degrees in the decreasing order.

In the following, we show the proposed algorithm, `mqr`, for finding a maximum clique in a graph G. In the algorithm, $c_{max}(i)$ represents the size of a maximum clique of the induced subgraph $G(V_i)$, and $c_{ub}(i)$ represents its upper bound.

Algorithm `mqr` for finding a maximum clique of G

```
Step0 (Initialize) $i = n$, $c_{ub}(n) = 1$, $M = \{v_n\}$.
Step1 $i = i - 1$. If $i = 0$, then terminate. Otherwise, if it isn't necessary to
      search for $G(V_i)$, then $c_{ub}(i) = c_{ub}(i + 1)$. Repeat Step1.
Step2 Using algorithm mq, a maximum clique $M_i$ of $G(V_i)$ is found.
Step3 If $c_{max}(i) > c_{ub}(i + 1)$, then $M = M_i$.
Step4 $c_{ub}(i) = c_{max}(i)$. Goto Step1.                                      □
```

3.2 Bounding Conditions

Effective vertex degree

The vertex degree of vertex v, denoted $d(v)$, is the number of vertices adjacent to v, that is, $d(v) = |\Gamma(v)|$. During the search of the proposed branch & bound method, the vertex set R, for which the search is performed to find a maximum clique, is called the candidate vertex set, and each element of the set is called the candidate vertex. The number of vertices in R adjacent to v is called the candidate vertex degree, denoted $d'(v)$.

For each candidate vertex v in R, let $\Gamma'(v)$ be a set of candidate vertices adjacent to v, and let F be a subset of $\Gamma'(v)$ such that no two vertices in F are adjacent to each other, that is, F is an independent set of $G(v \cup \Gamma'(v))$. Then, from the definition of a clique, at most one vertex in F can be an element of a maximum clique. Using this property, we can realize an efficient pruning of branch & bound search as follows. First, $\Gamma'(v)$ is partitioned into a set of vertex sets, F_0, F_1, ..., F_k, such that for each $i(1 \leq i \leq k)$, $|F_i| > 1$ and no two vertices in F_i is adjacent to each other, and F_0 consists of vertices not belonging to any $F_i(1 \leq i \leq k)$.

For each vertex v, the effective vertex degree of v, denoted $d_{\text{effective}}(v)$, is defined as follows.

$$\begin{aligned} d_{\text{effective}}(v) &= d'(v) - \sum_{i=1}^{k}(|F(i) - 1|) \\ &= |F_0| + k \end{aligned}$$

During the search, c_{max} represents the number of vertices of the maximal clique found so far. For each vertex v, if $d_{\text{effective}}(v) < c_{max}$ holds, then this vertex can be eliminated from the candidate vertex set R, since v will not become an element of a maximum clique.

[**Bounding condition 1**] For the candidate vertex set R, if there exists at most c_{max} candidate vertices in R whose effective vertex degree is larger than or equal to c_{max}, then there is no need for further search in R. □

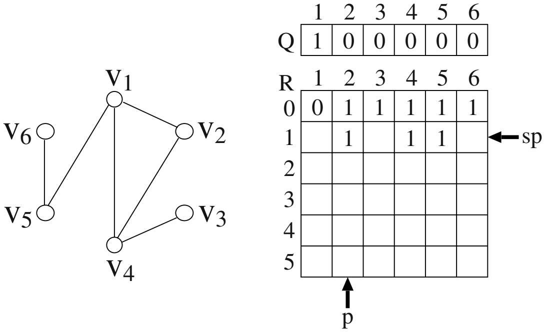
For each vertex v in R, if $d_{\text{effective}}(v) = |R| - 1$ holds, then R is a clique.

Note that the problem of finding $(k+1)$ vertex sets, $F_i(0 \leq i \leq k)$, is an NP-hard problem if we want to find an optimal solution of this partitioning problem. However, a good heuristic solution is enough to realize an efficient pruning in the proposed method. Thus, we construct them efficiently in a greedy manner. Due to the lace of space, we omit the details from here.

Bounding conditions on subgraphs

There are several cases that there is no need for further search in a subgraph of G.

[**Bounding condition 2**] In Step1 of the proposed method `mq`, vertex p is selected from the current candidate vertex set R, and R_p is defined as a next candidate vertex set. In this case, if there is another vertex q in R such that $R_q \subseteq R_p$ holds, then there is no need to search in R_q. □



	1	2	3	4	5	6
Q	1	0	0	0	0	0

R	1	2	3	4	5	6
0	0	1	1	1	1	1
1		1		1	1	
2						
3						
4						
5						

Fig. 1. Stack and the behavior of the method

[**Bounding condition 3**] For the candidate vertex set R, for each vertex $p \in R$, if $sp + c_{up}(ord(p)) \leq c_{max}$ holds, then there is no need to search in R_p. □

[**Bounding condition 4**] For vertex set V_i, assume that the maximum clique size of $G(V_i)$ is $c_{max}(i)$, and the maximum clique contains v_i. For any $v_j (j < i)$, if v_j has no connection with $v_k (j < k \leq i)$, then there is no need to search in $G(V_j)$. □

The bounding conditions 1, 2 and 3 are applied in Step 4 in algorithm `mq`, and the bounding condition 4 is applied in Step1 in algorithm `mqr`.

4 Hardware Implementation

4.1 Candidate Vertex Set

The proposed method will be implemented by hardware. The matrix $R(*)$ for storing the candidate vertex set is realized by a memory, whose memory word consists of $n = |V|$ bits. $R[a]_i$ represents the i-th bit of address a. In the proposed method, if $v_i \in R(a)$, then $R[a]_i = 1$, otherwise $R[a]_i = 0$. In the circuit, each bit in memory can be written with a binary value independently. Figure 1 shows the stack and the behavior of the proposed method.

4.2 Calculation of the Effective Vertex Degree

In the proposed method, it is important to calculate the effective vertex degree of each vertex in an efficient manner. In the following, using an example, we show how to calculate it by hardware. Assume that for vertex v, $\Gamma(v) = \{u_1, u_2, u_3, \ldots, u_8\}$, and let $F_0 = \{u_6, u_7, u_8\}$, $F_1 = \{u_1, u_2, u_3\}$, and $F_2 = \{u_4, u_5\}$, where F_1 and F_2 are non-adjacent vertex sets (i.e., independent sets of $G(v \cup \Gamma(v))$), respectively. For each u_i, if it is a candidate vertex, then $u_i = 1$, otherwise $u_i = 0$. Then, the effective vertex degree of v is calculated as

$$d_{\text{effective}}(v) = OR(u_1, u_2, u_3) + OR(u_4, u_5) + u_6 + u_7 + u_8,$$

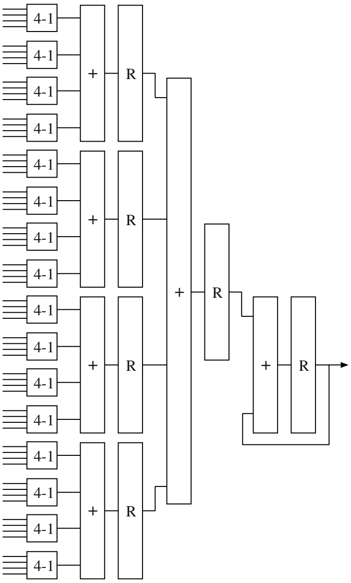


Fig. 2. A circuit for calculating an effective vertex degree.

where OR is a logical OR operation, and operation + is an arithmetic addition. Logical values of 0 and 1 are treated as binary numbers of 0 and 1 in arithmetic additions, and vice versa.

Figure 2 shows the circuit for calculating the effective vertex degree of a vertex, whose expression consists of 64 terms, each of which is either the output of an OR gate, or the output of the memory word, which represents that the corresponding vertex is either included in the current candidate vertex set or not. The entire circuit has a quad-tree structure to reduce the height of the adder tree. To shorten the critical path delay, registers are inserted in appropriate levels in the tree. Input signals are firstly put into 4-1 selectors, and then its output is fed to the adder tree. The output of the tree is fed to an accumulator. To reduce the whole computation time of calculating the effective vertex degree, the circuit realizes the pipeline processing, and registers inserted in the tree are used as pipeline registers. As a result, for the case of calculating the effective vertex degree shown in Figure 2, 6 clocks are required to generate the final result.

After calculating the effective vertex degree of each vertex in parallel, the result is compared with the maximum clique size currently hold in the circuit, and all vertices, whose effective vertex degree is smaller than the current maximum clique size, are eliminated from the current candidate vertex set in parallel. When some vertex is eliminated from the current candidate vertex set, then it may affect the effective vertex degree of another vertex. Thus, we repeat the calculation of effective vertex degrees until no update on the candidate vertex set is occurred. In general, a few times of repetition are required.

4.3 Search for the Starting Vertex

In the proposed method, in each step of branch, we have to find a vertex with the smallest value of $ord(p)$. In the straightforward implementation, it would require an $O(n)$ computation time, and this could cause an unacceptable degradation of the circuit performance. Thus, we introduce an mechanism, which is similar to the carry look-ahead in the adder, so that the length of the critical path of the circuit becomes $O(\log n)$.

4.4 Circuit Generation of Instance-Specific Hardware

We developed the system which generates an instance-specific HDL code from a design template and a given instance of the problem. The code generation system uses Verilog-HDL code templates, which describes the non-instance specific parts of the circuit, and adds this with a code generated specifically for the given instance, e.g., the circuit treating edge information among vertices. In this system, the adjacency list and the design template are given as inputs, and the HDL code of the circuit tailored to the given instance of the problem is produced as output. This system was developed with Perl. A generated HDL code is then given to the FPGA design tool to compile it. From an obtained compiled code (i.e., the gate level description of the circuit), place and route is performed to produce a configuration data of the target FPGA.

5 Evaluation

5.1 Experiments

The proposed method was implemented on an FPGA evaluation board to be evaluated. Environment of experiments was shown below. Verilog-HDL was used to describe the RTL description of the proposed method. A translation program from an instance of a graph and the template file to a circuit description was written with the Perl language. A translated code was then compiled and a configuration code was generated with the FPGA design tool Altera Quartus II version 3.0, which was executed on a PC (OS: Windows 2000, CPU: Pentium 4 2.4GHz). The FPGA board (Mitsubishi MYCOM Co. Ltd., PowerMedusa MU200AP1500) used in experiments consists of an FPGA, Altera EP20K1500E, which contains 2.39 million system gates. We also developed a simulation program of the proposed method with the C language, which simulates the behavior of the proposed method.

In the experiment, we used the benchmark software `dfmax`, which was published from DIMACS as a benchmark program for the maximum clique problem[2], and compared it with the proposed method. `dfmax` was written in the C language, and compiled with gcc 2.9.5 (O2 option was used), and executed on a PC with Pentium 4 2.4GHz and Linux 2.4.

As the benchmark data of the problem, we used the set of benchmark data published from DIMACS for the maximum clique problem. From the limitation

Table 1. Experimental results.

Data	n	$m(\rho)$	ω	`dfmax`[sec]	`mqr`[sec]	#LE	Tool[min]	#branch
brock200_1	200	14834 (0.745)	21	22.05	6.32	39593 (76%)	137	5218530
brock200_2	200	9876 (0.496)	12	0.04	0.007	28609 (55%)	91	9014
san200_0.7_1	200	13930 (0.7)	30	3738.69	1.08	33191 (64%)	97	11463
san200_0.7_2	200	13930 (0.7)	18	24192.73	0.02	24925 (48%)	73	47443
san200_0.9_1	200	17910 (0.9)	70	>24h	0.0018	47464 (91%)	148	62510
san200_0.9_2	200	17910 (0.9)	60	>24h	0.07	42302 (81%)	138	112205
san200_0.9_3	200	17910 (0.9)	44	63886.94	55	46445 (89%)	146	23157650
san400_0.7_1	400	55860 (0.7)	40	>24h				1611252
san400_0.7_2	400	55860 (0.7)	30	>24h				3790818

of the number of gates on FPGA, all the benchmark data implemented on FPGA consist of 200 vertices. For reference, we also used the benchmark data with 400 vertices, and for those data, we show the simulation results.

All the circuits of the proposed method were running on FPGA with a 40 MHz clock frequency. Execution time of the proposed method was measured by the hardware counter, which was embedded in hardware. The CPU time of the software program was measured by the `time` command of Linux. Execution time of the FPGA design tool was obtained from the tool. For each data to be solved on FPGA, additional time of translating the graph data into a Verilog code by the Perl program and downloading the configuration data to the FPGA chip was also required. For each data, this additional time in total was about 2 minutes on average, and at most less than 4 minutes.

5.2 Experimental Results

Table 1 show experimental results. In this table, n and m are the number of vertices and the number of edges of a graph, respectively, and ρ is the edge density of a graph($\rho = \frac{2\times m}{n\times(n-1)}$). ω is the size of a maximum clique. `dfmax` and `mqr` are the CPU time of the software solver `dfmax` and the execution time of the proposed method `mqr`, respectively. >24h means that no solution was obtained in 24 hours. #LE shows the number of logic elements of FPGA used for constructing the circuit. “Tool” shows the execution time of the FPGA design tool, including the compilation, placement, and routing. #branch shows the number of branches (i.e., the number of execution of step 5 in `mq`), which was obtained by the software simulation.

From the experimental results, the proposed method always produced the result in a shorter execution time compared with the software solver `dfmax`. In particular, for san200_0.7_2, san200_0.9_1, san200_0.9_2, and san200_0.9_3, the proposed method was faster, even if the compilation time was taken into account. From the simulation results for the data with 400 vertices (san400_0.7_1, san400_0.7_2), we predict that if we will use a larger FPGA, or we will implement the proposed method on multiple FPGAs, then the proposed method will be able to produce the results for those data in a short execution time (maybe, less than 10 seconds).

5.3 Discussion

In this section, we give some remarks and considerations on the proposed method. First, we consider the ability of finding a solution. From the results, for most data, the proposed method produced the better results in computation time than the existing method `dfmax`, and it shows that the proposed method realizes the efficient branch & bound search. However, for the data brock200_1, the efficiency of the proposed method was worth compared with the other cases. It shows that there may still be room for improvement of branch & bound search of the proposed method.

To improve the branch & bound search of the proposed method, we have to devise more effective methods of pruning the search. Another possibility of improvement is to introduce pipeline processing in search. For example, let u and v be different vertices in G. Then, for subgraphs $G(u \cup \Gamma(u))$ and $G(v \cup \Gamma(v))$, search for the maximum clique of those subgraph can be performed in parallel. Performance of this kind of parallel search may be improved if appropriate information such as the size of maximum clique of a subgraph is shared.

In the experiments, due to the size of an FPGA used in the experiment, graphs with more than 200 vertices were not used as benchmark data. However, as the rapid progress of semiconductor technology, we expect that FPGAs with 10 times larger number of gates, working with more than 200 MHz clock, will be available soon. Furthermore, if one FPGA is not enough to implement the circuit, multiple FPGAs may be used to implement the circuit. Thus, within a next few years, we believe that the maximum clique problem for any graphs, for which the exact solution can be obtained with the proposed branch & bound method in a practical time, will be solved on FPGAs.

Note that as the problem size becomes large, the advantage of the proposed method becomes large compared with the ordinary software solvers. The reason is as follows. As explained in Section 4, each step of branching in the proposed method requires $O(\log n)$ clock cycles. On the other hand, any state-of-the-art software solution proposed so far such as [5] and [7], each step of branching requires at least $O(n)$ time. Thus, the property above mentioned holds, even when the circuit compilation time is considered as the part of computation time of the proposed method.

Note also that, it is very difficult to implement the proposed method as an ordinary instance-independent circuit. For example, consider the circuit for calculating the effective vertex degree of of each vertex. The expression of the effective vertex degree of each vertex depends on a given instance of the graph. Thus, if we want to design a circuit for calculating the effective vertex degree for any graph, the circuit would become tremendously complicated, and even if it could be designed, the size of the circuit would be quite large. Nevertheless, it is very interesting to develop an instance-independent circuit for the maximum clique problem.

6 Conclusion

In this paper, we proposed a hardware algorithm for finding a maximum clique of a graph, and showed its effectiveness from experiments. In the proposed method, a circuit dedicated to an instance of the problem is automatically produced, and a maximum clique is found by branch & bound search. Experimental results showed that, compared with the software solver `dfmax` [2], the proposed algorithm produced a maximum clique in a very shorter running time even if the time for circuit synthesis and configuration of FPGA was taken into account.

As future work, we improve the proposed method so as to find a maximum clique in a shorter running time. Instead of using the design template of HDL code, using the hardware macros may be effective to reduce the circuit compilation time, and this is also interesting. To develop an instance-specific hardware algorithm for another combinatorial problems is also important.

References

1. Bomze, I. M., Budinich, M., Pardalos, P. M., Pelillo, M.: The maximum clique problem. Handbook of Combinatorial Optimization, Supplement Volume A, Kluwer Academic Publishers (1999) 1–74
2. The DIMACS Challenge, ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/
3. Platzner, M., Micheli, G. D.: Acceleration of satisfiability algorithms by reconfigurable hardware. Proc. 8th International Workshop on Field-Programmable Logic and Applications (1998) 69–78
4. Platzner, M.: Reconfigurable accelerators for combinatorial problems. IEEE Computer, **33**, 4 (2000) 58–60
5. Östergård, P. R. J.: A fast algorithm for the maximum clique problem. Discrete Applied Mathematics **120** (2002) 197–207
6. Suyama, T., Yokoo, M., Sawada, H.: Solving satisfiability problems on FPGAs. Proc. 6th International Workshop on Field-Programmable Logic and Applications (FPL'96) (1996) 136–145
7. Seki, T., Tomita, E.: Efficient branch-and-bound algorithms for finding a maximum clique. Technical Report of the Institute of Electronics, Information and Communication Engineers, COMP2001-50 (2001) 101–108, in Japanese
8. Zhong, P., Martonosi, M., Ashar, P., Malik, S.: Using configurable computing to accelerate Boolean satisfiability. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, **18**, 6 (1999) 861–868

IP Generation for an FPGA-Based Audio DAC Sigma-Delta Converter

Ralf Ludewig[1], Oliver Soffke[1], Peter Zipf[1], Manfred Glesner[1],
Kong Pang Pun[2], Kuen Hung Tsoi[2], Kin Hong Lee[2], and Philip Leong[2]

[1] Institute of Microelectronic Systems
Darmstadt University of Technology, Germany.
{ludewig,soffke,zipf,glesner}@mes.tu-darmstadt.de
[2] Department of Computer Science and Engineering
Chinese University of Hong Kong, Shatin NT HK.
{kppun,khtsoi,khlee,phwl}@cse.cuhk.edu.hk

Abstract. In this paper we describe a parameterizable FPGA-based implementation of a sigma-delta converter used in a 96kHz audio DAC. From specifications of the converter's input bitwidth and data sampling frequency, VHDL generic parameters are used to automatically generate the required design. The resulting implementation is optimized to use the minimum internal wordlength and number of stages. We prototyped the converter on an FPGA board for verification purposes and the results are presented.

1 Introduction

With recent improvements in the density of field programmable gate array (FPGA) devices, systems with an increasingly higher level of integration are possible. Digital signal processing is an important application area for FPGAs and such systems often require data converters to provide analog outputs from digital domain representations. Sigma-delta modulator based digital to analog converters (DAC) can be efficiently implemented on FPGA devices since they are entirely digital. Including such converters in the FPGA provides a high degree of flexibility and reduces costs and time to market. Moreover, it becomes possible to develop single chip, customized converters which are not available commercially, e.g. an application may require a mix of different converters of different orders operating at different frequencies. Including such converters into the FPGA provides a high degree of flexibility and reduces costs and time-to-market. In this paper, a flexible sigma-delta converter for an audio frequency DAC as proposed in [1] is described. It is modeled as a parameterizable IP core description which can be used to generate the actual implementation of converters accepting 16-, 20-, and 24-bit input values at data sampling rates between 32 and 96 kHz. Compared with the design in [1] which was made in Handel-C, the IP core presented in this paper is more modular, automatically determines

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 526–535, 2004.

the required internal bitwidths to avoid overflow, is written entirely in VHDL and is approximately two times faster yet occupies less resources.

The paper is structured as follows: Section 2 describes the basic principle of operation of $\Sigma\Delta$-converters. The architecture of the implemented DAC is discussed in Section 3. Section 4 deals with the system-level simulation of the model and in Section 5 the experimental results of an implemented prototype are shown. Finally, we end up with some conclusions in Section 6.

2 Basics of $\Sigma\Delta$-Converters

The basic task of a $\Sigma\Delta$-converter is to quantize the input signal, which can be either continuous or discrete in time. This applies for both the analog-to-digital and the digital-to-analog conversions. 1-bit quantization is most popular for simple circuit design [2]. The one-bit signal is perfectly suited as the input or output of a digital system.

The quantization with one bit resolution can be achieved without significant quality loss, by oversampling the signal by a factor of M and shifting the quantization noise to high frequencies where no signal frequency components are present. This is usually referred to as *noise shaping.*

The signal and the noise can then be separated with a lowpass filter of the appropriate order. This lowpass works in the digital domain for analog-to-digital converters and in the analog domain for digital-to-analog converters. As this paper deals with a digital-to-analog converter we will focus our considerations on this type, but the basic theory can be applied for both types.

Figure 1 shows a simple $\Sigma\Delta$-converter which accepts a discrete time, continuous value input signal and outputs a discrete time, discrete value output signal with one bit resolution shifting the quantization noise to high frequencies. For the analysis of this $\Sigma\Delta$-converter a linearized model can be constructed as shown in Figure 1(b) by replacing the quantizer by an injection of additive noise. The output $Y(z)$ is the superposition of the input signal $X(z)$ transformed by the system and the noise signal $N(z)$ also transformed by the system:

$$Y(z) = H_x(z)X(z) + H_n(z)N(z)\,, \tag{1}$$

where $H_x(z)$ denotes the signal transfer function and $H_n(z)$ denotes the noise transfer function, which can be derived from

$$H_x(z) = \left.\frac{Y(z)}{X(z)}\right|_{N(z)=0} \tag{2}$$

$$H_n(z) = \left.\frac{Y(z)}{N(z)}\right|_{X(z)=0} \tag{3}$$

This is depicted in Figure 2. From figure 2a we find that

$$Y(z) = H(z)\Big(X(z) - Y(z)\Big)\,, \tag{4}$$

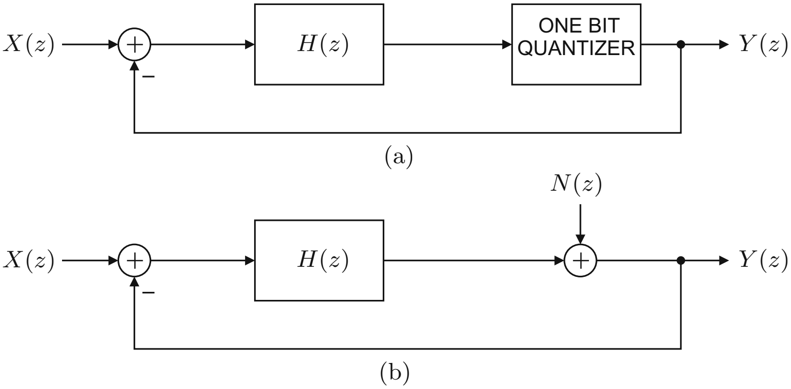


Fig. 1. Simple discrete time $\Sigma\Delta$-converter quantizing continuous value input signals with one bit resolution (a) and a system theoretic model of it with the quantizer replaced by additive noise injection (b).

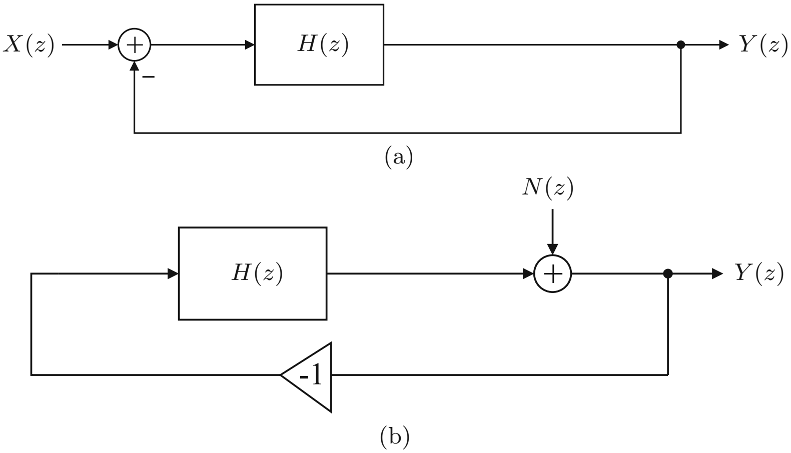


Fig. 2. Models for deriving the signal transfer function $H_x(z)$ (a) and the noise transfer function $H_n(z)$ (b).

which finally yields the signal transfer function $H_x(z)$:

$$H_x(z) = \frac{Y(z)}{X(z)} = \frac{H(z)}{1 + H(z)} \,. \tag{5}$$

Using the same procedure we also find the noise transfer function $H_n(z)$ from figure 2b:

$$Y(z) = N(z) - H(z)Y(z) \tag{6}$$

$$\Rightarrow H_n(z) = \frac{Y(z)}{N(z)} = \frac{1}{1 + H(z)} \,. \tag{7}$$

An integrator is used for $H(z)$ to implement the first order noise shaping. To avoid an algebraic loop due to the feedback in the $\Sigma\Delta$-converter, the integrator is chosen to have no direct feedthrough. Thus, it can be described in the time domain by

$$y_n = y_{n-1} + x_{n-1}\,. \tag{8}$$

Transforming this difference equation in the z-domain yields the transfer function $H(z)$:

$$Y(z) = z^{-1}Y(z) + z^{-1}X(z) \tag{9}$$

$$\Rightarrow H(z) = \frac{Y(z)}{X(z)} = \frac{z^{-1}}{1-z^{-1}}\,. \tag{10}$$

This finally gives the $\Sigma\Delta$-converter depicted in Figure 3 with the transfer functions for the signal and the noise:

$$H_x(z) = \frac{\frac{z^{-1}}{1-z^{-1}}}{1+\frac{z^{-1}}{1-z^{-1}}} = z^{-1} \tag{11}$$

$$H_n(z) = \frac{1}{1+\frac{z^{-1}}{1-z^{-1}}} = 1 - z^{-1}\,. \tag{12}$$

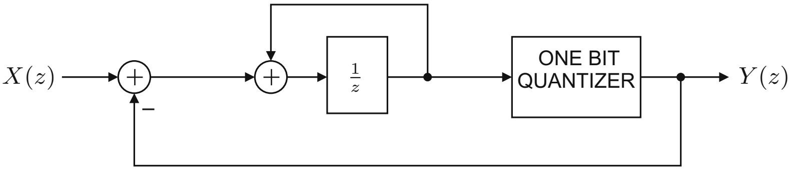


Fig. 3. $\Sigma\Delta$-converter with $H(z) = \frac{z^{-1}}{1-z^{-1}}$.

These transfer functions are depicted in Figure 4. As intended, the quantization noise is shifted to high frequencies, so that an analog lowpass at the output will suppress this noise and reconstruct the input signal.

Note, that the noise transfer function $H_n(z)$ is a first order highpass. Thus, the slope will be only 20 dB per decade. This can be improved furthermore by increasing the order of the $\Sigma\Delta$-converter as outlined in the next section.

3 Implementation of $\Sigma\Delta$-Converter

The $\Sigma\Delta$-converter can be used within a complete audio DAC as proposed in [1]. The structure of such a DAC is shown in Figure 5. It consists of the interpolator, the sigma-delta modulator, and the 1-bit DAC. The audio DAC accepts PCM

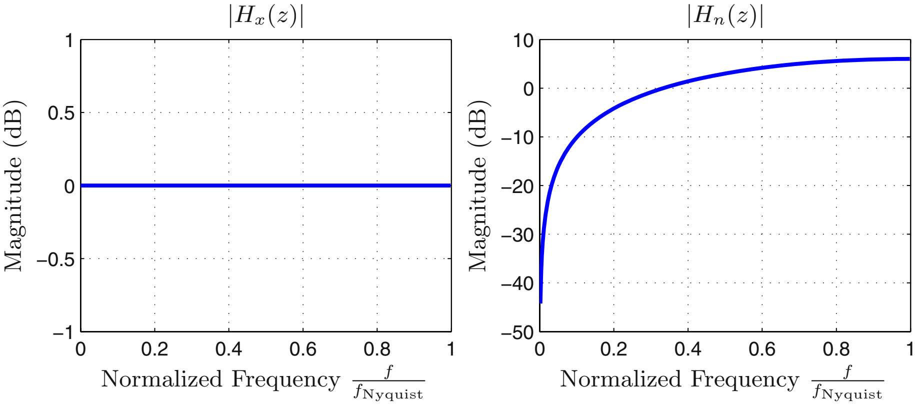


Fig. 4. Signal and noise transfer function of the $\Sigma\Delta$-converter depicted in Figure 3. As intended, the quantization noise is shifted to high frequencies.

input data at sampling rates of 32/44.1/48/88.2/96 kHz. The interpolation ratio of the interpolator can be configured to 64x, 128x, and 192x. For 44.1/88.2 kHz input signals, the interpolator gives the output data rate of 5.6448 MHz by setting the interpolation ratio as 128x/64x respectively. For 21/48/96 kHz input signals, the interpolator gives the output data rate of 6.144 MHz by setting the interpolation ratio as 192x/128x/64x respectively.

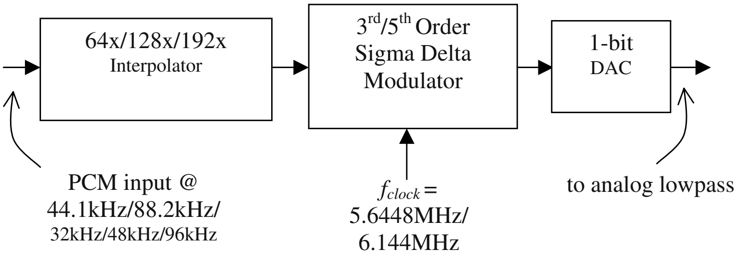


Fig. 5. Block diagram of the audio DAC.

The configurable $\Sigma\Delta$-converter proposed in this paper can be freely configured and can be used for the $\Sigma\Delta$-converter of the complete audio DAC. Our approach was to create a soft core which is based on a VHDL description that can be configured to produce a $\Sigma\Delta$-converter of arbitrary order.

As a starting point we used the architecture (see Figure 6) as proposed [1] for the $\Sigma\Delta$-converter which can be configured as either 3^{rd} order and as 5^{th} order modulator. The modulator coefficients were designed by using a Matlab toolbox [3]. As it can be clearly seen the modulator is composed of two basic blocks

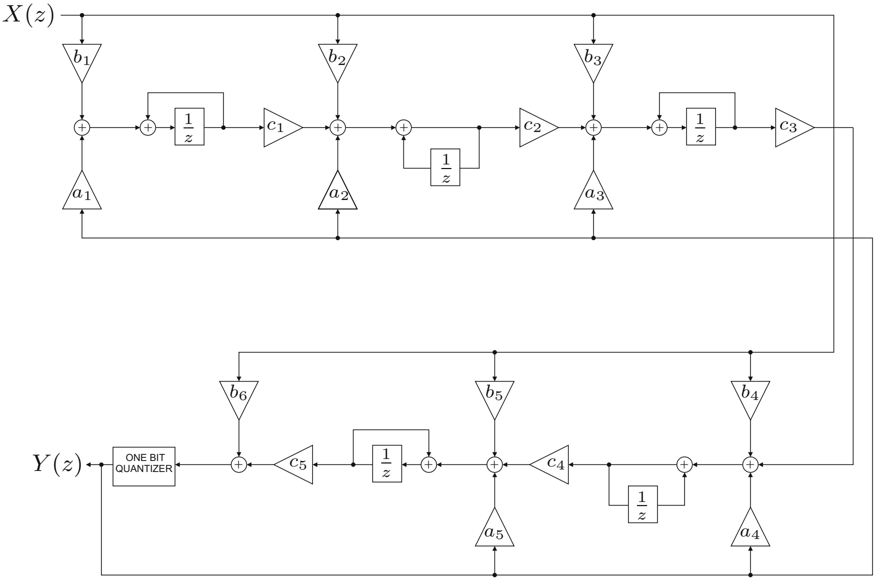


Fig. 6. Architecture of the $\Sigma\Delta$-converter. The stages are generated using a `for-generate`-loop, so the number of stages can be adapted to the desired signal-to-noise-ratio.

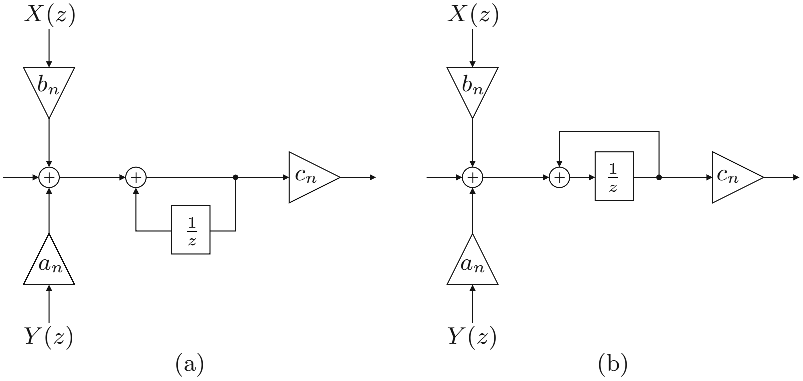


Fig. 7. Basic stages of $\Sigma\Delta$-converter

(see Figure 7). Each of the two basic blocks is composed of a register for the delay and 3 multipliers. In VHDL, the constants a_n, b_n, c_n can be specified as a generic for every block separately. Everything is combined to the final architecture in the top-level design that uses a generate-loop to create a $\Sigma\Delta$-converter of the specified order.

While the system level simulation uses floating point numbers, a fixpoint implementation had to be derived for the FPGA realization. For a $\Sigma\Delta$-converter with a bitwidth of n at the input, the parameters of the stages have to be scaled by $2^{n-1} - 1$. Due to the integrating nature of the delay stages, it is not sufficient to use n as internal bitwidth. So the number of bits in the middle section of each stage (adder, delay and multiplication with c_n) must to be increased to avoid overflows.

As the VHDL description of the model is fully generic, a $\Sigma\Delta$-converter IP-core of the desired order can be generated very easily by changing the value of the generics. The design parameters like the number of stages, the fixpoint bitwidth, the internal bitwidth and the converter coefficients are specified in a single configuration file.

4 System-Level Simulation

System level simulations were conducted using Matlab Simulink. Figure 8(a) shows the output spectrum for the 5th order configuration of the DAC, given a 20.48 kHz sinusoidal input signal (at the upper edge of the audio band), sampled at 96 kHz. It is then interpolated to a 6.144 MHz sampling rate using a two-stage sinc interpolator. The quantized 1-bit output of the sigma-delta DAC is then passed through an FFT to obtain the output frequency spectrum. The fifth-order noise shaping function is clearly observed. The audio-band SNR for this simulation is 138 dB. Figure 8(b) shows the output SNR versus the input signal level. From this figure, a maximum SNR of 140dB is obtained from the formula:

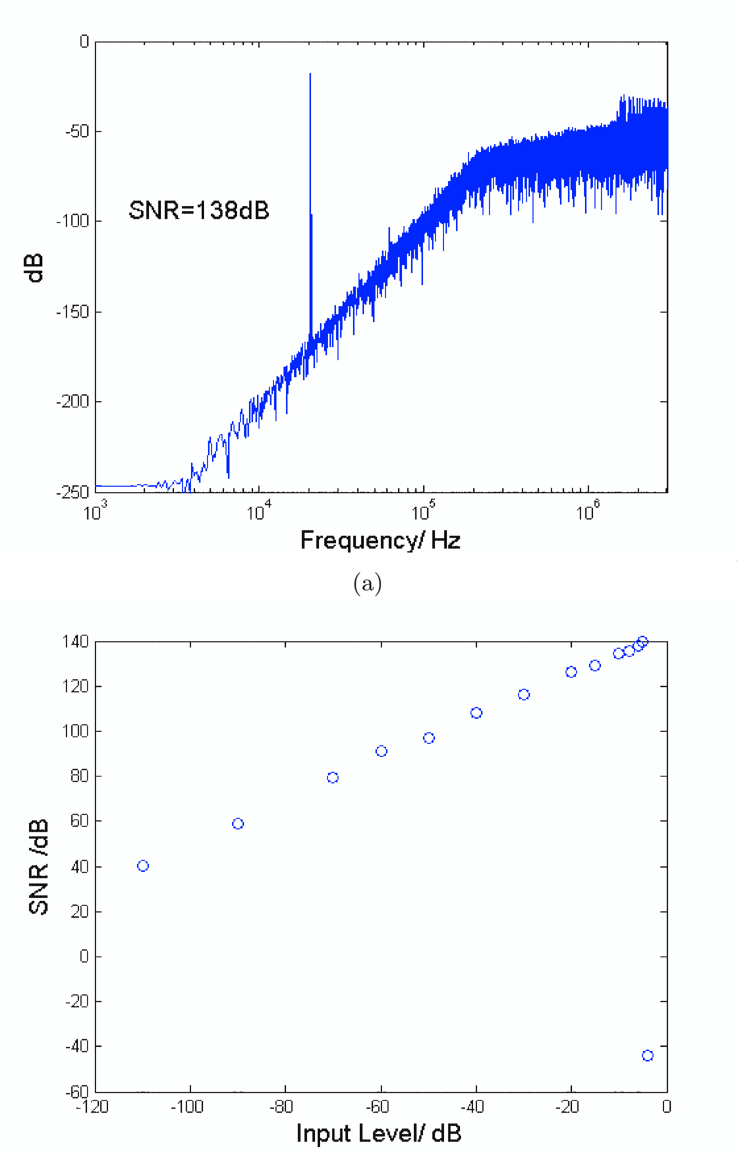
$$\text{Effective number of bits (ENOB)} = \frac{SNR - 1.76}{6.02},$$

so this DAC configuration can operate on inputs with word lengths of up to 23 bits. The SNR performance versus input SNR for the 3rd order configuration of the DAC is shown in Figure 9. The maximum SNR is 96.4 dB, which corresponds to an ENOB of 15.7 bits. The system-level simulations verify the correctness of the DAC architecture.

5 Experimental Results

The complete design has been prototyped using an FPGA board with a XILINX XC2V1000-4 Virtex II FPGA. In order to produce some reasonable output, a 24-bit look up table containing one period of a 10 kHz sine wave sampled at 6 MHz has also been mapped on the FPGA to provide a periodic input signal for the $\Sigma\Delta$-converter. In some designs (like in [1]) the c_n parameters of the $\Sigma\Delta$-converter are set to one so that the corresponding multipliers can be omitted. We have optimized our generic VHDL model so that the multipliers with a coefficient of one are automatically left out.

Table 1 shows the implementation details of our design. For a comparison with the results from the Handel-C implementation of [1] the first two values show the hardware resources and the maximum clock frequency for a coefficient set with $c_n = 1$. The other values are for a $\Sigma\Delta$-converter with all coefficients not equal to 1.



(a)

(b)

Fig. 8. System level simulation results for the 5th order configuration of the DAC. (a) output spectrum (b) output SNR versus input level.

It can be clearly seen that all the implementations meet the required 6 MHz operating frequency. Furthermore, using the $c_n = 1$ case for comparison with [1], this design uses the multipliers of the Virtex II device in order to achieve a

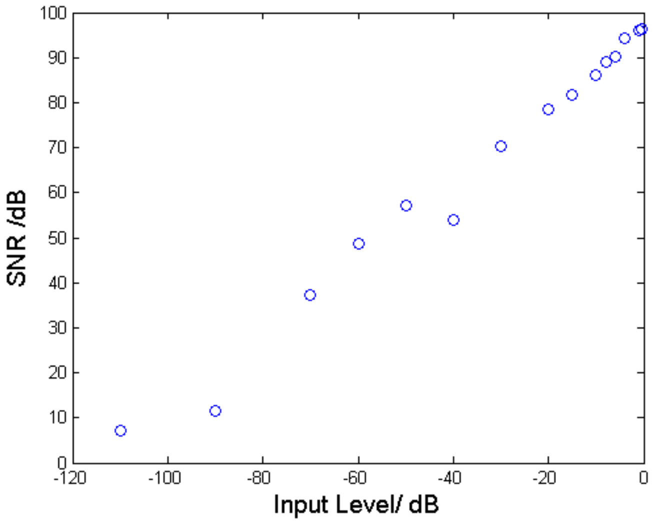


Fig. 9. Output SNR versus input level for the 3rd order configuration of the DAC.

Table 1. Details of the implementation of a 5^{th} order $\Sigma\Delta$-converter with 24 bit input

Design	Slices	Multipliers	max. Clock
5^{th} order SD ($c_n = 1$)	362/5120	20/40	51 MHz
5^{th} order SD with LUT ($c_n = 1$)	967/5120	20/40	27 MHz
5^{th} order SD ($c_n \neq 1$)	566/5120	39/40	19 MHz
5^{th} order SD with LUT ($c_n \neq 1$)	1163/5120	35/40	16 MHz

large reduction in slice utilization (362 compared with 3167) as well as a higher operating frequency (51.7 MHz compared with 27 MHz). Even in the case of $c_n \neq 1$, only a modest number of slices are required. When synthesizing the design without the multipliers of the Virtex II device we also achive a lower slice utilization of 2757 compared to 3167 of the Handel-C approach (for $c_n \neq 1$).

To validate our implementation the spectrum of the output signal of the VHDL implementation was compared with the spectrum generated by the system level simulation. Furthermore we connected an analog low pass filter to the FPGA output and observed a very smooth and stable sine wave with a frequency of exactly 10 kHz.

6 Conclusions and Future Work

A $\Sigma\Delta$-converter-IP-core has been presented, that can be adapted easily to different requirements. This adaptation includes the selection of the bitwidth of the input signal and of the internal signals, the configuration of the coefficients and the $\Sigma\Delta$-converter core can be generated with an arbitrary number of stages. Additionally we included some automatic optimizations to remove unnecessary hardware (like a multiplication with one). With our VHDL $\Sigma\Delta$-converter-core we achieved much better results than with the Handel-C implementation.

The $\Sigma\Delta$-converter has been studied using simulink and both the system level simulation and the RT-level simulation results have been presented. The model has been prototyped on an FPGA board and the reconstructed sine wave generated by a LUT which has been also mapped onto the FPGA, could be observed using an simple analog low pass filter.

The next step is to replace the LUT with an interpolator in order to provide the required oversampling. Using an interpolator the output of some conventional digital signal processing block (e.g. a MP3 decoder or a S/PDIF-receiver) can be used as input of the interpolator. Then it will be possible to listen to the output of the $\Sigma\Delta$-converter and judge the quality subjectively.

References

1. Ray C.C. Cheung and K.P. Pun and Steve C.L. Yuen and K.H. Tsoi and Philip H.W. Leong *An FPGA-based Re-configurable 24-bit 96kHz Sigma-Delta Audio DAC.* FPT 2003, Tokyo, Japan.
2. Gabor C. Temes and Shaofeng Shu and Richard Schreier *Architecture for Sigma Delta DACs*, in Delta Sigma Data Converters, edited by S.R. Norsworthy, et al, IEEE Press, 1997.
3. Richard Schreier, *http://www.mathworks.com*, MATLAB Central > File Exchange > Controls and Systems Modeling > Control Design > delsig.

Automatic Creation of Reconfigurable PALs/PLAs for SoC

Mark Holland and Scott Hauck

Department of Electrical Engineering
University of Washington, Seattle, WA 98195, USA
{mholland, hauck}@ee.washington.edu

Abstract. Many System-on-a-Chip devices would benefit from the inclusion of reprogrammable logic on the silicon die, as it can add general computing ability, provide run-time reconfigurability, or even be used for post-fabrication modifications. Also, by catering the logic to the SoC domain, additional area/delay/power gains can be achieved over current, more general reconfigurable fabrics. This paper presents tools that automate the creation of domain specific PLAs and PALs for SoC, including an Architecture Generator for making optimized arrays and a Layout Generator for creating efficient layouts. By intelligently mapping netlists to PLA and PAL arrays, we can reduce 60%-70% of the programmable connections in the array, creating delay gains of 15%-30% over unoptimized arrays.

1 Introduction

As device scaling continues to follow Moore's Law, chip designers are finding themselves with more and more real estate to work with. This is true in the design realm of System-on-a-Chip (SoC), where individual, pre-designed subsystems (memories, processors, DSPs, etc.) are integrated together on a single piece of silicon in order to make a larger device. Moore's Law can be seen as providing us with more silicon space to use, and one solution for SoC designers is to add reconfigurable logic to their devices.

Reconfigurable logic fills a useful niche between the flexibility provided by a processor and the performance provided by custom hardware. Traditional FPGAs, however, provide this flexibility at the cost of increased area, delay, and power. As such, it would be useful to tailor the reconfigurable logic to a user specified domain in order to reduce the unneeded flexibility, thereby reducing the area, delay, and power penalties that it suffers. The dilemma then becomes creating these domain specific reconfigurable fabrics in a short enough time that they can be useful to SoC designers.

The Totem project is our attempt to reduce the amount of effort and time that goes into the process of designing domain specific reconfigurable logic. By automating the generation process, we will be able to accept a domain description and quickly return a reconfigurable architecture that targets that domain.

Previous work in Totem has focused on using a 1-D RaPiD array [1-4] in order to provide reconfigurable architectures for domains that use ALUs, Multipliers, RAMs, and other coarse grained units. But many domains do not benefit from the use of

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 536–545, 2004.

coarse-grained units, and would require a finer-grained fabric. An appealing solution for these domains would be to create reconfigurable PALs and PLAs, which are efficient at representing seemingly random or non-regular logic functions. Providing SoC designers with reconfigurable PALs and PLAs will give them the ability to support many different fine-grained functions, perform run-time reconfiguration, or even perform bug fixes or other post-fabrication modifications.

2 Background

Reconfigurable PALs and PLAs have existed for many years in commercially available CPLDs, and are produced by companies including Xilinx, Altera, and Lattice. CPLDs are typically reprogrammable PALs or PLAs that are connected by fairly rich programmable routing fabrics, with other hardware added to increase the functionality of the devices. These commercial CPLDs, however, suffer from the same drawbacks as commercial FPGAs: their generality, while allowing them to be used in many domains, costs them in terms of area, delay, and power.

Additionally, many papers have been published with respect to PAL and PLA architectures. The most applicable of these was by A. Yan and S. Wilton [5]. In this paper they explore the development of "soft" or synthesizable programmable logic cores based on PLAs, which they call product term arrays. In their process they acquire the high-level requirements of a design (# of inputs, # of outputs, gate count) and then create a hardware description language (HDL) representation of a programmable core that will satisfy the requirements. This HDL description is then given to the SoC designer so that they can use the same synthesis tools in creating the programmable core that they use to create other parts of their chip.

Their soft programmable core has the advantages of easy integration into the ASIC flow, and it will allow users to closely integrate this programmable logic with other parts of the chip. The core will likely be made out of standard cells, however, whose inefficiency will cause significant performance penalties. As such, using these soft cores only makes sense if the amount of programmable logic required is small.

Our process differs from [5] in that we will create and provide domain-specific "hard" cores to be used in SoC. Our tools will intelligently create a PLA or PAL architecture that fits the specifications provided by the designer, and it will then use pre-optimized layouts to create a small, fast, low-power layout of the array that can be immediately placed onto the SoC. This will result in area, delay, and power improvements over pre-made programmable cores or soft cores, and it is automated in order to provide very fast turnaround time.

Another related work presents highly regular structures which provide ease of design and layout of PLAs for possible use in SoC [6]. Their proposal is to stack multiple PLAs in a uni-directional structure using river routing to connect them together, resulting in a structure that benefits from both high circuit regularity and predictable area and delay formulation. Their focus on circuit regularity and area/delay predictability prevent them from obtaining high performance, however. Our arrays will be tailored to the specifications of the designer, allowing us to both better suit their exact needs and to make modifications that will result in better area, delay, and power performance.

In regards to software, much work has been done on the concept of minimizing sum-of-products style equations so that they will require smaller PLA or PAL arrays for their implementation. To date, the most successful algorithm for minimizing these equations is ESPRESSO, which was developed at Berkeley in the 1980s [7]. ESPRESSO's basic strategy is to iteratively expand its terms (in order to encompass and remove other terms) and then reduce its terms (to prepare for a new, different expansion). This expand and reduce methodology results in a final equation that has a near-optimal number of product terms. ESPRESSO also reduces the number of literals in the equation, which is equivalent to minimizing the number of connections that need to be made in the PLA or PAL array.

3 Approach – Tool Flow

We envision the tool flow to be as follows. The input from the customer will be a specification of the target domain, most likely containing a set of netlists that the architecture will need to support. These netlists will first be processed by ESPRESSO in order to minimize the number of product terms and literals that they contain.

The resulting minimized netlists will be fed into the Architecture Generator, which attempts to create the smallest single PLA or PAL array that is capable of being configured to every netlist. Only one netlist must be supported at any given time. The Architecture Generator will then output a description of this array.

The array specification from the Architecture Generator is fed to the Layout Generator, which creates a full layout of the array. This layout includes the PAL/PLA array as well as all the remaining logic that is necessary for programming the array.

3.1 Architecture Generator

The Architecture Generator must read in multiple netlists and create a PAL/PLA array capable of supporting all of the netlists. The tool is written in C++.

The goal of the Architecture Generator is to map all the netlists into an array that is of minimum size and which has as few programmable connections as are necessary. For a PLA, minimizing the number of inputs, outputs, and product terms in the array is actually trivial, as each of them is simply the maximum occurrence seen across the set of netlists. For a PAL we minimize the number of inputs and outputs the same way as for a PLA, and we minimize the number of product terms in the array by making each output OR gate as small as is possible.

Having minimized the number of inputs, outputs, and product terms in the array, the next goal is to minimize the number of programmable connections that are necessary in order to support each netlist. Figure 1 displays the problem that we face when trying to minimize the number of programmable connections. In this example we are trying to fit two netlists (one grey and one black) onto the same array. A random mapping of the product terms is shown to require 16 programmable connections, while an intelligent mapping is shown to require only 8 programmable connections - a 50% reduction.

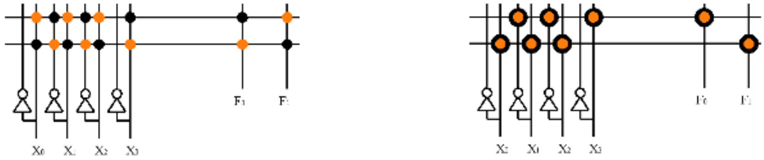


Fig. 1. On the left, two netlists are mapped randomly. This requires 16 programmable bits. On the right, they are mapped intelligently, requiring only 8 programming bits

For our application, simulated annealing has proven to be very successful at mapping netlists to an array. The algorithm's goal is to minimize the number of programmable connections. We define a basic "move" as being the swapping of two product term rows within a netlist (we will introduce more complicated moves later), and the "cost" of a mapping is the number of programmable bits that it requires. For our annealing we use the temperature schedules published in [8].

The development of a cost function requires serious consideration, as it will be the only way in which the annealer can measure netlist placements. At first glance, the cost of an array location is ideally either a 1 (programmable bit present) or a 0 (programmable bit not present). Using a simple 1/0 cost function, however, would hide a lot of useful information from the annealer. The degree to which a programmable bit is required (how many netlists are using it) is also useful information, as it can tell the annealer how close we are to removing a programmable connection.

Figure 2 displays this notion. In this example, we have one programmable connection used by two netlists and another connection used by five netlists. Both locations require a programmable bit, but it would be much wiser to move to situation A) than to situation B), because situation A) brings us closer to freeing up a connection.

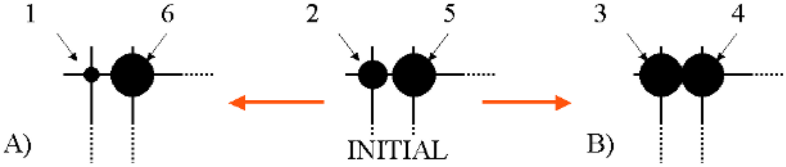


Fig. 2. A move to A) puts us closer to removing a programming bit than a move to B)

The cost function that we developed captures this subtlety by adding diminishing costs to each netlist that uses a programmable connection. If only one netlist is using a connection the cost is 1; if two netlists use a connection it costs 1.5; three netlists is 1.75, then 1.875, 1.9375, and so on. Referring back to Figure 2 and using this cost function, a move to figure A has a Δcost of -.45 (a good move) while a move to figure B has a Δcost of .19, which is a bad move. Mathematically, the cost function is:

$$\text{COST} = 2 - .5(x - 1)$$

where x is the number of netlists that are using a position. As seen, each additional netlist that loads a position incurs a decreasing cost, such that going from 7 to 8 is much cheaper than going from 1 to 2, for example.

Because PALs and PLAs are structurally different, we need an annealing algorithm that can work on both types of arrays. Additionally, we don't know what hardware might exist on the SoC at the periphery of our arrays. The existence of crossbars at the inputs and outputs to our arrays would allow us to permute the input and output locations between netlist mappings. Considering this, we are presented with a need for four annealing scenarios: using a PLA with fixed IO positions, using a PLA with variable IO positions, using a PAL with fixed IO positions, and using a PAL with variable IO positions.

The differences between the annealing scenarios are shown in Figure 3. Given a PLA with fixed IO positions, the only moves that we can make are swaps of product terms within a netlist (A). Given variable IO positions (B), however, we can also make swaps between the inputs of a netlist or between the outputs of a netlist.

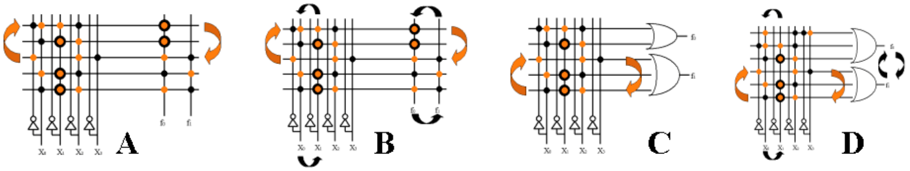


Fig. 3. Allowed annealing moves for the four scenarios

The outputs in a PAL put restrictions on where the product terms can be located, so the PAL with fixed IO positions only allows product terms to be swapped within a given output OR gate (C). In the PAL where we can vary the IO positions, we actually order the outputs by size (number of product terms) for each netlist such that the larger output gates appear at the bottom. This minimizes the overall sizes of the output OR gates. We are then permitted to make three types of moves: swapping input positions, swapping product term positions, and swapping outputs positions of equal size, as shown in (D).

For the PLA with variable IO positions, 50% of the moves are product term swaps and 50% are IO swaps, with the ratio of input to output swaps equal to the ratio of inputs to outputs. For the PAL with variable IO positions, 50% of the moves are product terms and 50% are input moves, with the output moves not currently considered because early results showed no gain from including them.

When the Architecture Generator is done annealing, it creates a file that completely describes the array. This file is then read by the Layout Generator so that a layout of the array can be created.

3.2 Layout Generator

The Layout Generator is responsible for taking the array description created by the Architecture Generator and turning it into a full layout. It does this by combining instances of pre-made layout cells in order to make a larger design. The Layout Generator runs in Cadence's LayoutPlus environment, and uses a SKILL routine that was written by Shawn Phillips. We are currently designing in the TSMC .18-micron process.

Figure 4 shows a PLA that our Layout Generator created (a very small array has been shown for clarity, the arrays we create are often orders of magnitude larger).

Pre-made cells exist for every part of a PLA or PAL array, including the decoder logic needed to program the arrays. The Layout Generator simply puts together these pre-made layout pieces as specified by the Architecture Generator, thereby creating a full layout.

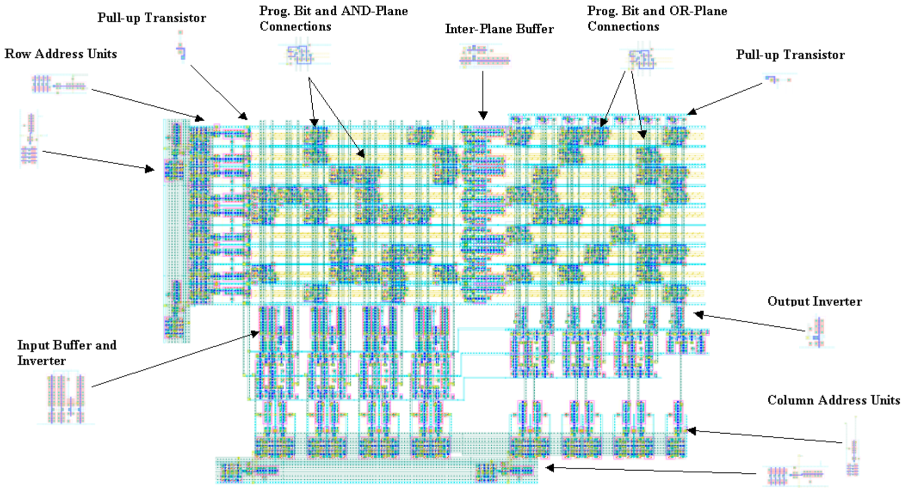


Fig. 4. The PLA is created by tiling pre-made, optimized layout cells

Currently, the PLAs and PALs are implemented using a pseudo-nMOS logic style. PALs and PLAs are well suited to pseudo-nMOS logic because the array locations need only consist of small pull-down transistors controlled by a programmable bit, and only pull-up transistors are needed at the edges of the arrays.

4 Methodology

The use of PALs and PLAs restricts us to the use of .pla format netlists. The first source of netlists is the ESPRESSO suite (the same netlists on which the ESPRESSO algorithm was tested). A second set of netlists comes from the benchmark suite compiled by the Logic Synthesis Workshop of 1993 (LGSynth93). As a whole, these netlists are commonly used in research on programmable logic arrays. The netlists are generally fairly small, but this suits our needs as we are currently only using single arrays to support them.

From these sources we found netlists with similar numbers of inputs, outputs, and product terms and we put them together in groups (1 through 4). The netlists are grouped according to size, as this will be a factor in how well our algorithms perform.

5 Results

The Architecture Generator uses simulated annealing to reduce the total number of programmable bits that the resultant array will require. While tools like VPR can have annealing results where costs are reduced by orders of magnitude, the best cost function improvement that our annealer can obtain is actually bounded. For a two netlists anneal, the cost function can never improve more than 25% and the number of programming bits required can never improve more than 50%. A three netlists anneal has a maximum cost function improvement of 41.7%, while the optimal reduction in the number of programming bits is 66.7%. Similar analysis can be performed on groups of four or more netlists. Since reducing the number of bits is our final objective, the results that we present will show the number of bits required for a mapping rather than the annealing cost.

We do not have a method for determining the optimal bit cost of an arbitrary mapping, but we can know the optimal mapping of a netlist mapped with itself: it is simply the number of connections in the netlist, as all the connections from the first netlist should map to the same locations as the connections from the second netlist. This can be done with any quantity of the same netlist, and the optimal solution will always remain the same. By doing this we can see how close our annealing algorithms come to an optimal mapping.

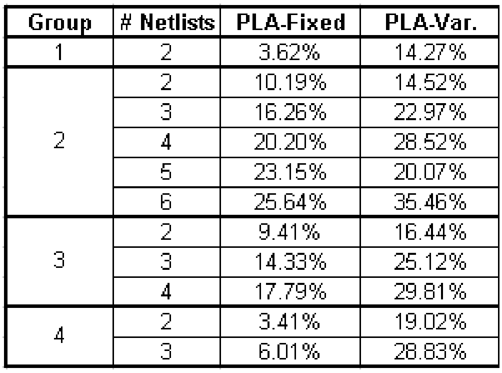
We applied this self-mapping test to five random netlists using each of the four algorithms: PLA-fixed, PLA-variable, PAL-fixed, and PAL-variable. The results showed that when two netlists are mapped with themselves using the PLA-fixed algorithm that the final mapping is always optimal. The PLA-variable had difficulty with only one netlists, *shift*, which was 15.21% from optimal. Note that for this example the random placement was 92.49% worse than optimal, so our algorithm still showed major gains.

For the PAL-fixed algorithm, all of the tests returned an optimal result. The PAL-variable algorithm had a similar result to the PLA-variable algorithm, as the *shift* netlist was only able to get 13.28% from optimal (vs. 92.23% from optimal for a random placement). These near optimal results give us confidence that our annealing algorithms should return high quality mappings for arbitrary netlist mappings as well.

Table 1 shows the results of running the PLA-fixed and PLA-variable algorithms on the different netlist groups that we created. The reduction in bit cost is the difference in the number of programmable connections needed between a random mapping of the netlists and a mapping performed by the specified algorithm. In the table, all possible 2 netlist mappings were run for each specific group and the results were then averaged. The same was done for all possible 3 netlist mappings, 4 netlist, etc., up to the number of netlists in the group.

There are some interesting things to note from the results in Table 1. Firstly, the PLA-variable algorithm always finds a better final mapping than the PLA-fixed algorithm. This is to be expected, as the permuting of inputs and outputs in the PLA-variable algorithm gives the annealer more freedom. The resulting solution space is much larger for the variable algorithm than the fixed algorithm, and it is intuitive that the annealer would find a better mapping given a larger search space. The practical implications of this are that an SoC designer will acquire better area/delay/power results from our reconfigurable arrays by supplying external hardware to support input and output permutations.

Table 1. Average improvement in programming bits for PLA-Fixed and PLA-Variable algorithms over random placement as a function of netlist count.

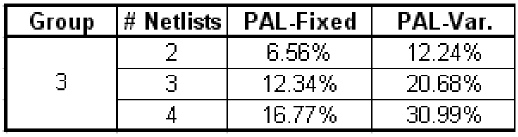


Group	# Netlists	PLA-Fixed	PLA-Var.
1	2	3.62%	14.27%
2	2	10.19%	14.52%
	3	16.26%	22.97%
	4	20.20%	28.52%
	5	23.15%	20.07%
	6	25.64%	35.46%
3	2	9.41%	16.44%
	3	14.33%	25.12%
	4	17.79%	29.81%
4	2	3.41%	19.02%
	3	6.01%	28.83%

Another thing to notice is that the reduction always increases as the number of netlists being mapped increases. This, too, is as we would expect, as adding more netlists to a mapping would increase the amount of initial disorder, while the final mapping is always close to optimally ordered. Note that this does not say that we end up with fewer connections if we have more netlist, it only says that we reduce a greater number of connections from a random mapping.

Table 2 shows the results of running the PAL-fixed and PAL-variable algorithms on netlist Group 3. We see the same results from the PAL algorithms as from the PLA algorithms: increasing the number of netlists in a mapping increases the resulting bit reduction, and allowing input and output permutations always results in a better mapping than requiring fixed input/output locations.

Table 2. Average cost improvement for PAL-Fixed and PAL-Variable algorithms over random placement as a function of netlist count.



Group	# Netlists	PAL-Fixed	PAL-Var.
3	2	6.56%	12.24%
	3	12.34%	20.68%
	4	16.77%	30.99%

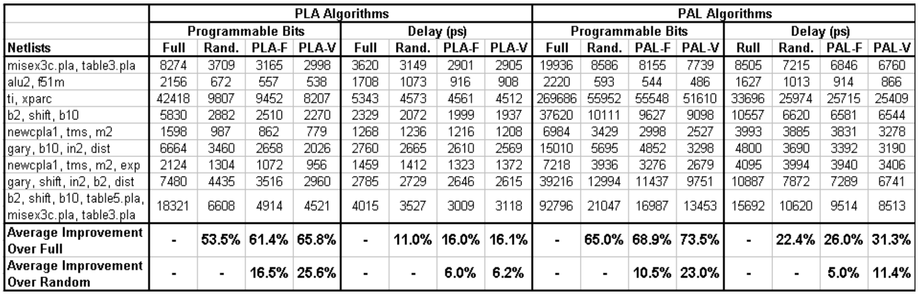
Another important concept is how well netlists match each other: higher reductions will be possible when the netlists being mapped have similar sizes or a similar number of connections. With regards to array size, any netlists that are far larger than another netlist will dominate the resulting size of the PLA or PAL array, and we will be left with a large amount of space that is used by only one or few netlists, resulting in poor reduction. Results showed that arrays of similar sizes showed better reductions than arrays of disparate sizes.

Mapping netlists with a similar number of connections also results in better reductions. If netlist A has far more connections than netlist B then the total number of connections needed will be dominated by netlist A: even if we map all of the connections from netlist B onto locations used by netlist A we will see a small reduction percentage because B contained so few of the overall connections. It is intuitive that having a similar number of connections in the netlists being mapped will allow a higher percentage of the overall programmable connections to be removed. This concept was also verified by experimentation.

One reason that we're trying hard to reduce the number of programmable bits is that it will allow us to use less silicon, and this will allow us to have a smaller array. The original array we create will be the same size but with empty space where programmable bits are not needed, and running a compactor on the layout will allow us to obtain more area savings. Additionally, the removed silicon will result in delay and power gains.

We used hspice to develop delay models of both the PLA and PAL arrays that we create. Table 3 shows the delay and programmable bit results obtained for several runs of the algorithms, along with average improvements over the full arrays and the random arrays (netlists randomly placed and unneeded connections removed).

Table 3. Reductions obtained in number of programmable bits and delay for PLA algorithms



	PLA Algorithms								PAL Algorithms							
	Programmable Bits				Delay (ps)				Programmable Bits				Delay (ps)			
Netlists	**Full**	**Rand.**	**PLA-F**	**PLA-V**	**Full**	**Rand.**	**PLA-F**	**PLA-V**	**Full**	**Rand.**	**PAL-F**	**PAL-V**	**Rull**	**Rand.**	**PAL-F**	**PAL-V**
misex3c.pla, table3.pla	8274	3709	3165	2998	3620	3149	2901	2905	19936	8586	8155	7739	8505	7215	6846	6760
alu2, f51m	2156	672	557	538	1708	1073	916	908	2220	593	544	486	1627	1013	914	866
ti, xparc	42418	9807	9452	8207	5343	4573	4561	4512	269686	55952	55548	51610	33696	25974	25715	25409
b2, shift, b10	5830	2882	2510	2270	2329	2072	1999	1937	37620	10111	9627	9098	10557	6620	6581	6544
newcpla1, tms, m2	1598	987	862	779	1268	1236	1216	1208	6984	3429	2998	2527	3993	3885	3831	3278
gary, b10, in2, dist	6664	3460	2658	2026	2760	2665	2610	2569	15010	5695	4852	3298	4800	3690	3392	3190
newcpla1, tms, m2, exp	2124	1304	1072	956	1459	1412	1323	1372	7218	3936	3276	2679	4095	3994	3940	3406
gary, shift, in2, b2, dist	7480	4435	3516	2960	2785	2729	2646	2615	39216	12994	11437	9751	10887	7872	7289	6741
b2, shift, b10, table5.pla, misex3c.pla, table3.pla	18321	6608	4914	4521	4015	3527	3009	3118	92796	21047	16987	13453	15692	10620	9514	8513
Average Improvement Over Full	**-**	**53.5%**	**61.4%**	**65.8%**	**-**	**11.0%**	**16.0%**	**16.1%**	**-**	**65.0%**	**68.9%**	**73.5%**	**-**	**22.4%**	**26.0%**	**31.3%**
Average Improvement Over Random	**-**	**-**	**16.5%**	**25.6%**	**-**	**-**	**6.0%**	**6.2%**	**-**	**-**	**10.5%**	**23.0%**	**-**	**-**	**5.0%**	**11.4%**

All algorithms show improvements in delay over the full and random placements. Additionally, as more netlists are mapped to the array, the delay gains tend to increase.

The PLA-Variable algorithm does significantly better than the PLA-Fixed algorithm with respect to programmable connections, but this does not scale to delay, as the PLA-V and PLA-F algorithms perform very similarly. This is because the algorithms have no concept of path criticality, and the connections that they are able to remove are often from non-critical paths. Thus further reduction in connections does not directly lead to further reduction in delay.

The PAL-Variable algorithm performs better than the PAL-Fixed algorithm in terms of both programmable connections and delay. This is largely because the PAL-V algorithm is able to permute the outputs (and therefore the output OR-gates), resulting in smaller OR-gates in many circumstances. And while the variable algorithms perform better than the fixed algorithms, more research would be required in order to determine whether the gains obtained would offset the cost of requiring a crossbar at the periphery of the array.

On average, the PLA-Fixed and PLA-Variable algorithms improved upon the delay of a full PLA array by 16.0% and 16.1% respectively. The PAL-Fixed and PAL-Variable algorithms improved upon the delay of a full PAL array by 26.0% and 31.3% respectively. Overall, delay improvements of 5.0% to 11.4% were achieved vs. a random placement.

6 Conclusions

In this paper we have presented a complete tool flow for creating domain specific PALs and PLAs for System-on-a-Chip. We have presented an Architecture Generator that, given netlists as an input, maps the netlists onto a PLA or PAL array of minimal size which uses a near-optimal number of programmable connections. Results show that higher reductions are possible if the inputs and outputs are permutable, suggesting that an SoC designer would want to have crossbars at the inputs and outputs of our arrays. Also, as the number of netlists being mapped to an array increases, the bit reduction between a random mapping and an intelligent mapping increases.

We also presented a Layout Generator that takes PLA or PAL descriptions from the Architecture Generator and successfully creates optimized layouts by tiling premade layout units.

References

[1] K. Compton, S. Hauck, “Totem: Custom Reconfigurable Array Generation”, IEEE Symposium on FPGAs for Custom Computing Machines Conference, 2001.

[2] K. Compton, A. Sharma, S. Phillips, S. Hauck, “Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems”, International Symposium on Field Programmable Logic and Applications, 2002.

[3] S. Phillips, S. Hauck, “Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2002

[4] A. Sharma, “Development of a Place and Route Tool for the RaPiD Architecture”, Master’s Thesis, University of Washington, 2001.

[5] A. Yan, S. Wilton, “Product Term Embedded Synthesizable Logic Cores", IEEE International Conference on Field-Programmable Technology, 2003.

[6] F. Mo, R. K. Brayton, “River PLAs: A Regular Circuit Structure”, DAC, 2002.

[7] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, Boston, 1984.

[8] V. Betz, J. Rose, “VPR: A New Packing, Placement and Routing Tool for FPGA Research”, International Workshop on Field Programmable Logic and Applications, 1997.

A Key Agile 17.4 Gbit/sec Camellia Implementation

Daniel Denning[1], James Irvine[2], and Malachy Devlin[3]

[1] Institute of System Level Integration,
Alba Centre, Alba Campus, Livingston, EH54 7EG, UK
daniel.denning@sli-institute.ac.uk

[2] Department of Electrical and Electronic Engineering, University of Strathclyde,
204 George St., Glasgow, G1 1XW, UK
j.m.irvine@strath.ac.uk

[3] Nallatech Ltd,
Boolean House, One Napier Park, Cumbernauld, Glasgow, G68 0BH, UK
m.devlin@nallatech.com

Abstract. In this paper we discuss several key agile Camellia implementations. The New European Schemes for Signatures, Integrity, and Encryption (NESSIE) selected Camellia in its portfolio of strong cryptographic algorithms for protecting the information society. In order for an encryption core to be key agile it must be able to accept new secret keys as well as data on every clock cycle. We discuss the design and implementation of the Camellia algorithm for a FPGA. We obtain a throughput of 17.4 Gbit/sec when running on a Virtex-II XC2V4000 FPGA device.

1 Introduction

At the end of February 2003 NESSIE (New European Schemes for Signatures, Integrity, and Encryption), a project with in the Information Society Technologies (IST) of the European Union, announced that the symmetric block cipher algorithm Camellia [1] was to be included in its portfolio of strong cryptographic algorithms [2]. The NESSIE project selected 12 encryption algorithms from the 42 that were submitted, which included digital signatures, identification schemes, block ciphers, public-key encryption, MAC algorithms, and hash functions. Only 3 block ciphers were chosen out of these 12, which were MISTY1, SHACAL-2, and Camellia. Five well established encryption algorithms were also added to the NESSIE portfolio and one of these five included the AES (Advanced Encryption Standard).

The Camellia algorithm is a 128-bit block cipher jointly developed by NTT and Mitsubishi Electric Corporation. The algorithm has also been submitted to other standardisation organisations and evaluation projects such as ISO/IEC JTC 1/SC 27, IETF, TV-Anytime Forum, and has also been included in the list of cryptographic techniques for Japanese e-Government systems selected by CRYPTREC (Cryptography Research and Evaluation Committees) in Japan.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 546–554, 2004.

The current fastest Camellia FPGA implementation is by Ichikawa [3] that proposes a pipelined architecture. The implementation targets a VirtexE device and runs with a throughput of 6.75Gbits/sec. The core takes up 9,692 slices and it is assumed that no block-RAM was used.

In this paper we discuss the implementation of three Camellia fully pipelined encryption architectures: a key agile architecture, a key agile sub-pipelined architecture, and the other where the key schedule is off-chip. Currently the architectures have been targeted to a specific Xilinx FPGA, and do use the features of the FPGA architecture to provide very fast implementations. Section 2 describes the Camellia algorithms including its decryption process. In Section 3 we discuss the implementation. Section 4 discusses the performance results. Finally Section 5 concludes the paper.

2 The Camellia Algorithm

The Camellia algorithm processes data blocks of 128-bits with secret keys of lengths 128, 192, or 256 bits. Note that Camellia has the same interface as the AES. In our implementation we focus on the algorithm using a key length of 128-bits.

Using a key length of 128-bits results in an 18 round Feistel structure. After the 6th and 12th rounds FL/FL^{-1} function layers are inserted to provide some non-regularity across rounds. This should provide some extra security against future attacks [4]. There are also two 64-bit XOR operations before the first round and after the last, also known as pre- and post-whitening. The top-level structure of the algorithm can be seen in Figure 1. The key schedule, discussed later in this section, generates subkeys for each round, FL/FL^{-1} layers, and pre- and post-whitening.

The FL-function is defined by:

$$Y_{R(32)} = ((X_{L(32)} \cap kl_{L(32)}) <<< 1) \oplus X_{R(32)}, \qquad (1)$$

$$Y_{L(32)} = (Y_{R(32)} \cup kl_{R(32)}) \oplus X_{L(32)}, \qquad (2)$$

where $Y_{L(32)}$ are the 32 most significant bits of the 64-bit output and $Y_{R(32)}$ are the 32 least significant bits. The FL^{-1} function is just the inverse of FL.

Each round can be composed of a *F*-function with an XOR. This can be seen in each 6 round phase structure shown in Figure 2. A different subkey is applied to each *F*-function and the output is XORed with the previous but one result.

The *F*-function is defined as

$$Y_{(64)} = P(S(X_{(64)} \oplus k_{(64)}). \qquad (3)$$

The *P*-function is constructed only of XOR components and is a linear transformation from 8 input bytes to 8 output bytes. The *S*-function represents a substitution using one of 4 s-boxes. All 4 s-boxes are closely related and defined by:

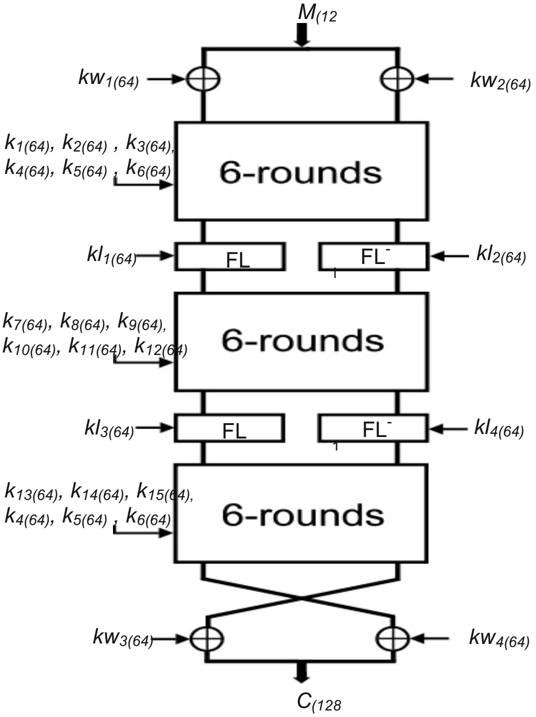


Fig. 1. Top-level procedure of Camellia

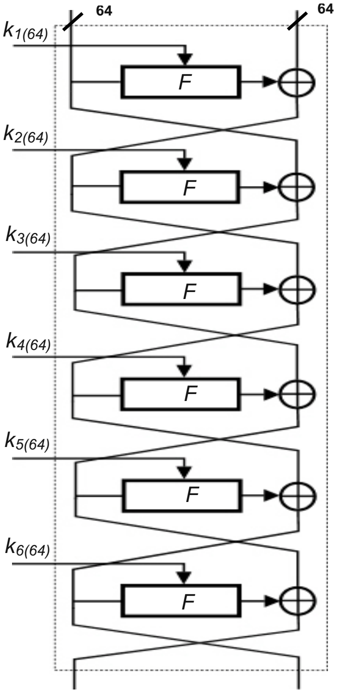


Fig. 2. Inner Structure of 6 Rounds

$$
\begin{aligned}
S_1(x) &= h(g(f(x \text{ XOR } a))) \text{ XOR } b, & (4)\\
S_2(x) &= S_1(x) <<< 1, & (5)\\
S_3(x) &= S_1(x) >>> 1, & (6)\\
S_4(x) &= S_1(x <<< 1), & (7)
\end{aligned}
$$

where f and h are linear mappings, g is an inverse over GF(2^8), and a, b are fixed constants

2.1 Key Schedule

The Camellia key schedule for a 128-bit key produces twenty-six 64-bit subkeys, for use in the 18 rounds, pre- and post-whitening and FL/FL^{-1} function layers. The key resembles that of MISTY [5] and is composed of two steps. The first involves deriving a 128-bit variable $K_{A(128)}$ from the original secret key K. Then the second step involves generating further round keys by cyclic rotating either K (now K_L) or K_A by 15 or 17. The structure for generating K_A can be seen in figure 3. $\Sigma i_{(64)}(i = 1,2,3,4)$ are 64-bit key constants.

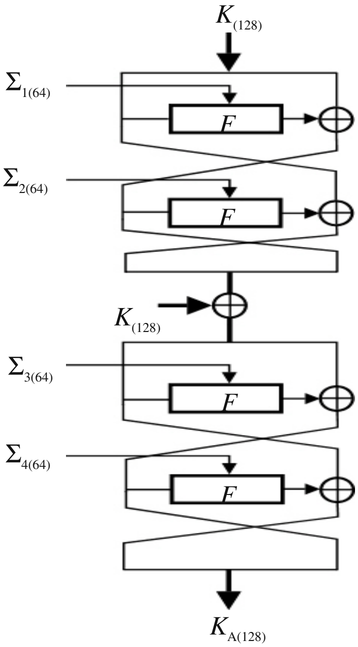


Fig. 3. Key Schedule for 128-bit Cipher Key

Each subkey variable is cyclic shifted left by 15 for rounds 3 to 12, and the first FL/FL^{-1}. Then cyclic shifted left by 17 for the rest of the rounds, the second layer of FL/FL^{-1} and the post-whitening. It must be noted that the key schedule for a 192-bit or 256-bit key differs slightly in that an extra subkey variable $K_{B(128)}$ is also produced from the procedure, which is then interleaved with K_L and K_A.

2.2 Camellia Decryption

The Camellia decryption procedure is exactly the same as the encryption, it does not have any inverse functions like with in the AES algorithm, but the keys are needed in reverse order. So the subkey that was needed to encrypt the data block in round 18 is now needed to decrypt in round 1. This can cause a delay in the overall decryption process if the subkeys need to be produced first, before decryption can take place. This can be overcome when the decryption receives the pre-computed subkeys from the sender.

3 Implementation

Here we describe three fully pipelined encryption architectures. The first architecture is key agile with only pipeline registers added between the rounds. The second architecture has had some sub-pipelining applied to the *F*-Function. The final architecture is non-key agile and has removed the key schedule from the FPGA. It is assumed that a microprocessor will work out the required subkeys for each data session and not each clock cycle. Although Camellia has several modes of operation we only consider the ECB (Electronic Code Book) mode in our implementation because there is no feedback in the datapath.

3.1 Key Agile Fully Pipelined Architecture

A key agile fully pipelined architecture provides very high security and throughput as new data blocks can be encrypted on each clock cycle with a different key. Most fully pipelined implementations of other encryption algorithms insert shift registers between each encryption round. An efficient way to delay data samples in a Virtex FPGA is to use the SRL16E or 'Shift Register Look-Up Table of 16-bits with clock Enable'. For delays greater than 17 cycles the SRL16E can be chained together and address inputs of the LUT are hard-wired to select the appropriate tapping point of the delay line. For the Camellia algorithm delays are inserted after each round, as shown in figure 4 below.

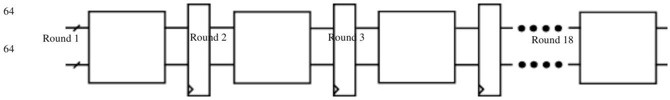


Fig. 4. Pipelined Registers In Between Camellia Rounds

Each round has its own associated F-function, as described earlier, and each function utilises 4 different s-boxes for byte substitution. For both implementations we have chosen to use block-RAMs for the s-boxes. Each block-RAM can be pre-configured with the appropriate substitution values. Each F-function makes 2 calls to the same s-box. This can be seen in Figure 5. As we are targeting the encryption core to a Virtex-II device it is possible to use dual-port block-RAM, which allows simultaneous reads of the same memory location and because we are only reading memory locations, treating the memory like ROM and not updating the s-box values we do not need to worry about the other contentious conditions with regards to writing to the memory. Therefore we can utilise the dual-port block-RAMs in each *F*-function. The new *F*-function structure for the Virtex-II can be seen in Figure 6.

For a key agile architecture each new subkey must be available on every clock cycle for each round. To achieve this, the key schedule must be unrolled and pipelined so that a new key can be accepted on each clock cycle. For the Camellia algorithm the

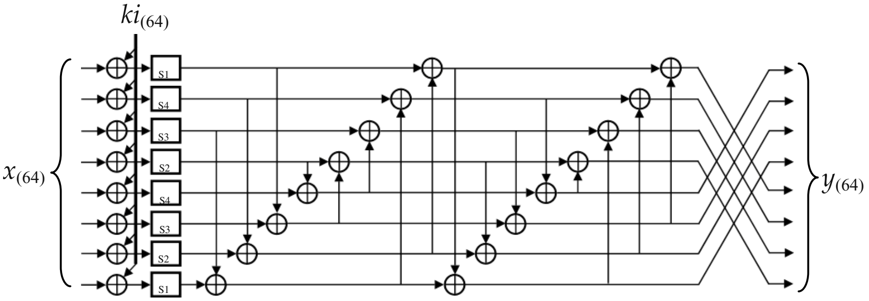


Fig. 5. Normal Structure of *F*-Function

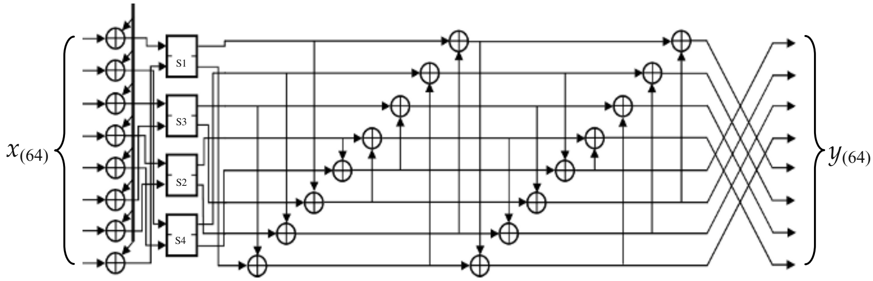


Fig. 6. Structure of *F*-Function Utilising Dual-Port Block-RAM

structure of Figure 3 is used to find the first subkeys for round 1 and 2, and for the following rounds each subkey is shifted either by 15 or 17. The key agile architecture can be seen in Figure 7. Note that the thicker lines between each of the subkey modules represents the two subkey variables K_A and K_L.

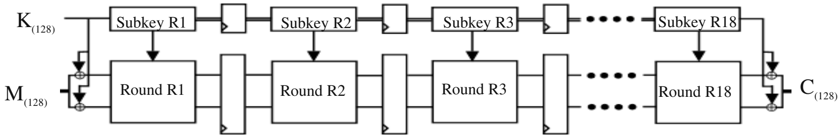


Fig. 7. Key Agile Single Pipelined Camellia Architecture

3.2 Sub-pipelining

For the sub-pipelining architecture we have added pipelining registers as described in the section above, but we have also added registers in to the *F*-Function. These registers have been added between the *S*-Function and the *P*-Function. Further sub-pipelining can be achieved by adding registers between the *S*-Function and the XOR of the F-function, but this has not been done in our implementation.

3.3 Non-key Agile Fully Pipelined Architecture

The non-key agile fully pipelined architecture is achieved by allowing the same key for each data transfer, thus the frequency of changing the cipher key is much lower. The subkeys can be either pre-computed on or off chip, but each subkey must be stored locally for the encryption pipeline until the arrival of a new cipher key. For our implementation we are allowing the subkeys to be pre-computed off the chip but are stored locally for each round. In terms of security this is the least secure option because the key schedule is off chip. This does mean though that there is less logic on the FPGA so a higher throughput should be obtainable.

4 Performance Results

We have implemented all the pipelined architectures on a Virtex-II XC2V4000. The cores were designed graphically using Xilinx's System Generator 3.2. The tool produced the associated VHDL and project files, which were then synthesised using ISE 5.2. Both cores were verified using the Camellia test vectors that were submitted to NESSIE.

The key agile core can run up to a frequency of 127Mhz, which results in a throughput of 16.3Gbits/sec. The core requires 5368 slices (23%) of the FPGA and uses 88 block-RAMs (73%). By partial sub-pipelining we can increase the throughput to 17.4Gbits/sec. This core requires 7837 slices (34%) and 88 block-RAMs (73%). Generally comparisons against other encryption cores can be difficult and can depend on architecture, device, use of such IP as block-RAM, key agility, on-chip key scheduling, and other such criteria. Compared to Ichikawa's [3] implementation that achieved 6.75Gbits/sec we are nearly 10Gbits/sec faster. The purpose of our non-key agile core was to find out the throughput cost associated with key scheduling on-chip but the more important results are the fully pipelined key agile architectures, as these cores have more of a practical usefulness.

Table 1 shows a list of other known high throughput pipelined block cipher architectures of the winners put forward by NESSIE as well as the finalists with in the project. For implementations that have used block-RAMs we have not included the efficiency. It is possible to calculate the extra number of slices that the block-RAMs would occupy if distributed memory was to be used like in [11], but this sort of calculation doesn't take into account the extra routing, which would have an effect on the total throughput and therefore the efficiency. Also it must be noted that efficiency measured in Mbps/slice is a measure of how well an implementation uses the slices to obtain throughput. It is not a measure of how well an implementation efficiently utilises the resources with in a FPGA device. A comparison against the winners and finalists shows that our implementation is comparable with other implementations. With regards to the finalists a SHACAL-2 implementation has yet to be done, although we expect the result to be near the SHACAL-1 implementation [8]. Both MISTY1 [6] and AES [10] provide high throughputs. When the AES uses RAM based substitution boxes the throughput is lower than the Camellia at 11.77Gbit/sec, but is higher when s-boxes are implemented using LUTs. MISTY1 has a higher throughput but also has a 208-pipelined delay, whereas our implementation has a 54-pipelined delay, which includes the block-RAM delay.

Table 1. Summary of NESSIE block cipher finalists and winners on FPGAs

Algorithm	**Authors**	**Device**	**Area** **Slices** **(BRAMS)**	**Key agility**	**Throughput** **Gbits/sec**	**Efficiency** **Mbps/slices**
Camellia pipelined	Authors	XC2V4000	5368 (88)	YES	16.3	-
Camellia sub-pipelined	Authors	XC2V4000	7837 (88)	YES	17.4	-
MISTY1	Rouvroy et al. [6]	XCV1000	6,322	YES	19.34	3.06
AES RAM s-boxes	Standaert et al. [10]	XCV3200E	2784 (100)	YES	11.77	-
AES LUT s-boxes	Standaert et al. [10]	XCV3200E	1767	YES	20.85	1.17
SHACAL-1[1]	McLoone et al. [8]	XC2V4000	13,729	NO	17.02	1.24
IDEA[1]	Gonzalez et al. [7]	XCV600	12,026 (estimated)	NO	8.3	0.69
RC6[1]	Beuchat[9]	XC2V3000	8,554 (80)	NO (assumed)	15.2	-

With regards to the non-key agile implementation, where the key schedule is completed off-chip, we get an implementation that has a throughput of 23.38Gbit/sec. The implementation uses 72 block-RAMs (60%) and 4133 slices (17%). This shows the cost on throughput with regards to key agility. By having the key schedule on-chip and key agile, costs us in theory 6Gbit/sec.

5 Conclusions

In this paper we present two key agile fully pipelined Camellia encryption cores. The partially sub-pipelined implementation offers the fastest Camellia implementation to date. New secret keys can be received on every clock cycle. We have utilised the available dual-port block-RAMs for use in the *F*-Function. We have also shown the cost throughput when implementing a key agile core. Camellia is an important block cipher for future secure systems and this has been proven by NESSIE. We have not used any algorithm optimisation but this will be the topic for further work with Camellia. We also intend to fully sub-pipeline the algorithm to achieve a higher throughput.

Acknowledgements. The authors would like to thank the financial support of the Engineering and Physical Sciences Research Council in the UK.

References

1. Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., Tokita, T., "Specification of Camellia – 128-bit Block Cipher", URL: http://info.isl.ntt.co.jp/camellia/#Spec, September 2001.
2. New European Schemes for Signatures, Integrity, and Encryption (NESSIE), "NESSIE Project Announces Final Selection of Crypto Algorithms", IST-199-12324, URL: http://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/press_release_feb27.pdf, February 2003.
3. Ichikawa, T., Sorimachi, T., Kasuya, T., Matsui, M., "On the Criteria of Hardware Evalution of Block Ciphers(1)", Techn report of IEICE, ISEC2001-53, September 2001.
4. Yin, Y.L, "A Note on the Block Cipher Camellia", Japanese Contribution for ISO/IEC JTC1 SC27, URL: http://infor.isl.ntt.co.jp/camellia/Publications, California, USA, 2000.
5. Matsui, M., "New Block Encryption Algorithm MISTY", 4th Fast Software Encryption Workshop, pp. 54-68, Israel, January 1997.
6. Rouvroy, G., Standaert, F.X., "Efficient FPGA Implementation of Block Cipher MISTY1", International Parallel and Distributed Processing Symposium 2003, France, April 2003.
7. Gonzalez, I., Lopez-Buebo, S., Gomez, F. J., Martinez, J., "Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm", 13th International FPL Conference, Portugal, September 2003.
8. McLoone, M., McCanny, J. V., "Very High Speed 17 Gbps SHACAL Encryption Architecture", 13th International FPL Conference, Portugal, September 2003.
9. Beuchat, J. L., "FPGA Implementations of the RC6 Block Cipher", 13th International FPL Conference, Portugal, September 2003.
10. Standaert, F.X, Rouvroy, G., Quisquater, J.J., Legat, J.D, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs", Cryptographic Hardware and Embedded Systms - CHES 2003, Germany, September 2003.

High Performance True Random Number Generator in Altera Stratix FPLDs

Viktor Fischer[1], Miloš Drutarovský[2], Martin Šimka[2], and Nathalie Bochard[1]

[1] Laboratoire Traitement du Signal et Instrumentation,
Unité Mixte de Recherche CNRS 5516, Université Jean Monnet,
10, rue Barrouin, 42000 Saint-Etienne, France
{Fischer,Nathalie.Bochard}@univ-st-etienne.fr

[2] Department of Electronics and Multimedia Communications,
Technical University of Košice,
Park Komenského 13, 04120 Košice, Slovakia
{Milos.Drutarovsky,Martin.Simka}@tuke.sk

Abstract. The paper presents a high performance True Random Number Generator (TRNG) embedded in Altera Stratix Field Programmable Logic Devices (FPLDs). As a source of randomness, an on-chip noise generated in the internal analog Phase-Locked Loop (PLL) circuitry is used. In contrast with traditionally used free running oscillators, it uses and extends a recently developed method of randomness extraction based on two rationally related clock signals. Although it was developed for the Stratix family, the principle can be easily employed in other digital devices containing analog PLLs. We use the large flexibility of PLLs embedded in Stratix family to demonstrate the relationship between PLL and TRNG configuration, the quality of output random bit-stream, and the speed of the generator. The quality of TRNG output is confirmed by applying statistical tests, which pass also for a high-speed version of the generator giving up to 1M random bits per second. The generator developed for cryptographic applications helps to increase the system security, but it can also be used in a wide range of other applications.

1 Introduction

The issue of random number generation is becoming crucial for implementation of cryptographic systems in Field Programmable Logic Devices (FPLDs). Random numbers are needed in particular for key generation, authentication protocols, zero-knowledge protocols, padding, in many digital signature schemes, and even in some encryption algorithms [1]. In all these applications, security greatly depends on the quality of the source of randomness. The quality of generated numbers is proved by passing statistical tests. In addition to good statistical properties of the obtained numbers, the output of the generator used in cryptography must be unpredictable. For this reason, pseudo-random generators easily implementable in FPLDs, are not suitable for cryptographic applications.

It is well-known that most attacks are directed at the implementations of the cryptographic algorithms and not at the algorithms themselves. This means that

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 555–564, 2004.

special attention should be paid to avoid weaknesses that help the attacker to break a system. Our aim was to find a solution completely that can be embedded in a modern FPLD. Digital circuits of modern FPLDs include only limited sources of randomness, e.g. metastability, frequencies of free-running oscillators, clock jitter, etc. Usually, reliable and fast True Random Number Generators (TRNGs) based on metastability are very difficult to implement. Free running oscilators are typicaly used in known FPLD based TRNGs [2,3]. In principle, TRNGs based on free running oscilators and intrinsic jitter contained in digital circuits can be used without any additional FPLD resources. Actual implementations [3] use off-chip components that generaly decrease the cryptographic security of the implementation. Implementation [2] requires very careful placement of ring oscillator pairs embedded into Xilinx FPLD. It can provide random bits at speeds up to 0.5Mbit/s with good statistical characteristics. In [4] we have proposed a novel method of randomness extraction based on two rationally related synthesized stable clock signals. It was shown that it is well suited for modern FPLDs with internal analog Phase-Locked Loop (PLL) circuitry (e.g. Apex, Cyclone or Stratix FPLDs from Altera [5,6]).

In this paper we present deeper analysis of the possible generator's configurations. In addition, we describe a detailed methodology for the design of TRNG, so that a reader will get a complete overview of how to set the parameters of the TRNG for the given requirements. We use the large flexibility of PLLs embedded in Stratix FPLDs to demonstrate the relationship between PLL and TRNG configurations, the quality of the output random bit-stream, and the speed of the generator. Although the TRNG was developed for the Altera Stratix family of devices, the principle can be easily employed in other digital devices containing analog PLLs.

The paper is organised as follows: in Section 2 we describe the PLL circuitry as the source of random jitter. Section 3 is dedicated to the basic principle of our TRNG. In Section 4 we present a general design methodology and propose several TRNG configurations for evaluation. In Section 5 we show the experimental results and discuss the features and possible advantages and limitations of the proposed configurations. Finally, Section 6 presents conclusions and perspectives.

2 PLL Blocks Embedded in Stratix FPGAs

Recent ASICs and FPLDs generate clock frequencies using PLL circuits to multiply an external low-frequency crystal by an order of magnitude. The analog variant of the PLL implemented in Altera FPLDs offers a source of unpredictable randomness applicable in cryptography. Each PLL block can provide at least one synthesized clock signal with frequency F_{OUT} [5]:

$$F_{OUT} = F_{IN} \frac{m}{n \times k} = F_{IN} \frac{K_M}{K_D} \tag{1}$$

where F_{IN} is the frequency of the external input clock source. The Altera Stratix devices include two types of PLLs:

Fast PLL (FPLL): Stratix devices include up to 8 FPLLs. The FPLLs offer general-purpose clock management with multiplication and phase shifting. The multiplication is simplified in comparison to EPLL and uses only m/k scaling factors with a range from 1 to 32 [5]. Input frequency can vary in dependency on m (for speed grade -5) from 15 to 717 MHz, output frequency from 9.4 to 420 MHz, and the frequency of the Voltage Control Oscillator (VCO) from 300 to 1000 MHz.

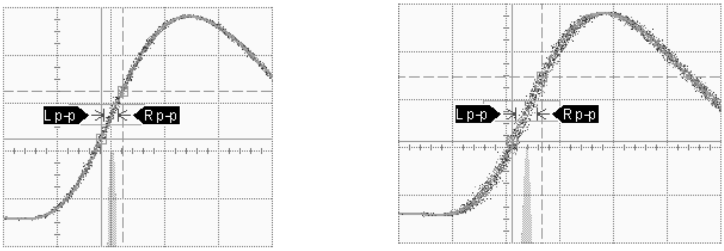
Enhanced PLL (EPLL): Compared to FPLL, EPLLs have additional configurable features like external feedback, configurable bandwidth, run-time reconfiguration, etc. The also have an enhanced range of parameters. The input frequency can vary for a speed grade -5 device from 3 to 684 MHz, output frequency from 9.4 to 420 MHz and the frequency of the VCO from 300 to 800 MHz. Reference-, feedback- and post-divider values n, m and k can vary from 1 to 512 (1024 for k) with a 50% duty cycle [5].

2.1 Jitter Generated in Stratix PLLs

In analog PLLs, various noise sources cause the internal VCO to fluctuate in frequency. The internal control circuitry adjusts the VCO back to the specified frequency and this change is seen as jitter. Under ideal conditions, the jitter is caused only by analog (non-deterministic) internal noise sources, and is noted as an intrinsic jitter. Other possible frequency fluctuations are caused by variations of supply voltage, temperature, external interference through the power, ground, or by the internal noisy environment generated by internal FPLD circuits [7].

The size of the intrinsic jitter depends on the quality factor Q of the VCO, on the bandwidth of the loop filter and on the so-called pattern jitter introduced by the phase frequency detector. The intrinsic jitter is often given in a peak-to-peak value or 1-sigma (RMS) value. The 1-sigma value of the jitter (σ_{jit}) depends on the technology and the configuration of the PLL and it can range up to 100 ps [5,6]. Since the technology of the PLL and the quality of the VCO are usually defined, a user can change the output jitter directly by modification of scaling factors (for FPLL and EPLL) and filter bandwidth (only for EPLL), but also indirectly by the design of the board (separation of the analog and digital ground, filtering of the analog power supply, etc.).

Since the size of the jitter is very important for our method, we needed to measure it for various PLL configurations. To reduce the subjectivity of the board design strategy, we have selected the Altera DSP Development board with a Stratix EP1S25F780C5 device [8] for jitter measurements and TRNG implementation. The jitter has been measured similarly in [9] using Agilent Infiniium DCA 86100B wide bandwidth oscilloscope. We have found that in comparison to the Nios board with APEX [10] (used as a reference in [4]) the jitter is significantly smaller. For example, for the FPLL and the ratio 12/7 the jitter achieves 1-sigma value of about 10 ps (see Figure 1(a)) and for the EPLL and the ratio 139/133 the 1-sigma value of the jitter is about 16 ps (see Figure 1(b)). Note that this value depends on the PLL settings and the type of power supply filter



(a) FPLL

(b) EPLL

Fig. 1. Jitter of the clock signal (horizontal scale: 200 ps/div)

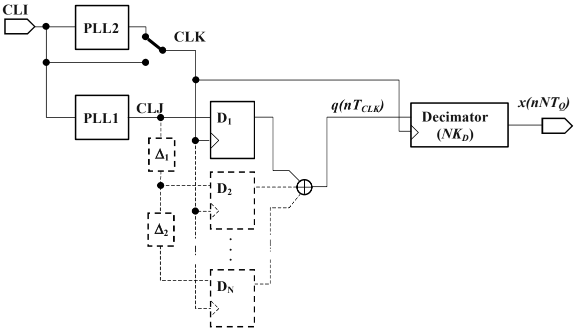


Fig. 2. Basic structure of the TRNG

included on the development board, but is never lower than the internal intrinsic jitter of FPLD.

3 Principle of the PLL-Based TRNG

The basic principle behind our method is to extract the randomness from the jitter of the clock signal synthesized in the embedded analog PLL. The jitter is detected by the sampling of a reference (clock) signal using a rationally related (clock) signal synthesized in the on-chip analog PLL. The fundamental problem lies in the fact that the reference signal has to be sampled near the edges influenced by the jitter. The basic structure of the random bitstream generator is depicted in Figure 2.

Let CLJ be an on-chip PLL-synthesized rectangular clock waveform with the frequency

$$F_{CLJ} = F_{CLK} \frac{K_M}{K_D} \tag{2}$$

where CLK is a reference clock signal and parameters K_M and K_D defined in (1) are related to the PLL structure. Signal CLJ is sampled into the D flip-flop using a clock signal with frequency F_{CLK}. There are K_D rising edges of CLK signal and $2K_M$ edges (rising and falling) of CLJ waveform during time period

$$T_Q = K_D T_{CLK} = K_M T_{CLJ} \ . \tag{3}$$

It has been shown in [4] that if K_M and K_D are relative primes, the set of samples creates an equidistant set of values. The worst-case distance between the two closest edges of CLK and CLJ during the period T_Q is given as

$$\mathrm{MAX}(\mathit{\Delta} T_{min}) = \frac{T_{CLK}}{4K_M}\mathrm{GCD}(2K_M, K_D) = \frac{T_{CLJ}}{4K_D}\mathrm{GCD}(2K_M, K_D) \tag{4}$$

where GCD means Greatest Common Divisor. If K_M, K_D, and F_{CLJ} are chosen so that

$$\sigma_{jit} > \mathrm{MAX}(\mathit{\Delta} T_{min}) \tag{5}$$

we can guarantee that during T_Q the sampling edge of CLK will fall at least once into the edge zone of CLJ (the edge zone means the time interval around the edge with a width smaller than σ_{jit}). Therefore during the period T_Q, K_D values of CLJ will be sampled into the D flip-flop and at least one of them will statistically depend on the random jitter, so the output value $q(nT_{clk})$ of the flip-flop will be nondeterministic. In [4] we used delay elements to increase the probability of overlapping of CLK and CLJ edge zones. In [9] we showed that the delay line is not needed for known values of jitter, when $\sigma_{jit} \gg \mathrm{MAX}(\mathit{\Delta} T_{min})$.

The decimated output signal

$$x(nT_Q) = q(nT_Q) \oplus q(nT_Q - T_{CLK}) \oplus \ldots \oplus q(nT_Q - (K_D - 1)T_{CLK}) \ , \tag{6}$$

which is generated at the output of an Exclusive-OR (XOR)-based decimator [11] as a bit-wise addition modulo 2 ($\oplus$) of samples $q(.)$ sampled with the frequency F_{CLK}, will be nondeterministic, too.

4 TRNG Architectures Embedded in Stratix FPGAs

As it can be seen in Figure 2, the TRNG can be designed using one or two PLLs, depending on the position of the switch. Our implementation strategy was to get the fastest and the best quality generator using a minimum amount of resources (PLLs). Since the Stratix family contains two types of PLLs, several configurations are possible. Although the most economic solution would be based on the use of one FPLL (since there are four FPLLs in the selected device), multiplication and division factors of a single FPLL cannot fullfil the implementation condition (5). However, the extended range of parameters of the EPLL enable one to build a single-PLL TRNG. For this reason, the following four architectures of the TRNG implemented in Stratix devices are possible:

1. Two FPLLs (referenced further as configuration A)
2. One FPLL and one EPLL (configuration B)

3. One EPLL (configuration C)
4. Two EPLLs (configuration D)

To follow our implementation strategy, we have analyzed the influence of individual parameters of PLLs on the output bit rate and on the sensitivity to the jitter expressed through the parameter MAX($\varDelta T_{min}$) defined in (4). Next, we present relations between the TRNG parameters, which are important in the TRNG design (note that for a single-PLL configuration, M_{CLK} and D_{CLK} are equal to one). The PLLs' output frequencies can be expressed as:

$$F_{CLK} = \frac{M_{CLK}}{D_{CLK}} F_{CLI} \tag{7}$$

$$F_{CLJ} = \frac{M_{CLJ}}{D_{CLJ}} F_{CLI} = \frac{M_{CLJ} D_{CLK}}{D_{CLJ} M_{CLK}} F_{CLK} = \frac{K_M}{K_D} F_{CLK} \ . \tag{8}$$

Since the TRNG requires at least

$$\text{MAX}(\varDelta T_{min}) \approx \sigma_{jit} \tag{9}$$

or better

$$\text{MAX}(\varDelta T_{min}) \ll \sigma_{jit} \tag{10}$$

then the first practical design condition is (see equation 4):

$$GCD(2K_M, K_D) = 1 \ . \tag{11}$$

If condition (10) is not fulfilled, the quality of the random bitstream output can be enhanced to some extent by the use of the delay elements and D flip-flops depicted in dashed lines in Figure 2.

Now, let us characterize the relationship between the jitter and the output bitrate of the TRNG. For the jitter we get:

$$F_{CLI}\text{MAX}(\varDelta T_{min}) = \frac{1}{4 M_{CLK} M_{CLJ}} \ , \tag{12}$$

so decreasing MAX($\varDelta T_{min}$) for fixed F_{CLI} requires maximization of M_{CLK} and M_{CLJ}. Coefficients D_{CLK} and D_{CLJ} have no influence on it. For the output bitrate $R = 1/T_Q = F_{CLK}/K_D$ we get the condition

$$R = \frac{F_{CLI}}{D_{CLK} D_{CLJ}} \tag{13}$$

so increasing R for fixed F_{CLI} requires minimization of D_{CLK} and D_{CLJ}. Of course, optimization of (12) and (13) cannot be done independently. There are system limits expressed by the condition

$$\frac{R}{\text{MAX}(\varDelta T_{min})} = 4 F_{CLK} F_{CLJ} \ . \tag{14}$$

The application of the presented analysis of the TRNG design will be illustrated by several implementation examples given in the following section.

5 Experimental Results

TRNG architectures presented in Section 4 were tested on an Altera DSP board with Stratix EP1S25F780C5 FPLDs [8]. Acquired bits were transmitted to the PC through a parallel port. The complete TRNG design including 1024 x 8-bit FIFO and a parallel interface controller needs up to 120 Logic Elements (LE) from about 25000 LEs available in the device. The signal *CLK* was used as a clock signal for the control logic and was therefore limited to about 250 MHz (although the output frequency of the PLL can be higher). The TRNG architectures were described in VHDL and implemented using the Altera Quartus II development system, version 3.0 SP2. Because the jitter depends on an analog process, the real TRNG output cannot be simulated. In order to test the basic quality of different versions of TRNGs, we evaluated the following parameters (all of them were computed for the record length of $N = 1,000,000$ bits):

1. *Bias* computed as

$$bias = E[b(n)] - 0.5 = E[b] - 0.5 \cong \frac{N_1}{N} - 0.5 \tag{15}$$

where N_1 is the number of $b(n) = 1$ for $n = 0, 1, \ldots, N-1$. For a good TRNG, the bias should converge to 0 (with deviation $\approx \pm 3/\sqrt{N}$).

2. *Maximal autocorrelation coefficient* computed as

$$\rho_{max} = \max\{|corr(b_k)|, k = 1, 2, \ldots, 100\} \tag{16}$$

where

$$corr(b_k) = corr(b(n), b(n-k)) = \tag{17}$$

$$= \frac{E\Big[\{b(n) - E[b(n)]\}\{b(n-k) - E[b(n-k)]\}\Big]}{\sqrt{var(b(n))var(b(n-k))}}$$

$$var(b(n)) = var(b) = E[\{b - E[b]\}^2] = E[b]\{1 - E[b]\} \tag{18}$$

Based on [1,11] it can be shown that for a good TRNG (with $bias \to 0$) and a finite record length N the $corr(b_k)$ follows standard normal distribution $N(0,1)$ and the following condition should be fulfilled (value $\chi = 2.576$ is from $P(X > \chi) = \alpha = 0.01/2$ valid for $N(0,1)$ distribution)

$$\rho_{max} \to \frac{2.576}{\sqrt{N}} = 0.002576 \tag{19}$$

3. *Standard FIPS140-2 statistical tests* [12] that analyze 20,000 bit records and define thresholds to assess TRNG randomness. FIPS140-2 tests include Monobit, Poker, Run and Long runs tests [1,13]. We analyzed 100 sequences for each tested TRNG architecture and evaluated relative number (t_M, t_P, t_R, t_L) of sequences that passed each test. A good TRNG should pass all FIPS140-2 tests so that $t_{NIST} = t_M t_P t_R t_L = 1$.

Table 1 includes parameters and results for selected TRNG architectures. As could be expected, the best output bitrate and quality (expressed through the *bias*, ρ_{max} and t_{NIST}) are obtained using a TRNG configuration with two EPLLs. Since the use of the delay line (from Figure 2) could hide quality differences between configurations, it has not been used to generate the results.

Table 1. Configuration parameters and quality evaluation of the tested TRNGs

Conf.	PLL1			PLL2			Final		MAX $\varDelta T_{min}$	R	σ_{jit}	Bias	ρ_{max}	t_{NIST}
	Type	K_M	K_D	Type	K_M	K_D	K_M	K_D	$[ps]$	$[kb/s]$	$[ps]$			
A	Fast	12	7	Fast	25	12	144	175	10.4	952.4	10	-0.358	0.043	0
B	Enh.	43	7	Fast	25	12	516	175	2.9	952.4	23	0.054	0.023	0
C	Enh.	212	207	-	1	1	212	207	14.7	386.5	12	-0.003	0.012	0.96
D	Enh.	43	7	Enh.	31	10	430	217	2.3	1142.9	13	0.002	0.003	1

To emphasize the effect of the number of delay elements on the quality of the generated bitstream, we have chosen a lower quality TRNG (configuration A). The results presented in Table 2 show that if more than two elements are used, the bias and correlation coefficient are significantly reduced. Statistical parameters expressed through the parameter t_{NIST} are less stable (below seven delay elements), because MAX($\varDelta T_{min}$) in configuration A and the jitter have comparable size.

Table 2. Quality evaluation of configuration A for different number of delay elements

# of elements	Bias	ρ_{max}	t_{NIST}
0	-0.358	0.0433	0
1	0.175	0.011	0
2	0.024	0.002	0.007
3	0.030	0.003	0
4	-0.001	0.002	1
5	-0.021	0.003	0.014
6	-0.027	0.002	0
7	0.000	0.002	1
8	0.007	0.003	0.98
9	0.000	0.003	1

The influence of the parameter MAX($\varDelta T_{min}$) (sensitivity to the jitter) on the quality of generated bitstream can be seen in Table 3. We use configuration A with eight delay elements as a reference, but by changing multiplication and division factors of both FPLLs we obtain various sensitivities and speeds of the generator. It can be seen that, in spite of the use of the delay line, the quality of the output bitstream is lower if MAX(ΔT_{min}) is bigger than the jitter (see last two lines of the Table 3).

We can conclude that the best performance TRNG in Stratix family can be obtained using two EPLLs. Usage of the delay elements can further improve the quality of the output. The final speed of the generator (more than 1Mbit/s) is much higher than that presented in [4], while the quality remains comparable.

Table 3. Quality evaluation of configuration A with eight delay elements for different multiplication and division coefficients

Conf.	PLL1			PLL2			Final		MAX ΔT_{min}	R	σ_{jit}	Bias	ρ_{max}	t_{NIST}
	Type	K_M	K_D	Type	K_M	K_D	K_M	K_D	$[ps]$	$[kb/s]$	$[ps]$			
A1	Fast	12	7	Fast	25	12	144	175	10.4	952.4	10	0.000	0.002	1
A2	Fast	12	7	Fast	23	12	120	161	11.3	1142.9	12	0.002	0.003	1
A3	Fast	12	7	Fast	17	12	72	119	15.3	1904.8	11	-0.007	0.003	0.98
A4	Fast	12	7	Fast	11	12	60	77	23.7	2285.7	14	0.133	0.032	0
A5	Fast	10	7	Fast	9	12	50	63	34.7	2285.7	15	-0.144	0.003	0

In order to demonstrate the quality of the proposed TRNG, we performed more strict statistical tests for the best version of the TRNG - configuration D, with eight delay elements. There are some well-documented general statistical tests that can be used to look for small deviations from an ideal TRNG [12], [13]. A very good TRNG should pass many of these tests. We performed testing with the NIST test suite [12] including the latest known corrections [14]. Our NIST statistical tests were performed on 1 Gigabit of continuous TRNG output records and followed the testing strategy, general recommendations, and result interpretation described in [12]. We have used a set of 1024 1-Megabit sequences produced by the generator and we have evaluated the set of P-values at a significance level $\alpha = 0.01$. We did not find any detectable deviations for the ensemble of 1024 1-Megabit records. Results of these tests are not included in the paper due to space limitations.

6 Conclusions

In this paper we have described the methodology and design of high performance PLL-based true random number generators embedded in modern FPLDs. We used the large flexibility of the analog PLLs embedded in the new Altera Stratix FPGA family to demonstrate the relationship between PLL parameters and TRNG configuration, the quality of the output random bit-stream, and the speed of the generator. The high quality of TRNG output was confirmed by applying special statistical tests, which are passed even for the high-speed version of the generator delivering more than 1M random bits per second. For the first time it was experimentally confirmed that delay-line elements can improve the quality of TRNG output if PLL jitter is very small.

The proposed solution is very cheap, uses few logic resources and is faster than comparable methods. Although the functionality of the proposed solution has been demonstrated for the Altera Stratix family, the same principle and design methodology can be used for all recent high-performance ASICs or FPLDs that include an on-chip reconfigurable analog PLL. The generator developed for embedded cryptographic applications helps to increase the system security, but it can also be used in a wide range of other applications.

Acknowledgments. This work has been done in the frame of the project CryptArchi included in the French national program ACI Cryptologie (project number CR/02 2 0041) and the project VEGA 1/1057/04.

References

1. Menezes, J.A., Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, New York (1996)
2. Kohlbrenner, P., Gaj, K.: An embedded true random number generator for fpgas. In: Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, ACM Press (2004) 71–78
3. Tsoi, K., Leung, K., Leong, P.: Compact FPGA-based true and pseudo random number generators. In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). (2003) 51–61
4. Fischer, V., Drutarovský, M.: True Random Number Generator Embedded in Reconfigurable Hardware. In Kaliski, Jr., B.S., Koc, C.K., Paar, C., eds.: Workshop on Cryptographic Hardware and Embedded Systems – CHES 2002. Volume 2523 of LNCS., Berlin, Germany, Springer-Verlag (2002) 415–430
5. Stratix Device Handbook: Volume 2, Chapter 1, Using General-Purpose PLLs in Stratix & Stratix GX Devices, v.2.2 (2003)
6. Altera Application Note 115: Using the ClockLock & ClockBoost PLL Features in Apex Devices, v.2.3 (2002)
7. Xilinx: Superior Jitter Management with DLLs, Virtech Tech Topic VTT013, v.1.2 (2003)
8. Altera Data Sheet: Stratix EP1S25 DSP Development Board, v.1.4 (2003)
9. Fischer, V., Drutarovský, M., Šimka, M., Celle, F.: Simple PLL-based True Random Number Generator for Embedded Digital Systems. In: Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop – DDECS 2004, Stará Lesná, Slovakia (2004) 129–136
10. Altera Data Sheet: Nios Embedded Processor Development Board (APEX device), v.2.2 (2003)
11. Davies, R.B.: Exclusive OR (XOR) and hardware random number generators (2002)
12. Rukhin, A., et al.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. (NIST Special Publication 800-22) (revised May 15, 2002).
13. NIST FIPS PUB 140-2: Security Requirements for Cryptographic Modules (2001)
14. Kim, S., Umeno, K., Hasegawa, A.: Corrections of the NIST statistical test suite for randomness. Cryptology ePrint Archive, Report 2004/018 (2004)

A Universal and Efficient AES Co-processor for Field Programmable Logic Arrays*

Norbert Pramstaller and Johannes Wolkerstorfer

Institute for Applied Information Processing and Communciations (IAIK), TU Graz,
Inffeldgasse 16a, A–8010 Graz, Austria
{Norbert.Pramstaller, Johannes.Wolkerstorfer}@iaik.at

Abstract. In this article we present a compact and efficient co-processor that calculates the Advanced Encryption Standard (AES). It implements the whole functionality of the AES algorithm: all key lengths (128-bit, 192-bit, and 256-bit) are supported for both, encryption and decryption. Furthermore, it supports the Cipher Block Chaining mode. Due to an innovative AES State representation the complete AES co-processor is well suited for low-end FPGAs. The integrated AMBA interface facilitates the integration of the co-processor in System-on-Chip designs too. An implementation on a Xilinx Virtex-E FPGA device uses only 1,125 CLB slices and no block RAMs. Our FPGA implementation reaches a throughput of 215 Mbps at a clock frequency of 161.0 MHz.

1 Introduction

The National Institute of Standards and Technology (NIST) selected the Rijndael algorithm among several other algorithms as the Advanced Encryption Standard (AES) in October 2000. In winter 2001, the AES algorithm became the Federal Information Processing Standard FIPS-197 [1].

Due to the increasing importance of reconfigurable devices, numerous FPGA AES implementations have been published within the last years. These implementations mainly focus on high throughput rates [4,5]. By using techniques like loop unrolling and pipelining, they are able to report throughput rates up to 12,160 Mbps [4]. Applying such techniques leads to AES hardware implementations that require a huge amount of FPGA resources that are only available for expensive devices and can only be used for high-end applications. Considering low-end applications, high throughput rates are not always required (e.g. wireless communications) and high-end FPGAs are too expensive.

In this article we present a new AES architecture that is supported by most of the FPGA product-families and can be implemented using inexpensive low-end FPGAs. It is the first known AES FPGA implementation that does not require on-chip block RAMs. Besides supporting the complete AES standard, it features the Cipher Block Chaining mode (CBC). The design relies on an

[*] The work described in this paper has been supported [in part] by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 565–574, 2004.

unconventional but effective hardware architecture that was conceived to map efficiently on reconfigurable hardware like FPGAs from Xilinx [12]. An innovative AES State representation and highly optimized VHDL code—without target specific extensions—helped to obtain a small circuit with a high maximum clock frequency.

The remainder of this article is structured as following: a short description of the AES algorithm is given in Section 2 and Section 3 presents the proposed architecture and discusses design considerations. Section 4 introduces the basic features of FPGAs we exploited for our implementation. Finally, we present results in Section 5 and conclusions are drawn in Section 6.

2 The AES Algorithm

The AES algorithm is a symmetric block cipher that encrypts 128-bit plaintext data with a 128-bit, 192-bit, or 256-bit cipher key [1]. As other symmetric ciphers, AES applies a so-called round function iteratively to the plaintext to compute the ciphertext. The number of iterations (Nr) depends on the key-length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. The round function consists of the transformations *SubBytes*,*ShiftRows*, *MixColumns*, and *AddRoundKey*. An extra round key for each round is used for the key-dependent transformation *AddRoundKey*. The round keys are derived from the cipher key with the key-expansion function. Figure 1 depicts the AES dataflow, the round-function transformations for encryption, and reveals that the 128-bit State is organized as a 4 × 4 matrix of bytes.

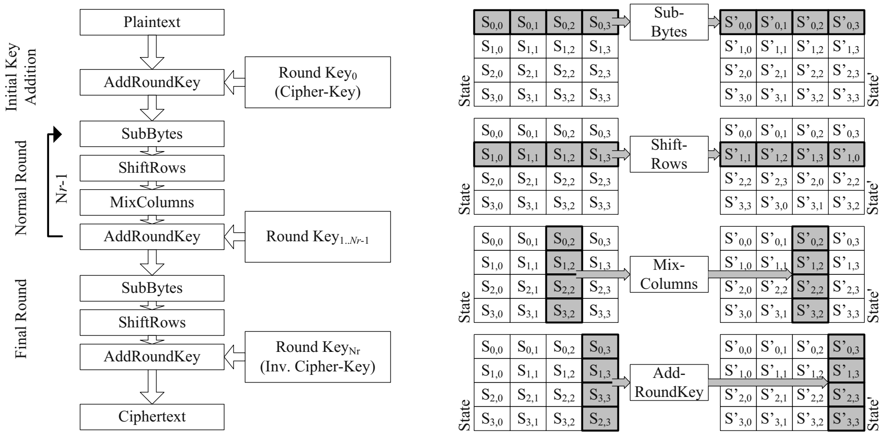


Fig. 1. AES dataflow (left) and round-function transformations (rigth) for encryption

The plaintext input to the AES algorithm becomes the initial State. During the initial key addition the plainttext is added with the cipher key. This is followed by iteratively applying the round function to the State (normal round). The last step of the AES algorithm it the final round that differs slightly from the normal round by omitting the *MixColumns* transformation. After transforming the State during the final round, the State holds the according ciphertext that is the output of the algorithm.

All intermediate 128-bit results are called State too. Each transformation of the round function transforms the 128-bit State into a modified 128-bit State. *SubBytes*, the only non-linear operation of AES, is a multiplicative inversion in $GF(2^8)$ followed by an affine transformation. This function is applied to each byte o the State individually. *ShiftRows* rotates each row of the State by an offset equal to the row index, and *MixColumns* is a constant coefficient multiplication of each column with coefficients that are elements of $GF(2^8)$. Finally, the *AddRoundKey* transformation is the bit-by-bit addition of data and round key. This addition corresponds to an XOR-operation.

Each iteration of the round function requires a 128-bit round key for the *AddRoundKey* transformation. The round key is derived from the cipher key by applying the key-expansion function [1]. This function is based on the *SubBytes* transformation and simple XOR-operations. Obtaining the initial round key requires no transformations: the first 128 bits of the cipher key are used for the initial key addition. All subsequent round keys are iteratively derived from its predecessor.

Decryption is done by inverting the process of encryption: the round iterations are executed in the reverse order. This requires to generate round keys in reverse order too. Even the sequence of the round functions (*SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey*) is reversed and their inverse functions are applied: *AddRoundKey*, *InvMixColumns*, *InvShiftRows*, and *InvSubBytes*. *AddRoundKey* requires no extra inverse function because the XOR-function is its own inverse.

Several modes of operation are defined for symmetric block ciphers [2]. Two common modes are the Electronic Codebook mode (ECB) and the Cipher Block Chaining mode (CBC). ECB is the simplest mode. It applies the cipher function to the plaintext blocks individually. This mode is not recommended to encrypt large quantities of data because repeated input blocks will produce the same output for a given key. The CBC mode alters each input block by combining the result of the previous cipher block with the current input block. This prevents repeated blocks to produce the same output.

3 Architecture of the AES Co-processor

This section describes the architecture of the AES co-processor. Starting with a swift overview, we will present details to highlight some innovative improvements that make it possible to come up with an efficient AES FPGA implementation.

3.1 Related Work

Gaj et al. [4] published the fastest known FPGA implementation. For encryption and decryption with 128-bit keys, a throughput of 12,160 Mbps on a Xilinx Virtex XCV1000BG560-6 device is reported. McLoone et al. [5] achieve a throughput of 6,956 Mbps for 128-bit keys only. They also presented encryption engines for 192-bit or 256-bit keys with accordingly lower throughput. Their combined encryption and decryption implementation can handle 128-bit keys and achieves a throughput of 3,239 Mbps on a Xilinx Virtex-E XCV3200E-8CG1156 device. The third implementation published by Dandalis et al. [7] also provides encryption and decryption for 128-bit keys. They achieve a throughput of 353 Mbps on a Xilinx Virtex XCV1000BG560-6 device. Fischer et al. [6] published a non-pipelined design supporting encryption and decryption for 128-bit keys. They report a throughput of 451 Mbps of their fast configuration and 115 Mbps for an economic configuration. A drawback of their design is the missing on-chip round-key generation. Chodowiec et al. [8] presented an implementation for low-end devices. Using only few resources they achieve a throughput from 139 Mbps up to 166 Mbps depending on the used FPGA device.

All implementations (except [6,8]) use a considerable amount of hardware resources. For instance, [5] requires 138 block RAMs for 256-bit keys. This demands the use of expensive million-gate FPGA devices.

As shown above, most published hardware implementations focus on high throughput rates and do not provide a non-parameterizable design to support the complete AES standard. Furthermore, the high throughput implementations [4,5] do not support the Cipher Block Chaining mode (CBC).

3.2 Architecture

Basic components of the AES co-processor, as shown in Fig. 2, are the AMBA APB interface [3], the data unit, the key unit, and the control unit. The key unit calculates the key-expansion function. All round keys are pre-calculated and stored in the key unit. Pre-calculated round keys allow fast encryption/decryption of different data blocks for the same cipher key because no additional key expansion is required. The data unit holds the State and performs all AES transformations: *AddRoundKey*, *(Inv)SubBytes*, *(Inv)ShiftRows* and *(Inv)MixColumns*. When encryption or decryption has completed, the ciphertext (plaintext in case of decryption, respect.) is stored in the data unit. The control unit receives commands from the AMBA interface and generates control signals for all other modules. In addition to control round-key calculation, encryption and decryption, it also sequences data loading and unloading.

The architecture is similar to the architecture presented in [9]. Differences are a modified State representation and a modified round-key calculation scheme. Due to a non-pipelined approach, the same performance for all modes of operations (ECB and CBC) is reached. Next, we describe the AES data unit, the AES State representation, and the key unit in detail.

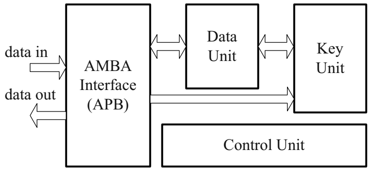


Fig. 2. Architecture of the AES co-processor

Data Unit. The data unit, schematically depicted in Fig. 3, stores the State, all intermediate results of the round function applied to the State and the output data when encryption or decryption has completed. The major difference to all other published AES implementations is the innovative State representation that consists of two States. One State contains the actual State values and the other State stores newly calculated values. Figure 3 depicts the two States, referred to as StateA and StateB. In each cycle, 32 bits (one row or one column) of either StateA or StateB are altered. Using a second State provides a lot of benefits without the need of additional recourses: *(Inv)ShiftRows* comes for free and no State transposition between column and row operations is required.

Storage elements in FPGAs can be efficiently implemented by using synchronous RAMs because the basic logic elements of FPGAs, called slices, can be configured as 16 × 1 bit synchronous RAM. Two slices provide 16 × 1 bit synchronous dual-port RAM functionality (see Section 4). Dual-port RAMs allow concurrent reading and writing to the RAM. Due to these technology features, the State-RAM as depicted in Fig. 3 is implemented as four slices of 8 × 8 bit synchronous dual-port RAMs to allow addressing the slices independently.

The data unit performs all transformations of the round function: *(Inv)-ShiftRows*, *(Inv)SubBytes*, *(Inv)MixColumns* and *AddRoundKey*. *AddRoundKey* and *(Inv)MixColumns* are applied to the State column-by-column, whereas *(Inv)ShiftRows* and *(Inv)SubBytes* are applied to the State row-by-row. Due to the slice architecture of the RAM that holds the State, it is not possible to read/write from/to the RAM column-by-column. Hence, a transposition of the State is necessary if a row-oriented operation follows a column-oriented operation, or vice versa. Transposition would require a reorganization of the State before further operations can be performed. By using two States, transposition can be implemented by accordingly addressing the State-RAM. Furthermore, *(Inv)ShiftRows* can be combined with transposing the State. As a consequence of this, *(Inv)ShiftRows* and transposition come for free. In the sequel we describe the memory organization and State transposition for encryption. The same approach can easily be modified for decryption.

When a row-oriented operation follows a column-oriented operation (or vice versa), the State must be transposed. Combining row and column transformations minimizes the number of required transpositions: *ShiftRows* is combined with *SubBytes* and *AddRoundKey* is combined with *MixColumns* (see Fig. 3). This approach requires only one transposition per round. Encryption requires

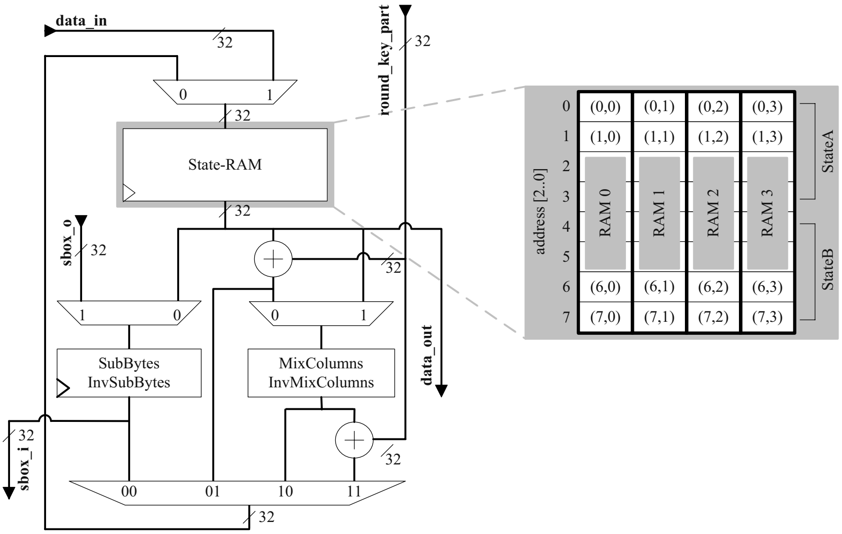


Fig. 3. Architecture of the data unit and State-RAM

SubBytes followed by *ShiftRows*. Since *ShiftRows* does not affect the byte values and *SubBytes* is applied to each byte of the State individually, the order of both operations does not matter. This fact eases the address generation for the State-RAM.

For explaining the State transposition we consider the State as 4 × 4 matrix: $\mathbf{S} = (s_{i,j})_{j=0..3}^{i=0..3}$. The *ShiftRows* transformation described in [1] can then be expressed as follows:

$$\mathbf{S}' = ShiftRows(\mathbf{S}) = (s_{i,j-i\,mod\,4})_{j\,=\,0..3}^{i\,=\,0..3} \quad . \tag{1}$$

If we replace the State by the transposed State, we obtain:

$$\mathbf{S}'^{\mathbf{T}} = ShiftRows(\mathbf{S}^{\mathbf{T}}) = (s_{i+j\,mod\,4,j})_{j\,=\,0..3}^{i\,=0..3} \quad . \tag{2}$$

With the result of (2) the addressing of the StateB-RAM can be determined: the indices (i, j) must be substituted with $(i + j\,mod\,4, j)$. Due to the even number of AES rounds for all key lengths, *ShiftRows* is always applied to StateB only. Thus, the resulting index tuples can be directly mapped to the RAMs. The first part of the tuple index specifies the RAM slice and the second part specifies the RAM address. Since we operate on StateB, we must add an offset of 4 to the index value to get the correct address. Figure 4 shows the transposition of the State, including *ShiftRows* and *SubBytes* for encryption.

Implementation of *(Inv)SubBytes* and *(Inv)MixColumns*. The *(Inv)-SubBytes* transformation is based on [10]. One difference is that the byte inversion in GF(2^8) is implemented by using a synchronous ROM.

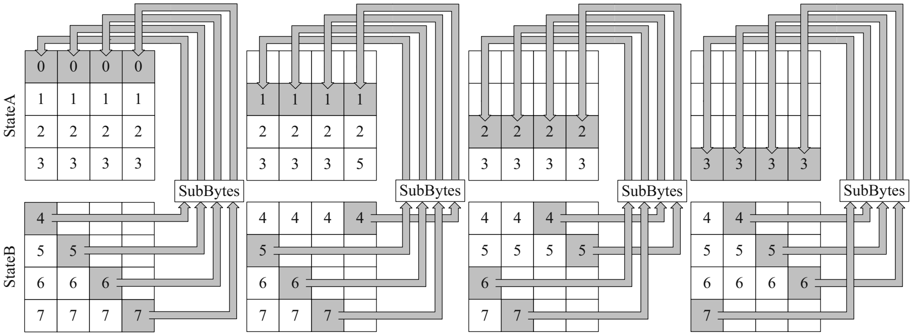


Fig. 4. ShiftRows and SubBytes for encryption

(Inv)MixColumns is similar to the architecture presented in [11]. For further details refer to [10,11].

Key Unit. The key unit holds the round keys and performs the key-expansion function. For each new cipher key, the round keys are pre-calculated to allow rapid encryption of subsequent data blocks for the same cipher key—no further key expansion has to be done. Because decryption uses the encryption round-keys in the reverse order, the key-expansion function must only be calculated once. Hence, the round keys stored in the key store are used for both, encryption and decryption.

The key-expansion function needs the *SubBytes* functionality. To keep the required hardware resources small, *SubBytes* is shared between key unit and data unit (multiplexor-input *sbox_o* in Fig. 3). This can be done easily because the four *SubBytes* units are not used by the data unit during the calculation of the round keys.

The memory of the key unit is separated from the memory of the data unit, because the access of a common memory would be a throughput bottleneck. The key store is implemented as a 64 × 32 bit synchronous single-port RAM.

An innovative aspect of our implementation is that the key unit can handle 128-bit, 192-bit and 256-bit keys with minimal additional hardware requirements. Supporting all key lengths increases the needed hardware resources for the key unit by only 7.8%. The size of the key memory for 256-bit keys is the same as for 128-bit keys. For 128-bit keys, the key-expansion function derives 44 32-bit round-key parts from the cipher key. This requires a 64 × 32 bit RAM. 256-bit keys produce 63 32-bit round-key parts fitting the 64 × 32 bit RAM.

4 Exploiting FPGA Features

The basic building blocks of Xilinx FPGAs are Configurable Logic Blocks (CLBs) [12]. CLBs are arranged in a rectangular matrix and are wired by programmable interconnect. A CLB contains four logic cells (LUTs) that can be programmed to have different functionality: combinational logic (an arbitrary Boolean function of four inputs), logic and a register, or synchronous 16 × 1 bit RAM. Combining two logic blocks allows to implement a 16 × 1 bit dual-port RAM. Besides CLBs, Xilinx FPGAs offer block RAM that can store 4096 bits. Block RAM can be configured at ratios between 4096 × 1 and 256 × 16 and may have dual-port functionality. Block RAMs are also suitable for implementing synchronous ROMs.

When multiple, fast, and small RAMs are required, distributed (LUT-based) RAMs offer an ideal solution. The benefit is that the RAM cell is adjacent to the logic and thus, the wiring from the logic to the RAM is negligible. This improves the timing behavior. Multiple distributed RAMs can be merged to either enlarge the address space or the word width. Enlarging the word width is unproblematic (LUTs in parallel), but enlarging the address space can cause performance loss. For instance, a 32 × 1 bit RAM requires two 4-input LUTs whose outputs need to be multiplexed. This leads to a worse timing behavior and an increased amount of hardware resources. In such cases it makes sense to use block RAMs instead of using distributed RAMs.

When customizing hardware for FPGAs, it is in general more efficient to use RAM instead of registers for storage because the cost of RAM is relatively low in comparison to storing information in registers. For instance, the State-RAM (see Section 3.2) would require 31.2 times more hardware resources when implemented with registers. As stated in Section 3.2, the second State of the State-RAM comes for free. This is due to the fact that a RAM with a depth between 1 and 16 requires always one 4-input LUT. So, the second State causes no additional cost.

Using synchronous RAMs and ROMs provides more flexibilities for the implementation. Depending on the target technology and available on-chip resources, it can be chosen whether distributed RAM or block RAM should be used for implementing the storage elements.

ALTERA devices do not support distributed RAM but provide Embedded System Blocks (ESBs) that provide the same functionality as block RAMs in XILINX devices. Hence, our design is also suitable for ALTERA FPGAs.

5 Implementation Results and Comparisons

This section compares the proposed AES co-processor with the works referred to in Section 3.1. In order to provide comparable results, we implemented our co-processor on a Xilinx Virtex-E XCV1000EBG560-8 device.

The performance results given in Table 1 are for the ECB mode. Most of these implementations claiming high throughput rates will have similar performance figures when operating in CBC mode. The CBC mode is strictly recommended and commonly used for encrypting high-speed data streams (e.g. as it is used for encrypting data transfers over networks) and hence, the above-listed high throughput rates lose their significance.

Table 1. Hardware resources and throughput comparison

Work	**Device**	**#CLB-slices**	**#BRAM**	ECB mode **Throughput** [Mbps]
Gaj et al. [4]	Xilinx XCV1000	12,600	80	12,160
McLoone et al. [5] (I)	Xilinx XCV812E	2,222	100	6,956
McLoone et al. [5] (II)	Xilinx XCV3200E	2,577	112	5,800
McLoone et al. [5] (III)	Xilinx XCV3200E	2,995	138	5,000
McLoone et al. [5] (IV)	Xilinx XCV3200E	7,576	102	3,239
Dandalis et al. [7]	Xilinx XCV1000	5,673	?	353
Fischer et al. [6] (I)	FLEX 10KE200-1	2,530	24	451
Fischer et al. [6] (II)	ACEX 1K50-1	1,213	10	115
Chodowiec et al. [8]	Xilinx XC2S30-6	222	3	166
Our proposal	Xilinx XCV1000E	**1,125**	**0**	**215**

[5]: enc.: (I)AES-128, (II)AES-192, (III)AES-256, enc./dec.:(IV)AES-128
[6]: AES-128 enc./dec.: (I) fast configuration, (II) economic configuration

As shown in Table 1 our implementation is the only one that does not require any block RAMs and in contrast to most of the other implementations, it supports the complete AES standard. Furthermore, the presented AES co-processor supports the CBC mode and is equipped with a 32-bit AMBA APB interface that eases the integration with processors used in System-on-Chip designs [3]. If we do not consider the CBC mode and the AMBA bus interface, our approach is still comparable with the above-listed works but we would require less hardware resources (-26 %).

Our implementation utilizes 9.16% of the available logic cells on a Xilinx Virtex-E XCV1000EBG560-8 device. 90.8% of the logic resources and 100% of the on-chip BRAMs can be used by other circuits like a LEON2 or an ARM processor. For a stand-alone application a low-end FPGA (e.g. Xilinx SpartanII XC2S100-6) is sufficient for implementing the complete AES co-processor— the other approaches (except [8]) do not fit on a SpartanII device. The high throughput designs do not support this flexibility and require expensive million-gate FPGAs.

The maximum clock frequency on a XCV1000 FPGA is 161 MHz. At this frequency, a throughput of 215 Mbps for AES-128, 180 Mbps for AES-192, and 156 Mbps for AES-256 is achieved for both ECB mode and CBC mode.

6 Conclusion

In this article we presented a compact AES co-processor for low-end FPGA devices. It implements the whole functionality of AES. In addition to covering the complete AES standard it supports the Cipher Block Chaining mode (CBC). We have shown that due to an innovative State representation the co-processor is well suited for inexpensive low-end FPGAs—most of the competing approaches require expensive multi-million gate FPGAs. An implementation on a Xilinx Virtex-E device uses only 1,125 CLB-slices and no block RAMs. Our FPGA implementation reaches a throughput of 215 Mbps at a clock frequency of 161 MHz for encryption and decryption. The AES co-processor has a convenient 32-bit interface that facilitates the integration in System-on-Chip designs too.

References

1. National Institute of Standards and Technology (NIST), *Advanced Encryption Standard (AES)* Federal Information Processing Standards Publication 197 (FIPS PUB 197), Nov. 2001.
2. National Institute of Standards and Technology (NIST), *Recommendation for Block Cipher Modes of Operation – Methods and Techniques*, NIST Special Publication SP 800-38a, `http://csrc.nist.gov/publications/nistpubs/`, Dec. 2001.
3. ARM Limited, *AMBA 2.0 Specification*, `http://www.arm.com/armtech/`.
4. P. Chodowiec, P. Khuon, and K. Gaj, *Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining*, Proceedings of the Symposium on Field Programmable Gate Arrays – FPGA 2001, pp. 94–102, ACM Press, 2001.
5. M. McLoone and J. McCanny, *High Performance Single Chip FPGA Rijndael Algorithm Implementations*, Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2001, LNCS 2162, pp. 65–76, Springer Verlag, 2001.
6. V. Fischer and M. Drutarovský, *Two Methods of Rijndael Implementation in Reconfigurable Hardware*, Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2001, LNCS 2162, pp. 77–92, Springer Verlag, 2001.
7. A. Dandalis, V. Prasanna, J. Rolim, *A Comparative Study of Performance of AES Final Candidates Using FGPAs*, The Third Advanced Encryption Standard (AES) Candidate Conference, Available from `http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/aes3agenda.html`, 2000.
8. P. Chodowiec and K. Gaj, *Very Compact FPGA Implementation of the AES Algorithm*, Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2003, LNCS 2779, pp. 319–333, Springer Verlag, 2003.
9. S. Mangard, M. Aigner, and S. Dominikus, *A Highly Regular and Scalable AES Hardware Architecture*, IEEE Transactions on Computers, Vol. 52, No. 4, pp. 483–491, April 2003.
10. J. Wolkerstorfer, E. Oswald, and M. Lamberger, *An ASIC implementation of the AES SBoxes*, Proceedings of the Cryptographer's Track at the RSA Conference 2002, LNCS 2271, Springer Verlag, Feb. 2002.
11. J. Wolkerstorfer, *An ASIC implementation of the AES-MixColumn operation*, Proceedings of Austrochip 2001, pp. 129–132, Vienna, Austria, 12 October 2001.
12. Xilinx Incorporated, *Silicon Solutions — Virtex Series FPGAs*, `http://www.xilinx.com/products/`.

Exploring Area/Delay Tradeoffs in an AES FPGA Implementation*

Joseph Zambreno, David Nguyen, and Alok Choudhary

Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA
{zambro1, dnguyen, choudhar}@ece.northwestern.edu

Abstract. Field-Programmable Gate Arrays (FPGAs) have lately become a popular target for implementing cryptographic block ciphers, as a well-designed FPGA solution can combine some of the algorithmic flexibility and cost efficiency of an equivalent software implementation with throughputs that are comparable to custom ASIC designs. The recently selected Advanced Encryption Standard (AES) is slowly replacing older ciphers as the building block of choice for secure systems and is well suited to an FPGA implementation. In this paper we explore the design decisions that lead to area/delay tradeoffs in a single-core AES FPGA implementation. This work provides a more thorough description of the defining AES hardware characteristics than is currently available in the research literature, along with implementation results that are pareto optimal in terms of throughput, latency, and area efficiency.

1 Introduction and Motivation

Cryptography is one of the strongest tools for controlling against many kinds of security threats [1]. These algorithms and techniques form the basic building blocks of secure systems that serve a variety of purposes, including cryptographic hashing, secure key exchange, and digitally signing documents. Secure storage and transmission solutions are needed for all types of platforms, ranging from embedded devices where area is key to massively parallel machines that emphasize high performance. Such a diversity of requirements motivates the exploration of a wide range of cryptographic implementation characteristics.

Field-Programmable Gate Array (FPGA) technology is becoming a popular target for designing cryptographic ciphers, as witnessed by the wealth of recent research [2,3,4,5,6] and commercial [7] implementations. This increased interest in FPGAs from the cryptographic community has been driven by several factors:

- the individual operations required by this class of algorithms are generally simple in terms of required logic; as such any hardware implementation can increase efficiency by reducing the overhead introduced by software.

* This work was supported in part by the National Science Foundation (NSF) under grant CCR-0325207 and also by an NSF graduate research fellowship.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 575–585, 2004.

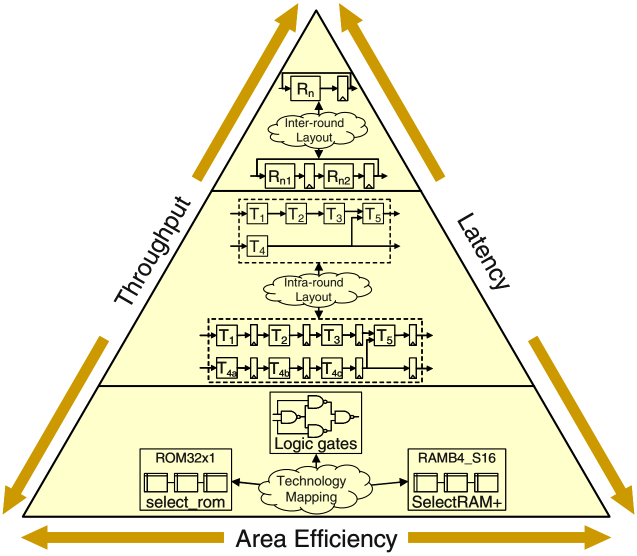


Fig. 1. Top-down block cipher design methodology

- the development process for FPGAs is extremely effective in terms of time-to-market and overall cost when compared to that of custom ASIC designs.
- the reconfigurable nature of FPGAs is especially attractive to cipher designers as it gives them the ability to apply modifications to the implemented algorithm after the initial time of programming [8]. This feature can be used to switch between a set of cryptographic algorithms at runtime [9], to address a freshly-discovered flaw in the cipher algorithm, or to optimize the architecture for a fixed range of inputs [5].

Much recent work in this field has focused on maximizing the theoretical throughput for these cryptographic block ciphers, including both the Data Encryption Standard (DES) [4] and the newly-introduced Advanced Encryption Standard (AES) [2,3,6]. The key distinction between the contributions of this paper and those implementations proposed previously stems from our top-down design methodology (Fig. 1) that allows for area and delay tradeoffs to be managed at several levels of the design hierarchy using a single parameterizable AES core.

As can be seen in Fig. 1, many current block ciphers can be described as a series of logic operations (rounds) that are repeated in an iterative fashion. At the *inter-round* level, decisions can be made as to how each round is laid out in terms of classical optimizations such as unrolling, tiling, and pipelining. Internal to each round structure there are *intra-round* decisions, which can include transformation partitioning and internal pipelining. Also, when considering FPGA hardware, *technology mapping* is especially important, as these decisions can greatly influence designs by allocating specialized resources towards individual computations. Ultimately, making these decisions at each level of the design hierarchy provides much additional control over the performance and area characteristics of the resulting AES FPGA implementation. Our experimental results using Xilinx Virtex-II technology demonstrate that the careful application of this concept can lead to different designs with small area, high throughput, and low latency. One such implementation obtained a maximum throughput of 23.57 Gbps, which to the authors' knowledge is greater than any previously published value.

The remainder of this paper is organized as follows. In Sect. 2, an overview is provided of the AES encryption algorithm, with an introduction to the organization and functionality of the individual transformations from a hardware designer's perspective. Section 3 explores the key design decisions that are possible at various levels in the AES FPGA implementation process, discussing how these choices can result in significant area and delay tradeoffs. Experimental results are presented in Sect. 4, demonstrating how a single soft core can be fine-tuned to implement optimized designs in terms of performance, area, and efficiency metrics. Finally, the paper is concluded in Sect. 5 with a broad summary of some relevant ideas that require further exploration.

2 Overview of AES

In 1997, the U.S. National Institute of Standards and Technology (NIST) announced an open international competition for cipher designs to replace the aging DES as the federal information processing standard. The fifteen submissions to become the new AES standard were publicly evaluated based on algorithmic security, simplicity, and suitability to both hardware and software implementations. Among these submissions was the Rijndael algorithm, which was developed by Vincent Rijmen and Joan Daemen [10]. As it was well fitted to the above factors, Rijndael was selected as the AES competition winner in 2000.

AES is what is known as a symmetric key block cipher, *block cipher* meaning that it operates on fixed-length blocks of data at a time, *symmetric key* meaning that the same key is used during encryption and decryption [1]. Although the original Rijndael specification allowed for both blocks and keys of various lengths, AES is restricted to 128-bit blocks and keys of 128, 192, or 256 bits. In symmetric block ciphers, the algorithms for encryption and decryption using various key lengths often contain quite a large amount of similar features. Consequently, little context will be lost by restricting our focus for the remainder of this paper on AES encryption using a 128-bit key (AES-128E).

The structure of AES-128E is as follows (see Fig. 2). The initial 128-bit key is fed into the `KeyExpansion` function which produces separate keys for each of the 10 required rounds. These rounds combine their scheduled keys with a two

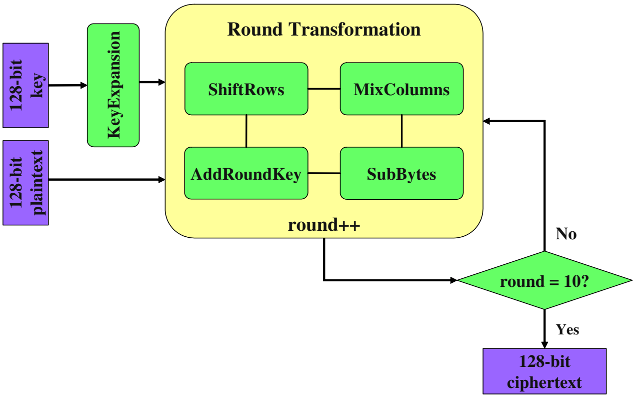


Fig. 2. Algorithmic view of AES-128E

dimensional representation of the input (the "state") using various transformations [11]:

- SubBytes calculates a non-linear function independently on each byte of the state. The substitution used by this transformation can be more simply represented as a lookup table which is referred to as an "S-box".
- MixColumns separately modifies each column of the state in what is essentially a matrix multiplication operation. Fortunately, in the 8-bit finite mathematical field relied on by this class of block ciphers, multipliers can be replaced with simpler fixed-length shifts and XOR operations.
- ShiftRows cyclically shifts the bytes in the last three rows of the state. As this function requires no computational hardware it can be implemented on an FPGA as simple wiring.
- AddRoundKey adds the round key to the state using a bitwise XOR operation.

For those interested in a more thorough description of the AES algorithm, both the official AES standardization documentation [11] and the developers' own writings [10] are informative reads.

3 Design Space Exploration

As the effects of FPGA design decisions on performance and area are often specific to individual architectures, it is necessary to further refine the FPGA target before proceeding in the analysis. For our experiments, we selected the Xilinx Virtex-II device family [12]. Like most Xilinx FPGAs, the Virtex-II devices can be best described as a two-dimensional array of Configurable Logic Blocks (CLBs) that are surrounded by I/O resources and routed together using a programmable interconnect mesh. These CLBs contain functional elements for implementing both combinatorial and synchronous logic, and also include some

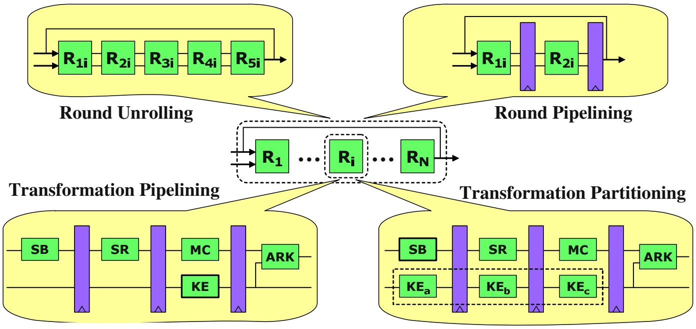


Fig. 3. AES-128E design decisions: *unrolling*, *pipelining*, and *partitioning*

sequential storage. Apart from the CLBs, Virtex-II FPGAs also contain a dedicated amount of dual-ported Block SelectRAM (BRAM) memory modules, each of which can hold up to 18 Kbits of data.

Given a target FPGA similar to the Virtex-II, the first design decision that needs to be made is in regards to the `KeyExpansion` routine. While the operations required to generate a key schedule from the original input key are not complicated, it makes intuitive sense to consider splitting this functionality into smaller `KeyExpansion` modules that would be placed alongside the actual round operations, in what is known as *online* key generation. This is due to the fact that the round key is not used until the final operation in each round (the `AddRoundKey` operation).

3.1 Inter-round Layout

Given that AES-128E is, at its highest level, essentially an iterative looping structure, it is interesting to look at the effect of some classical loop layout optimizations. *Unrolling* replaces a loop body with N copies of that loop body (Fig. 3). As the AES-128E algorithm is a single loop that iterates 10 times, any unrolling amount $1 \leq N \leq 10$ is valid, with $N = 1$ corresponding to the original looping case and $N = 10$ specifying a fully unrolled implementation.

Unrolling the rounds makes them highly amenable to *pipelining*, which is a technique that increases the number of blocks of data that can be processed concurrently. As can be seen in Fig. 3, pipelining in FPGA designs can be implemented by inserting registers between the modules that need to operate independently. Different implementations can be created that split the unrolled rounds into a certain number of pipeline stages, with a similar restriction as before that each stage should be of equal length.

The main advantage of unrolling and pipelining is that it increases the parallelism of the AES encryption algorithm, which should have a positive effect on throughput. It is also possible that a fully unrolled but not pipelined implementation will have a lower latency than in its iterative form. These performance advantages do not come without a price, as unrolling will increase the required FPGA resources by approximately a factor of N; registers used for pipelining will also consume CLBs.

3.2 Intra-round Layout

The clock frequency that FPGAs can operate at is dependent on the critical logic path of the design. As such, the unrolling and pipelining of the rounds as discussed in the previous section will have little positive effect on the critical path when compared to an iterative round structure. In general, this maximum delay will be dependent on the individual transformations inside each round.

Fortunately, these sub-modules are also eligible for pipelining. Figure 3 shows an example of this *transformation pipelining*, where each of the AES transformations are represented by their initials (e.g. `SB` for `SubBytes`). This will reduce the critical path to that of the individual transformation with the greatest delay. We can improve upon this value even further by *partitioning* some of the transformations. Assuming that the `KeyExpansion` operation is performed online, it becomes a prime candidate for partitioning as it has the most slack between the time of its valid input and expected output. After that point it is likely that the maximum delay path will shift to another transformation, which can often also be partitioned. The level of partitioning can be tuned by initially creating highly-partitioned versions of the AES block transformations, and then connecting them with a variable number of pipeline registers.

When combined with round pipelining, transformation partitioning can lead to extremely large gains in throughput, with a relatively small increase in area due to the additional registers that would be needed. However, these heavily pipelined configurations will have extremely long latencies when compared to the base iterative version of AES-128E.

3.3 Technology Mapping

While the majority of the computations needed for the round transformations can be directly mapped to CLBs using the proper synthesis tools, the `SubBytes` operation, or more specifically the S-box tables found in `SubBytes`, can be implemented in one of several ways using Virtex-II technology:

- *Block SelectRAM* - the values in the lookup table for each S-box can be loaded onto these memories at configuration time. Since the memories are dual-ported, each RAM block can implement two separate S-boxes. Block SelectRAMs are dedicated resources on Virtex-II FPGAs, meaning that there is a hard upper limit on the number of them in any design.
- *Distributed SelectRAM* - distributed ROM primitives with the pre-loaded S-box values can also be synthesized directly using CLBs. These are often faster than the Block SelectRAMs, but will require additional glue logic.

– *Logic* - the lookup table code can be converted to a logical representation to be implemented on the CLBs. An advantage of this option is that it provides additional room for the synthesis tools to optimize for area and delay.

Because they are so numerous, the choice of lookup table technology can have a significant effect on the area/delay profile of the `SubBytes` operation and of AES-128E as a whole. Although less in number, these S-box operations are also needed in `KeyExpansion`, whether it is performed online or off.

4 Area and Performance Results

4.1 Experimental Setup

The AES-128E algorithm was implemented using a single VHDL core, with a configuration file to drive preset macros for controlling the round layouts and explicit synthesis directives to determine the mapping of S-boxes. For synthesis we used Synplify Pro 7.2.1 from Synplicity, which was configured to target a Xilinx XC2V4000 FPGA. The XC2V4000 is a medium-sized member of the Virtex-II device family, containing 5760 CLBs (equivalent to 23040 slices) and 120 Block SelectRAM modules. Xilinx ISE 5.2i was used for the place-and-route and timing analysis.

For each design we used these tools to measure the maximum possible clock rate (f_{clk}), the number of utilized slices (N_{slice}), and the number of Block SelectRAMs (N_{bram}) From these base statistics we calculated the resultant maximum throughput using the following equation for a block cipher in non-feedback mode:

$$Tput = \frac{128 \cdot f_{clk}}{blocks\_per\_cycle} , \tag{1}$$

where the number of blocks per cycle is 1 for a fully unrolled implementation, and greater than 1 for any design that re-uses the round structures to process a single input. Also, the latency required to encrypt a single block can be calculated as:

$$Lat = \frac{10 \cdot stages\_per\_round}{f_{clk}} , \tag{2}$$

where the number of clock cycles needed to process a single round is an average and may be a non-integer value. Finally, some idea about the efficiency of an implementation can be obtained by analyzing the following metric:

$$Eff = \frac{Tput}{N_{slice}} , \tag{3}$$

which is measured in throughput rate (bps) per utilized CLB slice.

Table 1. AES-128E implementation results for a Xilinx XC2V4000 FPGA

Design	f_{clk} (MHz)	N_{slice}	N_{bram}	$Tput$ (Gbps)	Lat (ns)	Eff ($\frac{\text{Mbps}}{\text{slice}}$)
UF1-PP0B	110.16	**387**	10	1.41	90.78	3.64
UF1-PP0D	77.91	1780	0	1.00	128.4	0.56
UF1-PP0L	59.00	2744	0	0.76	169.5	0.28
UF1-PP3D	178.09	1940	0	2.28	168.5	1.18
UF1-PP3L	147.75	2909	0	1.89	203.0	0.65
UF2-PP1B	118.57	753	20	3.04	84.34	4.04
UF2-PP2B	150.02	1011	20	3.84	133.3	3.80
UF2-PP2D	119.96	3445	0	3.07	166.7	0.89
UF2-PP3B	173.37	1254	20	4.44	173.0	3.54
UF2-PP3L	118.30	5570	0	3.03	253.6	0.54
UF5-PP0B	72.438	1532	50	4.64	**27.61**	3.03
UF5-PP1D	68.521	7995	0	4.39	145.9	0.55
UF5-PP2B	169.92	2206	50	10.88	117.7	4.93
UF5-PP2L	76.26	11974	0	4.88	262.3	0.41
UF5-PP3B	173.73	2810	50	11.12	172.7	3.96
UF10-PP1B	95.129	2518	100	12.18	105.1	4.84
UF10-PP1D	50.239	15365	0	6.43	199.0	0.418
UF10-PP2B	179.147	3766	100	22.93	111.6	**6.09**
UF10-PP3B	183.58	4901	100	23.50	163.4	4.79
UF10-PP3D	**184.16**	16938	0	**23.57**	162.9	1.39

4.2 Results

A selection of our experimental results can be found in Table 1. Each design is labeled UF*X*-PP*YZ*, where X corresponds to the round unrolling factor $X \in \{1, 2, 5, 10\}$. The Y value specifies the amount of transformation partitioning and pipelining; for $Y = 0$ the design has no pipelining, for $Y = 1$ each unrolled round is pipelined, for $Y = 2$ each round is split into two stages, and for $Y = 3$ each round is split into three stages, with transformations being partitioned across those stages. The Z value specifies the S-box technology mapping in the design, where for $Z =$ [B] Block SelectRAM is chosen, for $Z =$ [D] distributed ROM primitives are chosen, and for $Z =$ [L] logic gates are instantiated. For sake of brevity, unexceptional results from the set of possible designs were pruned when forming Table 1.

From these results several trends can be observed. As was expected, unrolling increased the number of slices by a significant amount. See UF10-PP3D which uses over 8.7× the amount of slices of its iterative counterpart UF1-PP3D. The gain in throughput often out-paced this increased area consumption, leading to an improved area efficiency with the larger unrolling factor. Also, for many of the designs using the distributed SelectRAM resulted in slightly higher clock rates when compared to the dedicated Block SelectRAM. The advantage to using the Block SelectRAM for the S-boxes can be seen in both the area consumption and area efficiency – see the UF2-PP2B design which for the cost of 20 BRAMs saves 71% of the slices that UF2-PP2D requires. Since the performance of the two designs were fairly similar, this choice leads to a 4.26× increase in efficiency. It should also be noted that using logic gates to directly implement the S-box transformations was sub-optimal in all measured metrics when compared to using either SelectRAM type. Finally, aggressive transformation partitioning was quite effective in improving the clock rate and resultant throughput. This technique was successful even in the iterative case, illustrated by the UF1-PP3L design which obtained 2.5× the throughput of UF1-PP0L by partitioning the critical path.

The bolded values in Table 1 represent the designs which are optimal in terms of the selected area/performance characteristics. As was expected, the design that required the least amount of area (387 slices) was UF1-PP0B, an iterative implementation with no transformation partitioning that used Block SelectRAMs. Without using these SelectRAMs, the smallest design was UF1-PP0D, which required 1780 slices. Two of the most aggressively unrolled and pipelined designs (UF10-PP3B and UF10-PP3D) obtained clock rates of over 183 MHz, resulting in throughputs of over 23.5 Gbps. A design that demonstrates the usefulness of partial unrolling is UF5-PP0B, which has the lowest latency of all designs (27.61 ns). The UF10-PP2B design had the highest area efficiency (6.09 Mbps/slice).

4.3 Related Results

It is difficult to make direct comparisons between FPGA implementations of any algorithm since the specific hardware target is often different. However, many recent AES implementations have provided maximum throughput numbers for Xilinx FPGAs that can be used as a measuring stick. For example, Helion Technology reports throughputs of over 16 Gbps [7] for their high-performance commercial AES core. Also, several academic groups have reported high throughput values for designs that are similar to our most aggressively pipelined version. Järvinen et al. [6] created a cipher that operated at 17.8 Gbps, while Saggese et al. [3] reached just over 20 Gbps. Besides ours, the highest reported throughput for an AES implementation belongs to Hodjat and Verbauwhede [13], who designed a 21.54 Gbps core. Our superior throughput numbers can be explained by the fact that our top-down design flow motivated the discovery of additional ways of partitioning transformations, and that doubly packing S-boxes into Block SelectRAMs allowed for a completely unrolled design to fit into a relatively small device.

As a comparison to non-FPGA technologies, a hand-optimized assembly implementation of AES encryption in feedback mode achieved 1.538 Gbps on a 3.2 MHz Pentium IV processor [14]. An ASIC version of the design from [13] targeting $0.18\mu m$ technology was able to achieve greater than 30 Gbps [15]. While this ASIC implementation is far superior to any published FPGA implementation in terms of throughput, this shortcoming is tolerable when considering the other advantages to FPGA technology as listed in Sect. 1.

5 Conclusions

In this paper a top-down methodology for implementing cryptographic block ciphers on FPGAs was proposed and evaluated. For AES-128E it was shown that these design decisions can be managed in a fashion that allows for fine tuning some of the area and delay characteristics. This methodology has been used to discover implementations that are competitive with others in terms of area, latency, and area efficiency. For the future, it would be interesting to see what technology-specific features of other FPGA device families could be exploited to further optimize AES. This same design methodology should also be extended to other cryptographic algorithms. Finally, it would be useful to further investigate how partial reconfiguration can optimize a block cipher given some knowledge of the input key pattern.

References

1. W. Stallings. *Cryptography and Network Security*, Prentice Hall, 2003.
2. A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists. In *Proc. of the Third Advanced Encryption Standard (AES3) Candidate Conference*, pages 13–27, 2000.
3. G. P. Saggese, A. Mazzeo, N. Mazzoca, and A. G. M. Strollo. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *Proc. of the 13th Int'l Conference on Field-Programmable Logic and its Applications (FPL)*, pages 292–302, 2003.
4. J-P. Kaps and C. Paar. Fast DES implementation for FPGAs and its application to a universal key-search machine. In *Proc. of the 5th Annual Workshop on Selected Areas in Cryptography (SAC)*, pages 234–247, 1998.
5. I. Gonzalez, S. Lopez-Budeo, F. J. Gomez, and J. Martinez. Using partial reconfiguration in cryptographic applications: an implementation of the IDEA algorithm. In *Proc. of the 13th Int'l Conference on Field-Programmable Logic and its Applications (FPL)*, pages 194–203, 2003.
6. K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proc. of the Int'l Symposium on Field Programmable Gate Arrays (FPGA)*, pages 207–215, 2003.
7. Helion Technology, Inc. AES Xilinx FPGA core data sheet. available at http://www.heliontech.com, 2003.
8. T. Wollinger and C. Paar. How secure are FPGAs in cryptographic applications? In *Proc. of the 13th Int'l Conference on Field-Programmable Logic and its Applications (FPL)*, pages 91–100, 2003.
9. A. Dandalis, V. Prasanna, and J. Rolim. An adaptive cryptographic engine for IPSec architectures. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 132–144, 2000.

10. J. Daeman and V. Rijmen. The block cipher Rijndael. *Smart Card Research and Applications,LNCS 1820*, J-J. Quisquater and B. Schneier, Eds., Springer-Verlag, pages 288–296, 2000.
11. National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). *FIPS PUB 197*, available at http://csrc.nist.gov, 2001.
12. Xilinx, Inc. Virtex-II complete data sheet. available at http://www.xilinx.com, 2003.
13. A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.
14. H. Lipmaa. AES implementation speed comparison. available at http://www.tcs.hut.fi/˜aes/rijndael.html, 2003.
15. A. Hodjat and I. Verbauwhede. Minimum area cost for a 30 to 70 Gbits/s AES processor. In *Proc. of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 83–88, 2003.

Reconfigurable Instruction Set Extension for Enabling ECC on an 8-Bit Processor

Sandeep Kumar and Christof Paar

Chair for Communication Security
Ruhr-Universität Bochum, 44780 Bochum, Germany
{kumar,cpaar}@crypto.rub.de

Abstract. Pervasive networks with low-cost embedded 8-bit processors are set to change our day-to-day life. Public-key cryptography provides crucial functionality to assure security which is often an important requirement in pervasive applications. However, it has been the hardest to implement on constraint platforms due to its very high computational requirements. This contribution describes a proof-of-concept implementation for an extremely low-cost instruction set extension using reconfigurable logic, which enables an 8-bit micro-controller to provide full size elliptic curve cryptography (ECC) capabilities. Introducing full size public-key security mechanisms on such small embedded devices will allow new pervasive applications. We show that a standard compliant 163-bit point multiplication can be computed in 0.113 sec on an 8-bit AVR micro-controller running at 4 Mhz with minimal extra hardware, a typical representative for a low-cost pervasive processor. Our design not only accelerates the computation by a factor of more than 30 compared to a software-only solution, it also reduces the code-size, data-RAM and power requirements.

1 Introduction

Ubiquitous computing with low cost pervasive devices has started to become reality with RFID applications and smart textiles. These computing devices form large-scale collaborating networks by exchanging information. Privacy and security of this information is important for the overall reliability of these networks and ultimately to the trustworthiness of pervasive applications. In fact, security is often viewed as a crucial feature, a lack of which can be an obstacle to the wide-spread introduction of pervasive applications.

High-volume, low-cost and very small power budgets of pervasive devices implies they have limited computing power, often not exceeding an 8-bit processor clocked at a few MHz. Under these constraints, secure public-key cryptography for authentication are nearly infeasible in software and are therefore usually not available in these systems. On the other hand, public-key cryptography offers major advantages when designing security solutions in pervasive networks. An alternative is to use a cryptographic co-processor such as used for high security applications like smart-cards, but its downside are considerable costs (in terms

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 586–595, 2004.

of power and chip area) which makes it unattractive for many cost sensitive pervasive applications. In addition, a fixed hardware solution may not offer the cryptographic flexibility (i.e., change of parameters or key length) that can be required in real-world applications. An instruction set extension (ISE) is a more viable option because of the smaller amount of additional hardware required and because of its flexibility. The efficiency of an ISE is not just measured by the speed-up it achieves, but also in the decrease in code-size, data-RAM and power consumption.

We use reconfigurable hardware attached to an 8-bit processor to simulate the ISE and to obtain reliable cost/benefit estimates. However, we view the reconfigurability not only useful for prototyping purposes, but a small reconfigurable hardware extension is also an attractive platform for embedded devices as the extension can offer many speed and power benefits for computationally intensive applications as demonstrated in this paper. It should be noted that public-key operations are typically only needed at the initial or final stage of a communication session. Hence, it is perceivable that the ISE can be runtime reconfigured for other applications when public-key operations are not required.

The paper is organized as follows. Section 2 discusses previous work in this field. In Sect. 3 we describe ISEs and in Sect. 4 the mathematical background of ECC is discussed. Section 5 discusses our implementation and results.

2 Previous Work

The use of group of points of an elliptic curve in cryptography was first suggested by Neal Kobilitz [14] and independently by Victor Miller [17]. There has been considerable work since then on efficient implementation of ECC in software, typically targeting high end processors [19,21,7,9]. In the following is a discussion of ECC implementations on constrained environment.

ECC on 8-bit processors have been reported in [5] and [22], both implemented over Optimal Extension Fields (OEF's), originally introduced in [2]. It should be noted that OEFs are not standardized, and their security in conjunction of ECC is not clear. In [5], the ECC implementation is over the field $\mathbb{F}_{p^m}$ with $p = 2^{16} - 165$, $m = 10$, and irreducible polynomial $f(x) = x^{10} - 2$. A performance of 122 msec at 20 Mhz is reported for a 160-bit point multiplication. The sub-field multiplication is done using the math co-processor. [22] implements ECC over $GF((2^8 - 17)^{17})$ on an 8051 micro-controller without co-processor but instead uses the internal 8-by-8-bit integer multiplier. The authors achieve a speed of 1.95 sec for a 134-bit fixed point multiplication using 9 pre-computed points and 8.37 sec for a general point multiplication using binary method of exponentiation. In [15], the authors improve the general point multiplication, to set up an end-to-end wireless ECDH key exchange within 3 sec on a Chipcon CC1010, which is based on the 8051 architecture. An ECDSA implementation on a 16-bit microcomputer M16C, running at 10 Mhz, is described in [10]. The authors propose the use of a field $\mathbb{F}_p$ where prime characteristic $p = e2^c \pm 1$, e an integer within the machine size and c a multiple of machine word size.

The implementation uses a 31-entry table of precomputed points to generate an ECDSA signature in 150 msec and ECDSA verification takes 630 msec. A scalar multiplication of a random point takes 480 msec. The authors in [7] describe an efficient implementation of ECC over $\mathbb{F}_p$ on the 16-bit TI MSP430x33x family of low-cost micro-controllers running at 1 Mhz. A scalar point multiplication over $GF(2^{128} - 2^{97} - 1)$ is performed in 3.4 sec without any precomputation.

It should be stressed that all previous ECC implementations on 8-bit processors have been based on non-standardized ECC parameters in order to overcome the performance bottleneck. This has two disadvantages: first, such solutions are incompatible with standardized protocols; secondly, there is always the possibility that non-standardized parameters have security shortcomings, e.g., new attacks through special mathematical properties introduced in non-standardized underlying finite fields.

Another approach has been to add a crypto co-processor to these micro-controllers. A survey of commercially available co-processors can be found in [8]. However, a full-size ECC co-processor is may be prohibitively expensive for many pervasive applications. Hardware assistance in terms of Instruction Set Extensions (ISE) is more favorable as the cost of extra hardware is quite negligible compared to the whole processor. Previous attempts in this direction [6,13] are only reported for ECC with not more than 133-bits.

3 Instruction Set Extension

The Instruction Set Architecture (ISA) of a microprocessor is the unique set of instructions that can be executed on the processor. General purpose ISA are often insufficient to satisfy the special computational needs in cryptographic applications. A more promising method is extending the ISA to build Application Specific Instruction-set Processors (ASIP.)

There are different ways of extending a processor. We consider an extension as shown in Fig. 1. Here the additional hardware is closely coupled with the arithmetic logic unit (ALU) of the processor, reducing the interface circuitry. The control circuit of the processor is extended to support this extra hardware. The extension can also directly access the data-RAM which is important if the computation is done over several data elements. For multi-cycle instructions, the software has to take special care not to call the custom hardware until the multi-cycle operation is completed.

An efficient ISE implementation requires a tightly coupled hardware and software co-design. In a first step, we used a software-only implementation of ECC to identify the functional elements and code-segments that would provide efficiency gains if implemented as an ISE. Then, a hardware model of the new processor determines the effects of the new extension on the parameters running time, code-size and data-RAM usage.

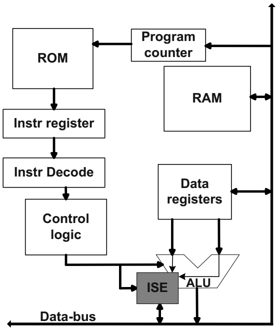


Fig. 1. Processor Core with Extension

4 Elliptic Curve Cryptography

Among public-key algorithms, there are three established families: RSA, DL and elliptic curve cryptography (ECC.) Among them, ECC is considered the most attractive one for embedded environments [4] due to its smaller operand lengths and *relatively* lower computational requirements. ECC is also standardized [18, 1,11,12] and has become a commercially accepted method. ECC is an universal public-key family, allowing mechanisms such as digital signature, key exchange and data encryption. The vast majority of modern protocols, including wireless protocols attractive for pervasive applications, require these public-key functions. A more detailed description of the main operations for an ECC realization can be found in, e.g., [3].

4.1 Elliptic Curves

An elliptic curve over $\mathbb{F}_{2^m}$is of the form $y^2 + xy = x^3 + ax^2 + b$, where $a, b \in \mathbb{F}_{2^m}$ and $b \neq 0$. (x, y) satisfying this equation is called a *point* $\mathbf{P}$ on the curve. Together with $\mathcal{O}$, the identity element, they constitute an abelian set $E(\mathbb{F}_{2^m})$. For our implementation, we use a NIST-recommended 163-bit random curve [18].

The main EC cryptographic operation is the *scalar point multiplication*, $\mathbf{Q} = k \cdot \mathbf{P}$, where $\mathbf{P}, \mathbf{Q}$ are points on the elliptic curve and k is an m-bit integer.

4.2 Elliptic Curve Point Arithmetic

For the implementation of the scalar point multiplication, $k \cdot \mathbf{P}$, we need to implement the group operations point addition and point doubling. The arithmetic is described in detail in [19]. For our implementation, we use the projective co-ordinates where the projective point (X, Y, Z) with $Z \neq 0$ corresponds to the affine point (x, y) where $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$. This projective co-ordinate is chosen because we use the Montgomery point multiplication introduced in [16].

The efficiency of EC group operations depends largely on the efficiency of the underlying field arithmetic. We use the polynomial basis representation $(a_{m-1}...a_1a_0)$ with the reduction polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$. In our implementation, an element from $\mathbb{F}_{2^{163}}$ is represented as an array of 21 eight bit words, with the five last most-significant bits being ignored.

Field Addition is the simplest of all operations, since it is a bit by bit addition in $\mathbb{F}_2$ which maps to word-level XOR operation in software.

Field Multiplication of two elements is a polynomial multiplication followed by reduction modulo $F(x)$. The polynomial multiplication can be implemented in software efficiently using the comb method.

Field Squaring is a simple expansion in $\mathbb{F}_{2^m}$in polynomial basis which is efficiently implemented as a table-lookup.

Field Reduction. Multiplication and squaring require reduction of the polynomial of degree not greater than $(2m - 1)$. Reduction is effectively done using a table-lookup for the 8-bit locations in a byte.

5 Implementation

Our development platform is the Atmel Inc.'s AT94K family of FPSLIC devices (Field Programmable System Level Integrated Circuits). This architecture integrates an AVR 8-bit micro-controller core (used widely in smart-cards and micromotes), FPGA resources, several peripherals and 36K bytes SRAM within a single chip. The platform is appealing for simulating an ISE. The implementations are done on the ATSTK94 FPSLIC demonstration board clocked at 4 Mhz.

We first implemented the ECC algorithm on the 8-bit AVR processor in assembly. The results of the software only implementation is given in Table 1. It is important to mention that the multiplication routine based on the comb method that we used is among the fastest known software algorithms for Galois field multiplication.

The analysis of the software-only implementation shows that $\mathbb{F}_{2^m}$multiplication is the most costly operation with respect to execution time and memory requirement. Moreover, in the Montgomery algorithm (Sect. 4.2), field multiplications are extremely frequent making it the bottleneck operation for ECC. A closer look at the multiplication block showed that the major part of the time was spent for load/store operations because of the small number of registers which cannot hold the large operands. Therefore an ISE for this functional block which also reduces the memory bottleneck can greatly speed-up ECC.

Table 1. $\mathbb{F}_{2^{163}}$ ECC software-only performance on an 8-bit AVR μC (@4 Mhz)

Operation	**Time** (clocks)	**Code-size** (bytes)	**Data RAM** (bytes)
Addition	151	180	42
Multiplication	15044	384	147
Squaring	441	46	63
Reduction	1093	196	63
Point Multiplication ($k.P$)	4.14 sec	8208	358

The popular approach to multimedia extensions has been to divide a large 32/64-bit data-bus into smaller 8-bit or 16-bit multimedia variables, and to run those in parallel as an SIMD instruction. But for public-key cryptographic applications the reverse is true: The operands are much larger than the data-path, requiring many bit operations on long operands. For such applications, bit-parallel processing is required where multiple data-words are operated upon simultaneously. One important issue here is the provision of the ISE with the operands. We approached this situation by implementing a complete $\mathbb{F}_{2^{163}}$ multiplier with minimum possible area.

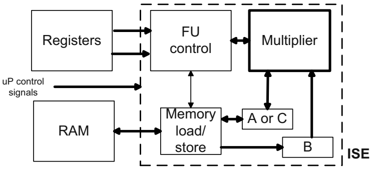


Fig. 2. ISE Interface and Structure

Figure 2 shows the general layout of functional unit (FU) of the ISE we are simulating. Four processor registers are initially loaded with the memory addresses of the two operands A and B. The ISE is then initiated by a control signal to the FU control (FUC) along with the first memory address byte. In our proof-of-concept implementation, this behavior is achieved by sending the byte over the data-lines from the processor to the FPGA, and confirming its reception through interrupt-lines from the FPGA to the processor. After the last memory address byte is received, the FUC initiates the memory load/store circuit within the ISE to load both the 21-byte operands directly from the SRAM in 21-cycles each. Then, the FUC runs the multiplier for 163-cycles to get the result C. During this period, the processor loads the memory address of C, sends it to the FPGA and goes into polling state for the final interrupt from the FPGA. After the result C is obtained, the ISE stores it back directly in the memory in another 21-cycles and then sends the final interrupt signalling the completion of the multiplication. This method of handshaking leads to extra control overheads which can be reduced by having a more tightly coupled ISE to the processor without requiring confirmation interrupts. During the idle polling state, the processor could also be used in other computational work which is independent of the multiplication result. Memory access conflicts during such computation between the processor and the ISE is avoided by using a dual ported SRAM.

5.1 Bit-Serial Multiplier

A bit-serial $\mathbb{F}_{2^m}$ hardware multiplier is the most simple solution and requires the least area. The core of the multiplier is as shown in Fig. 3. The reduction circuit is hardwired here. A modification for implementing a more general reduction polynomial or variable size multiplication is discussed in Sect. 5.3. A 163-by-163 multiplication is computed in 163 clocks, excluding data input and output. In our implementation, control and memory access overheads lead to a total execution time of 313 clocks. Since the multiplier is much faster than the squaring in software, we use the multiplier also for squaring by loading $A = B$. The results (Table 2) show a drastic speed-up using this multiplier. It should be noted that the control overhead can be considerably reduced when the hardware is more tightly coupled within the processor, e.g., in an ASIC implementation of our architecture.

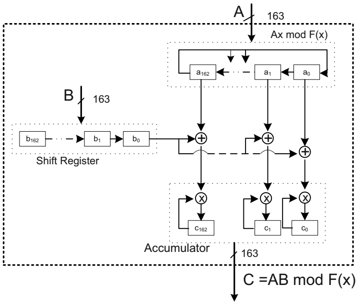


Fig. 3. Bit-Serial LSB Multiplier

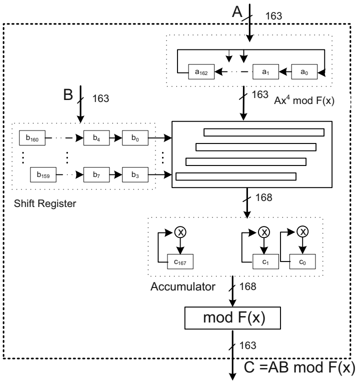


Fig. 4. Digit-4 Serial LSD Multiplier

5.2 Digit-Serial Multiplier

Another trade-off between area and speed is possible by using digit-serial multipliers [20]. Compared to the bit-serial multiplier where only one bit of operand B is used in each iteration, here multiple bits (equal to the digit-size) of B are multiplied to the operand A in each iteration (Fig. 4). We use a digit size of 4 as it gives a good speed-up without drastically increasing the area requirements. A 163-by-163 multiplication with reduction requires 42 clocks. In our implementation, the control overheads leads to a total of 193 clocks.

Table 2. ICE-based ECC point multiplication (@4 Mhz)

Multiplier Type	**CLBs**	**Time** (sec)	**Code-size** (bytes)	**Data RAM** (bytes)
Software-only		4.14	8208	358
163*163 multiplier	245	0.169	2048	273
163*163 digit-4	498	0.113	2048	273

5.3 A Flexible Multiplier

Flexibility of crypto algorithm parameters (especially operand lengths) can be very attractive because of the need to alter them when deemed insecure in the future, or for providing compatibility in different applications. Considering the high-volume of pervasive devices, replacing each hardware component seems improbable. We discuss here how the multiplier can be made more flexible to satisfy these needs.

Support of a generic reduction polynomial with a maximum degree of m of the form $F(x) = x^m + G(x) = x^m + \sum_{i=0}^{m-1} g_i x^i$ requires storage of the reduction coefficients and additional circuitry as shown in Fig. 5 (a similar implementation for a digit-serial multiplier is straightforward). The reduction polynomial can be initialized once in the beginning of the point multiplication. Thus the total number of clocks required for multiplication remains the same.

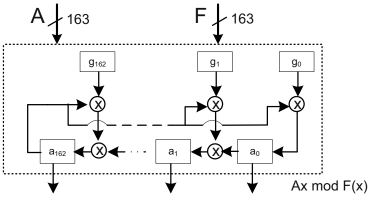


Fig. 5. Bit-serial reduction circuit

Different bit-length multipliers for different key-length ECC can also be supported using this structure. We show as an example, how the 163-bit multiplier could be also used to multiply two 113-bit operands A' and B' with 113-bit reduction polynomial G'.

The operands A, B and the reduction polynomial are initially loaded as

$$\begin{aligned} A &= (a'_{112}...a'_1 a'_0 0...0) = A'x^{50} \\ B &= (0...0b'_{112}...b'_1 b'_0) \;= B' \\ G &= (g'_{112}...g'_1 g'_0 0...0) \;= G'x^{50} \end{aligned}$$

If $C' = A' \cdot B' \bmod F'(x)$ then

$$A \cdot B \bmod F(x) = A'x^{50} \cdot B' \bmod (F'(x)x^{50}) = C'x^{50}$$

Thus the result is stored in the most-significant bits of operand C after 113 clock cycles. The memory load/store circuit and the FU control unit takes care to load the operands appropriately and to fetch the result after the required number of clocks from the multiplier and store it back appropriately in memory.

6 Conclusions

Huge performance gains are possible in small 8-bit processors by introducing small amounts of extra hardware. The results show 1–2 orders of magnitude increase in speed-up for the ECC implementation. The hardware costs are in the range of 250–500 extra CLBs. There is also saving in the code size and data RAM usage for the algorithm. The performance gain due to the ISE can be used to reduce the total power consumption of the devices by running the whole device at a lower frequency, which can be a major benefit in wireless pervasive applications. The proof-of-concept implementation can also be used directly as a reconfigurable ISE. Since public-key exchange is done only in the initial phase of the communication, the FPGA can be run-time reconfigured for an ISE suitable for a different application (like signal processing) running later on the device. Thus two different sets of ISEs can be run on the same constrained device, accelerating both applications without increasing the total hardware cost.

References

1. ANSI X9.62-1999. The Elliptic Curve Digital Signature Algorithm. Technical report, ANSI, 1999.
2. D. V. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume LNCS 1462, pages 472–485, Berlin, 1998. Springer-Verlag.
3. I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, London Mathematical Society Lecture Notes Series 265, 1999.
4. M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M. Kirkup, and A. Menezes. PGP in Constrained Wireless Devices. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
5. Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee. Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 57–70, Berlin, 2000. Springer-Verlag.
6. M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel. A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2n). In *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2002*, pages 381–399. Springer-Verlag, 2002.
7. J. Guajardo, R. Bluemel, U. Krieger, and C. Paar. Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers. In K. Kim, editor, *Fourth International Workshop on Practice and Theory in Public Key Cryptography - PKC 2001*, volume LNCS 1992, pages 365–382, Berlin, February 13-15 2001. Springer-Verlag.

8. Helena Handschuh and Pascal Paillier. Smart Card Crypto-Coprocessors for Public-Key Cryptography. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Proceedings of the The International Conference on Smart Card Research and Applications*, pages 372–379. Springer-Verlag, 2000.
9. D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS, Berlin, 2000. Springer-Verlag.
10. Toshio Hasegawa, Junko Nakajima, and Mitsuru Matsui. A Practical Implementation of Elliptic Curve Cryptosystems over $GF(p)$ on a 16-bit Microcomputer. In Hideki Imai and Yuliang Zheng, editors, *First International Workshop on Practice and Theory in Public Key Cryptography — PKC'98*, volume LNCS 1431, pages 182–194, Berlin, 1998. Springer-Verlag.
11. IEEE. *Standard Specifications for Public-Key Cryptography*, 2000.
12. ISO/IEC. *Information technology – Security techniques – Cryptographic techniques based on elliptic curves*, 2002.
13. S. Janssens, J. Thomas, W. Borremans, P. Gijsels, I. Verhauwhede, F. Vercauteren, B. Preneel, and J. Vandewalle. Hardware/software co-design of an elliptic curve public-key cryptosystem, 2001.
14. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, vol. 48:203–209, 1987.
15. S. Kumar, M. Girimondo, A. Weimerskirch, C. Paar, A. Patel, and A. S.Wander. Embedded End-to-End Wireless Security with ECDH Key Exchange. In *Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems — MWSCAS 2003*, December 2003.
16. J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2 m) without precomputation. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES '99*, volume LNCS 1717, pages 316–327, Berlin, August 1999. Springer-Verlag.
17. V. S. Miller. Use of elliptic curves in cryptography. *CRYPTO '85*, pages 417–426, 1986.
18. NIST. *Recommended Elliptic Curves for Federal Government Use*, May 1999.
19. R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO '95*, volume LNCS 963, pages 43–56, Berlin, 1995. Springer-Verlag.
20. L. Song and K. K. Parhi. Low energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing*, 19(2):149–166, June 1998.
21. E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In *Asiacrypt '96*, volume LNCS 1233, pages 65–76, Berlin, 1996. Springer-Verlag.
22. A. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *CARDIS 2000*, Bristol, UK, September 20–22 2000. Kluwer.

Dynamic Prefetching in the Virtual Memory Window of Portable Reconfigurable Coprocessors

Miljan Vuletić, Laura Pozzi, and Paolo Ienne

EPFL I&C LAP
Swiss Federal Institute of Technology Lausanne
1015 Lausanne, Switzerland
{miljan.vuletic, laura.pozzi, paolo.ienne}@epfl.ch
http://lap.epfl.ch

Abstract. In *Reconfigurable Systems-On-Chip (RSoCs)*, operating systems can primarily (1) manage the sharing of limited reconfigurable resources, and (2) support communication between reconfigurable accelerators and user applications. It has been shown in previous work that the operating system can dramatically simplify the interface to reconfigurable coprocessors and isolate the programmer from all details of the hardware. A further potential of the operating system is developed here: the operating system can observe accelerators at runtime and dynamically take actions which improve their execution. The strength of involving the operating system consists in achieving better performance without any information from the end user and without changes either in the coprocessor hardware design or in the software application. Specifically, this paper presents an operating system module that monitors reconfigurable coprocessors, predicts their future memory accesses, and performs memory prefetching accordingly; the goal is to hide completely memory-to-memory communication latency. The module uses a lightweight hardware support to detect coprocessors memory access patterns. The effectiveness of the technique is demonstrated for two applications on an embedded RSoC board running the Linux operating system. Significant speedup is achieved compared to the nonprefetching version, and the improvement is obtained in a manner completely transparent to the application programmer.

1 Introduction

Reconfigurable accelerators, running on behalf of user applications, exploit the potentials of spatial computation in reconfigurable logic. It is a natural task for the *Operating System (OS)* to control the reconfigurable logic and facilitate its use. Reconfigurable resources can be shared, physically and virtually partitioned between applications [3,4,14]. By supporting the virtual memory address space sharing between an application and its coprocessor [12], the OS can enable a transparent way of interfacing: it hides the actual interface and automatically copies data from user memory to the coprocessesor memory and back.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 596–605, 2004.

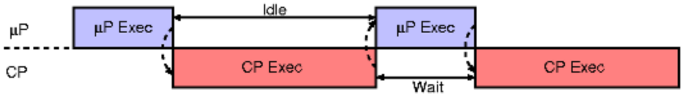


Fig. 1. Simple execution. The processor and coprocessor change in turn.

The OS is not only limited to providing resource sharing and transparent interfacing: it can survey the execution of the coprocessor, optimise communication, and even adapt the interface dynamically. A virtualisation layer makes such improvements possible without any changes in the application and coprocessor code. Although it is intuitively expected that the additional layer brings overheads, it is shown here that it can also lower execution time by taking advantage of run-time information. In this paper, the strength of delegating the interfacing tasks to the OS is presented. As opposed to the simple execution model shown in Figure 1, where the main processor is idle during the coprocessor busy time, we explore the scenario where the idle time is invested into anticipating and supporting future coprocessor execution: with simple hardware support, the OS can predict coprocessor memory accesses, schedule prefetches, and thus decrease memory communication latency.

This paper is organised as follows: In Section 2, the basic concepts of *Virtual Memory Window (VMW)* for reconfigurable coprocessors are presented. Section 3 discusses related work. The OS memory prefetching concept, its hardware and software design elements are presented in Section 4. Section 5 shows the measurements that prove the benefits of prefetching. Finally, some conclusions are given in Section 6.

2 Virtual Memory Window

Nonstandard programming paradigms and HW/SW interfacing models have certainly hindered the acceptance of reconfigurable computing. The *Virtual Memory Window (VMW)* addresses these problems [12] by reusing the simple and well-known concept of virtual memory. The VMW enables the coprocessors to share the virtual memory address space with user applications, thus simplifying the programming paradigm and hardware interfacing.

Figure 2 shows how a reconfigurable coprocessor is interfaced with the main processor. The OS provides a uniform and abstract virtual memory image hiding all details about the physical memory. The fast translation from virtual to physical addresses is enabled by hardware accelerators: (1) *Memory Management Unit (MMU)* in the main processor case, and (2) *Window Management Unit (WMU)* in the coprocessor case. The *Virtual Memory Manager (VMM)* and *Virtual Memory Window (VMW) Manager* in the OS ensure that the translation is transparent to the end users. In the same manner as the VMM copies pages between the mass storage and the main memory, the VMW manager copies pages between the main memory and the window memory. Both managers do the tasks *transparently* from the end user.

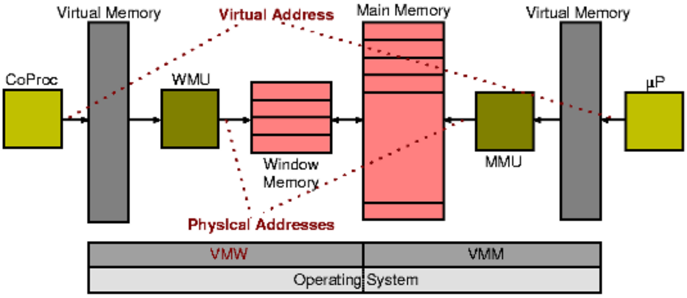


Fig. 2. Virtual Memory Window for reconfigurable coprocessor. The coprocessor and user application share the virtual memory address space. The Window Management Unit supports the address translation, which is managed by the OS.

Benefits of unifying the memory pictures from the main processor and the coprocessor side are: (1) programming software and designing hardware is made simpler—calling a coprocessor from a user application is as simple as a common function call, and designing the coprocessor hardware imposes no memory constraints but only requires complying to the WMU interface; (2) application software and accelerator hardware are made portable—hiding platform-related details behind the VMW manager and the WMU deliberates applications and coprocessor designs of platform dependence.

3 Related Work

Different approaches for virtualisation of reconfigurable resources are proposed [3,4]. The tasks of management and sharing the resources are delegated to the OS [7,14]. An orthogonal approach [12] that involves the OS to support interface virtualisation is used as the basis of the work presented in this paper.

Hardware and software prefetching techniques were originally developed for cache memories to support different memory access patterns. Stream buffers [5] were introduced (and later enhanced [8]) as an extension of tagged-based prefetching to improve sequential memory accesses. Other techniques exist that cover nonsequential memory accesses (e.g., recursive [9], and correlation-based [11] where a user-level thread correlation prefetching is shown). Hardware prefetching techniques have also been used for configurable processors [6]. Besides caching, prefetching techniques have been used for efficient virtual memory management: in hardware (e.g., speculatively preloading the TLB to avoid page faults [10]) and in software (prefetching virtual memory pages for user application [2]).

The technique presented in this paper is in its essence a dynamic software technique with limited hardware support. Its strongest point is the transparency: neither user applications nor hardware accelerators are aware of its presence.

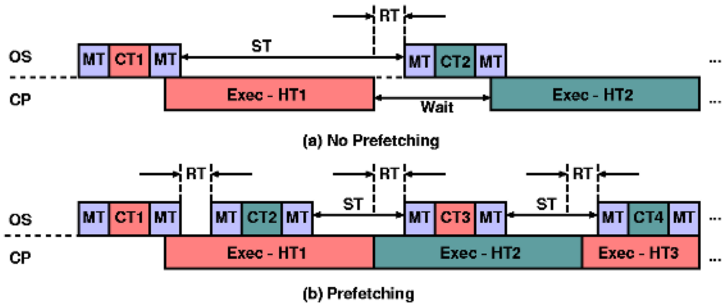


Fig. 3. The OS module (OS) activities related to coprocessor execution (CP). The OS manages VMW data structures (Management Time—MT) and copies pages from/to user memory (Copy Time—CT). When finished, it sleeps (Sleep Time—ST). The coprocessor executes (Hardware Time—HT) until it finishes or misses a page. In both cases, it waits for the OS action. The OS responds after some time (Reponse Time—RT). Instead of sleeping, the VMW can fetch in advance pages that may be used by the coprocessor. This results in uninterrupted coprocessor execution.

Even more importantly, an OS module is developed to optimise execution of coprocessors, and it is not limited to the presented prefetching technique.

4 OS-Based Prefetching

In this section, the basic motivation for applying OS-based prefetch techniques is presented. Afterwards, hardware and software requirements to implement a prefetching system for a VMW are discussed in detail.

4.1 Memory Copy Overhead

The sequence of the OS events during a VMW-based coprocessor execution is shown in Figure 3. Assuming a large spatial locality of coprocessor memory accesses (e.g., stream oriented processing), it can be seen in Figure 3a that the OS sleeps for a significant amount of time. Once the management is finished, the manager goes to sleep waiting for future coprocessor requests.

During idle time, the VMW manager could instead survey the execution of the coprocessor and anticipate its future requests, thus minimising the number of page misses. Figure 3b shows hardware execution time overlapped with the VMW management activities. During coprocessor operation, the WMU informs the manager about the pages accessed by the coprocessor. Based on this information, the manager can predict future activities of the coprocessor and schedule prefetch-based loads of virtual memory pages. If the prediction is correct, the coprocessor can use the prefetched pages without generating miss interrupts. In this way, the involvement of the operating system may completely hide the memory communication latency. *The approach requires no action on the software programmer nor on the hardware designer side.*

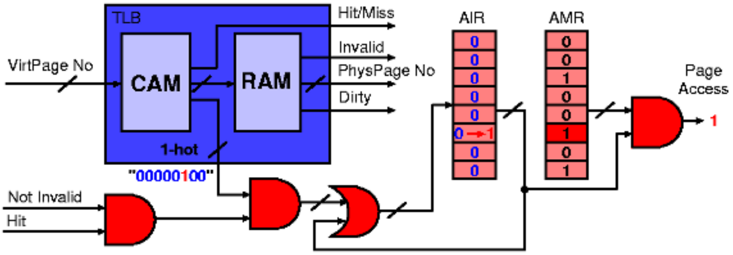


Fig. 4. Page access detection. On a hit in the *Content Addressable Memory (CAM)*, 1-hot bit lines will set the corresponding bit in the *Access Indicator Register (AIR)*. If the mask (*Access Mask Register (AMR)* allows the access, an interrupt is raised.

4.2 Hardware Support

The WMU provides hardware support for the translation of the coprocessor virtual addresses and for accessing the window memory. The window memory is divided into pages that map onto the different regions of the user memory. The optimal number of pages depends on the characteristics of the coprocessor memory access pattern. The WMU supports multiple operation modes—i.e., different page sizes and number of pages. A simple extension—two 32-bit registers and few tens of logic gates—to the WMU is introduced that supports the detection of a page access. Figure 4 contains the internal organisation of the WMU related to address translation. As in typical MMUs, address mapping is performed by a *Translation Lookaside Buffer (TLB)*. If there is a match in the *Content Addressable Memory (CAM)*, the 1-hot bit lines are used to set the appropriate bit in the *Access Indicator Register (AIR)*. If the OS wants to detect the first access to a particular page, it simply sets the correct mask in the *Access Mask Register (AMR)*. When the access appears, an interrupt is raised requesting OS handling. Nested interrupts are prevented by the OS resetting to 0 the appropriate mask bit. While the interrupt is being handled, there is no need to stop the coprocessor: interrupt handling and coprocessor run in parallel—space is left for speculative work. The OS actions need not be limited to this simple access detection mechanism. A more sophisticated but still reasonably simple hardware can be employed in order to support detection of more complex memory access patterns.

4.3 VMW Module

The three main design components of the VMW module are: (1) initialisation and interrupt handling, (2) prediction of future accesses, and (3) fetching of pages from main memory.

Interrupt Handling. Once invoked, the OS service first initialises the internal data structures and the WMU hardware. It then goes to sleep awaiting for interrupts. There are three possible interrupt types coming from the WMU: (1)

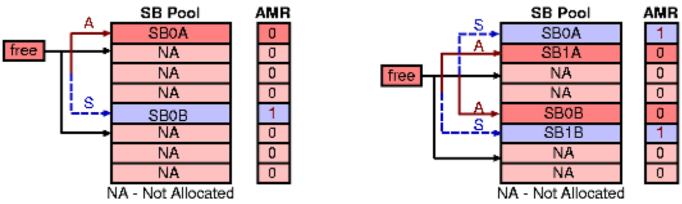


Fig. 5. Stream Buffer (SB) allocation and the AMR. On a miss, a pair of pages (SB) is allocated. The miss page is active (A); it is being accessed by the coprocessor; the prefetched page is speculative (S). The AMR helps to detect accesses to (S) pages.

finish, (2) miss, and (3) access. When *finished*, the module returns control back to the user application. If a *miss* appears, the load of the miss page is scheduled into the fetch queue. Afterward, the predictor is called to attempt predicting future page accesses and speculative pages are also scheduled for loading. The coprocessor is resumed by the fetcher, once all miss-related requests are satisfied. If an *access* appears, it indicates that the coprocessor accessed a page for which this information had been requested. The predictor is called to validate or confute its past predictions and schedule future page loads into the fetch queue. During access handling, the coprocessor is active.

The Predictor. It attempts to guess future memory accesses and to schedule page loading. The only input parameters to the predictor are miss addresses and access page numbers—i.e., there is no information about the state of the coprocessor. The approach is similar to classic prefetching techniques where no information is available about the instructions issued by the main processor [8] but only the addresses on the bus. The current predictor assumes that for each miss a new stream is detected; thus, it requires a stream buffer allocation (i.e., a pair of window memory pages, refer to Figure 5) and it schedules a speculative prefetch for the page following the missing one. By setting appropriately the AMR, it ensures that the WMU hardware will report the first access to the speculatively-loaded page. When the access is reported, the predictor is invoked again and, with this information confirming the correct speculation, further prefetches are scheduled. Each speculative prefetch is designated to its corresponding stream buffer. Ideally, for a correctly-guessed memory access stream and good timing of the prefetching, only one miss per stream should appear: all others misses should be avoided through prefetching.

Since the number of stream buffers is limited, the coprocessor may require more streams than it can be provided. In this case, a stream buffer should be selected for deallocation, to satisfy a new allocation request. For the moment, a simple eviction policy is implemented; yet, since the predictor is a software-only component, more sophisticated eviction policies can be easily added. Furthermore, potential trashing and deadlocks (due to the capacity problems of the window memory) can be resolved *dynamically and transparently for the end-user* simply by changing the operation mode of the WMU.

The Fetcher. The fetcher is responsible for loading pages from/to user space memory. The memory requests are scheduled by the miss handler and by the predictor, with miss-generated requests being always scheduled before speculative ones. The fetcher executes the fetch queue, until all the requests are serviced. It determines the type of fetch (mandatory or speculative), its destination in the window memory, and whether it requires a stream buffer allocation. If the destination is occupied by a dirty page, it is copied back to the user space. The page is then fetched from user memory and the request is deleted from the queue. The coprocessor can be resumed if needed—if the fulfilled request is miss-based and there are no outstanding miss-based requests.

5 Experiments

The system described is implemented on an Altera Excalibur based board with the EPXA1 device [1]. The device consists of an ARM processor with basic peripherals and a reconfigurable part. The ARM processor is running on 133MHz and executes user applications under the GNU/Linux OS. The WMU with access indication support, is synthesised from VHDL code (less than thousand lines) to reconfigurable logic and interfaced to the ARM processor using the Avalon bus [1]. A dual-ported 16KB on-chip memory is used for the VMW window memory. The VMW manager with prefetching support is implemented as a loadable kernel module (in a couple of thousands lines of C-code).

5.1 Results and Comparisons

Two applications are ported to the system: (1) IDEA cryptography application and (2) ADPCM decoder from MediaBench. For both applications, coprocessors have been designed (IDEA running at 6MHz and ADPCM running at 40MHz) complying to the WMU interface and implementing critical parts of the algorithms in hardware. Even without OS controlled prefetching, both applications achieve significant speed up compared to their software-only versions [12]. Notice that no change whatsoever has been made to the user C and VHDL code to take advantage of prefetching—the code is *exactly the same* that was developed in previous work [13,12], and only the WMU and the VMW manager differ.

Figure 6 compares total execution times of ADPCM decoder with and without prefetching in the VMW module. Although running at the same speed, in the prefetching case the coprocessor finishes its task almost twice as fast compared to the nonprefetching case. As indicated in Figure 3, the sleep time reduces: the module handles access requests in parallel with the execution of the coprocessor. Counterintuitively, the management time slightly decreases, because the number of miss-originated interrupts is dramatically lower (e.g., in the case of 32KB input data size it goes down from 48 to only 2). Meanwhile, multiple access-originated interrupts may appear within a relatively short time interval (e.g., two streams usually cross the page boundary at about the same time) and

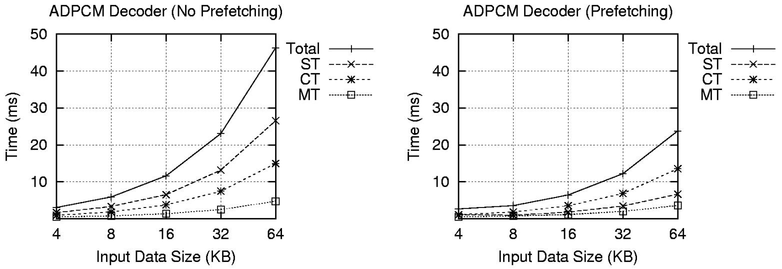


Fig. 6. ADPCM decoder: Total execution times with/without prefetching. The execution time consists of sleep time (ST), copy time (CT), and manage time (MT).

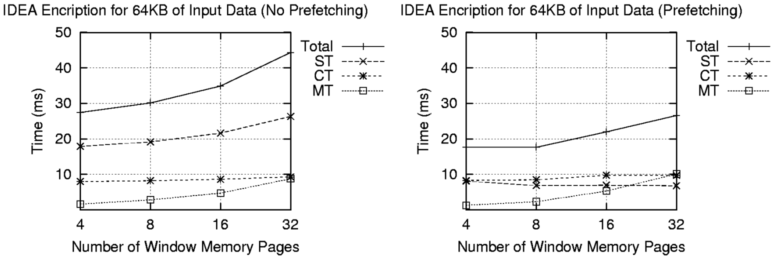


Fig. 7. IDEA encryption: total execution times with/without prefetching.

the VMW manger services them at the same cost. This is not the case for the misses: for a miss to appear, the previous miss needs to be already serviced.

The ADPCM decoder has a specific access pattern: the decoder is producing four times more data than it consumes. Due to the simple FIFO policy used for page eviction in the non-prefetching case, it may happen that a page still being used gets evicted: the page will need to be reread before the coprocessor continues execution. On the other hand, the prefetching approach with stream-buffer allocation is less sensitive to the applied page eviction policy because distinct stream-buffers are allocated for input and output streams.

Figure 7 shows the total execution times of IDEA encryption for different number of window memory pages. A significant improvement in the IDEA execution time is achieved with prefetching. Management time increases with the increasing number of window memory page, since larger data structures are managed. In the prefetching case, the management time is slightly larger than without prefetching. With smaller page sizes, manage and copy time intervals become comparable to the hardware execution intervals: increasingly often, the coprocessor generates a miss while the missing page is already being prefetched. This miss is called a late miss, and it is less costly than a regular one. Still, the VMW manager needs to acknowledge it once its corresponding prefetch is finished—hence the slight increase in the management time. Table 1 shows

Table 1. Interrupts by the IDEA coprocessor. In the prefetching case, beside misses, there are accesses (used to trigger prefetching) and late misses (when the missing page is already being transferred).

Number	Page	Nonprefetching		Prefetching	
of pages	size	Misses	Misses	Late Misses	Accesses
4	4KB	32	2	0	15
8	2KB	64	2	2	31
16	1KB	128	2	24	63
32	0.5KB	256	3	53	131

how the number of miss-originated and access-originated interrupts grows with smaller page sizes. It also shows how late misses start to appear.

Although it seems costly to manage larger number of window memory pages, in some cases the flexibility of the WMU and the VMW manager may be required, since the WMU operation mode can affect the performance. For example, supposing only two window memory pages, and prefetching, the coprocessor experiences memory trashing problems and performs dramatically slower then the non-prefetching one (e.g., for the IDEA encryption, on 16KB input data and two 8KB pages in the window memory, 757ms vs. 7ms, and 1366 vs. 6 misses!). It is the task of the VMW module to detect this misbehaviour and change to the operation mode that corresponds better to the coprocessor needs.

6 Conclusion

This paper presented an OS module supporting the execution of reconfigurable coprocessors running within the VMW framework. Not only it allows the coprocessors to share transparently the same address space with user applications, but it also makes possible advanced and yet simple runtime optimisations, *without any intervention by the end user.*

In order to demonstrate the presented concept, a stream-based memory prefetch technique was implemented within the OS module (with a simple hardware support in the WMU). A significant execution time improvement is demonstrated for two application-specific reconfigurable coprocessors, *without any change in either application software or coprocessor hardware.*

Future extensions of this work are not limited to implementing other prefetch techniques (e.g., recursive and correlation-based prefetching): the involvement of the OS enables novel runtime optimisations (e.g., changing the number and size of window memory pages in order to fit better application needs).

Acknowledgments. The authors would like to acknowledge and appreciate help and support provided by Nicolas Blanc and Cédric Gaudin.

References

1. Altera Corporation. *Altera Excalibur Devices*, 2003. `http://www.altera.com/`.
2. K. Bala, M. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *In the Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI, 1994)*, pages 243–53, Monterey, Calif., Nov. 1994. USENIX Assoc.
3. E. Caspi, M. Chu, R. Huang, J. Yeh, A. DeHon, and J. Wawrzynek. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. In *Proceedings of the 10th International on Field-Programmable Logic and Applications*, pages 605–14, Villach, Austria, Aug. 2000.
4. M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 10980–85, Munich, Mar. 2003.
5. N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–73, Seattle, Wash., May 1990.
6. H. Lange and A. Koch. Memory access schemes for configurable processors. In *Proceedings of the 10th International on Field-Programmable Logic and Applications*, pages 615–25, Villach, Austria, Aug. 2000.
7. V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *Reconfigurable Architectures Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, Apr. 2003.
8. S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, Ill., Apr. 1994.
9. A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–26, San Jose, Calif., Oct. 1998.
10. A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based TLB preloading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 117–27, Vancouver, British Columbia, Canada, June 2000.
11. Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–82, Anchorage, Ala., May 2002.
12. M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. In *Proceedings of the 41st Design Automation Conference*, pages 948–53, San Diego, Calif., June 2004.
13. M. Vuletić, L. Pozzi, and P. Ienne. Virtual memory window for portable reconfigurable cryptography coprocessor. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 2004.
14. H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 290–95, Munich, Mar. 2003.

Storage Allocation for Diverse FPGA Memory Specifications

Dalia Dagher and Iyad Ouaiss

Department of Computer Engineering
Lebanese American University
Byblos, Lebanon

Abstract. A previous study [1] demonstrates the advantages of replacing registers by FPGA embedded memories during the storage allocation phase of High-Level Synthesis. The trend in new FPGAs to have large amounts of on-chip embedded memories motivated this proposition and resulted in substantial area decrease in the synthesized designs. This paper elaborates further on the various possibilities involved during storage allocation onto embedded memories, and presents new memory binding techniques. These techniques include modifications to the memory mapping procedure presented in [1] and cater to various memory specifications. The embedded memories differ in their assumptions of the number of memory banks, the number of ports on each bank, and the read/write types of each port. The paper highlights the benefits of the new techniques and discusses the pros and cons involved in each case. The Discrete Cosine Transform (DCT) benchmark illustrates the area improvements obtained in the new approaches compared to conventional register binding (up to 47%). The results are evaluated through an analysis of both area and delay performances.

1 Introduction

Datapath Synthesis in Field Programmable Gate Arrays (FPGAs) is currently an important and wide area of research. Synthesis of digital systems includes the two main tasks of scheduling and allocation. The scheduling step consists of deciding on the number of functional units (FUs) needed in the design, determining the total number of control time steps needed, and specifying which control step corresponds to each operation. The datapath allocation step achieves three objectives: first, register allocation binds variables to registers or register files, then operation assignment assigns operations in the design to functional units, and third, interconnection allocation determines interconnections between registers and functional units whether in form of buses or multiplexers ([2], [3], [4], [5], and [6]). Conventional register allocation techniques ([9] and [10]) try to minimize the number of registers used for storage in the design; however, since all new FPGAs contain large amounts of embedded memory, new high-level synthesis techniques are sought to efficiently utilize these memories. Therefore, the goal of storage allocation should not be limited to minimizing the number of registers, but to mapping the design variables onto the embedded memories. Thus, flip-flops in the FPGA logic elements would be saved for use by the functional units of the design. This paper discusses various choices for

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 606–616, 2004.

storage allocation onto memories by motivating the need for new techniques; by performing a literature survey on related work; by introducing the concepts of the new techniques; by presenting the implementations of the new approaches illustrated with an example; by discussing the results obtained; and finally, by a conclusion.

2 Motivation

A previous research effort demonstrated the possibility of using memories instead of registers in storage allocation [1] especially with the current trend that all new FPGAs contain a large amount of embedded memory. There are many reasons why binding to the on-chip memory banks is needed to replace register binding such as the following:

- The need for larger storage area increases with larger designs that have more variables, but registers do not provide these required large areas.
- Registers use the flip flops or latches of FPGA logic elements whereas using the embedded memories saves these logic resources for the functional unit allocation needs and reduces placement and routing congestion on the logic elements.
- Memory allocation improves performance because its criteria for binding change from those for registers. Register binding imposes restrictions like accepting only variables with non-overlapping lifetimes, whereas memories accept overlap and check only for simultaneous read or write conflicts.
- Memory mapping also reduces interconnection complexity and area cost because a memory bank can incorporate more variables at a time than one register which decreases the number of multiplexers used at storage inputs in the design.
- In the case of multi-port memories, parallel access to variables from the same memory is possible which could not be achieved from the same register.

3 Related Work

Several studies were previously conducted in the field of using embedded memory banks as storage units in datapath synthesis. Stok [7] presented a method for grouping variables before assigning them to registers and thus generating memory modules. The technique starts with an edge coloring algorithm, and after the variables are grouped, the total interconnection is minimized by using simulated annealing. This technique limits its use of memories to single port memories, but it showed that when variables are grouped into register files and operations are assigned to modules, savings in the number of interconnections are observed but at the expense of additional register area. Tseng and Siewiorek [9] presented a method for memory allocation that is based on building a graph to represent the concurrent access requirements of the variables and then finding a clique partition of the vertices of the graph. This technique follows at first the same sequence of steps used to map variables to registers. In other words, it takes the same assumptions for the variable lifetime condition and maps variables to registers, and at the end of this process it tries to group registers into sets of scratch pad memories provided that variables having disjointed access times can be grouped together. This method does not take into account minimizing interconnection costs and is also limited to single port memories.

Other techniques were developed that target multi-port memories. Balakrishnan [12] elaborated on the use of multi-port memories in datapath allocation where registers were grouped based on a 0-1 integer linear programming formulation. Memories with read ports, write ports, and read/write ports were considered. Ahmad and Chen [2] extended this work to find the minimal number of multi-port memory banks required. This technique also minimizes the interconnection costs using a 0-1 ILP formulation while minimizing the total design area. Kim and Liu [8] used multi-port memories stressing on the importance of minimizing interconnection costs before grouping variables into memories. This approach also does not allow multiple copies of variables to get mapped to more than one location. The goal is to minimize the cost function that includes interconnection cost as well as memory cost. A common goal for these memory binding techniques is to occupy a minimal number of memory banks. This objective, although essential in ASIC designs, is not as important in FPGA designs where the number of available banks is fixed.

Luthra et al. [14] presented a memory mapping methodology that assigns large data elements to shared memories. The target is to place data in the FPGA embedded memories so that it can be easily accessed by both hardware and software. These HW/SW co-designs among general purpose processors and FPGA platforms are used for compute-intensive tasks. However, the approach focused on showing the speedup obtained through acceleration in hardware over software. No performance evaluation was done to compare the implemented memory binding with conventional register binding.

Al Atat and Ouaiss [1] proposed a technique for memory binding that allows variables to be assigned to the same memory if they are accessed in different scheduled time steps. The technique capitalizes on the limitation of register binding where variables cannot be assigned to the same register if their *lifetimes* overlap. In memory binding, even if the lifetimes overlap, the variables can still be assigned to the same memory bank provided that the *access times* do not overlap. Furthermore, the technique highlights the savings in multiplexer area and reduction in logic area occupied by registers. The approach, however, is limited to banks with one read port and one write port. Moreover, the goal to minimize the number of memory banks utilized, as described above, might yield sub-optimal solutions in terms of both area and delay.

4 Memory Binding Implementations

This paper elaborates on the memory binding technique presented in [1]. The technique was based on the assumption that embedded memories have exactly one read port and one write port. It starts by assigning variables to the first available memory bank. For each memory, two lists keep track of the time steps in which variables are read and the time steps in which variables are written. Thus, when a variable is assigned to a memory, its read and write times should not conflict with any of the items in these two lists. When there are no more variables satisfying that condition, variables would be assigned to another memory bank for which two new lists would be created. The process is repeated until all the variables are assigned to memories. At that point, the interconnection between the memories and the functional units would be determined according to the read and write times of the variables. The behavior of that technique is illustrated through an example Fig. 1 and the results are shown in Fig. 2. Fig. 1 shows the Scheduled Control Data Flow Graph (CDFG) for a resource bag of three adders and two multipliers as functional unit resources. The labels *add#* and *mult#* on the figure show the results of the FU assignment. The implementations of all subsequent techniques will be shown on this example. In all the figures showing the output of memory binding: the FUs consist of the adders and multipliers of the example with their outputs named according to FU numbers; labels next to FUs show the operations allocated to each corresponding FU; the memory banks show the variables mapped to them and their outputs are named according to the memory number and the port number in the case of multi port memories; and the multiplexers take for inputs variables, FU outputs, or memory outputs and have their outputs connected to memory or FU inputs.

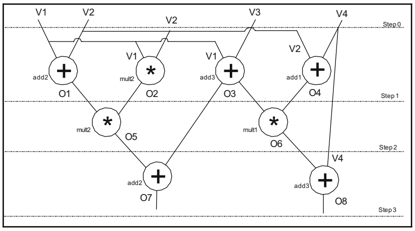


Fig. 1. Scheduled CDFG

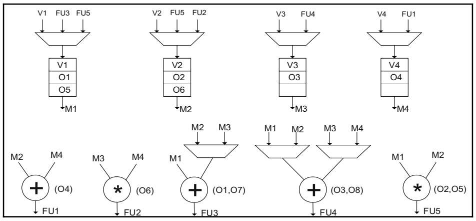


Fig. 2. Output of Memory Mapping

This paper discusses two new memory binding techniques that target various memory specifications: dual-ported memory banks are first targeted in a technique that is different from the one used in [1] since it distributes the variables on all banks of the FPGA. Second, the general case of multi-read and multi-write port memory banks is presented. This latter approach also targets banks with single R/W ports as well as multi bi-directional R/W port memories.

4.1 Distributed Mapping onto Dual-Ported Memories

This proposed algorithm extends the memory mapping technique presented in [1]. Although it caters to the same type of memory structures (one read and one write ports), it assigns variables using a different approach. The algorithm takes advantage of all embedded memory banks on the target FPGA by trying to intelligently distribute the variables onto all available banks. In the *distributed memory mapping* approach variables are assigned to different memory banks. If there are more variables than memory banks, the algorithm assigns the remaining variables provided their write and read access times do not conflict with pre-assigned variables in the contemplated bank. The drawback of this approach is that it can complicate the chip routing since all on-chip memory banks are utilized. Such a technique would be useful if only one design was implemented on the FPGA. However, if multiple designs or cores were to be mapped onto the FPGA, the target would be to optimize the number of used memories. On the other hand, the number of multiplexers at the inputs of the memories is reduced since fewer variables are assigned to each bank. Furthermore, at the output of the memory units, the interconnection complexity depends on the datapath implementation. For implementations with large resource bags (light FU re-use), the number of multiplexers at the output of the banks is reduced to a minimum. However, if the datapath implementations include small resource bags (heavy FU re-use), the mapping algorithm would generate additional interconnection since variables distributed on different banks need to connect to the same functional unit. For the example of Fig. 1, the output of distributed memory mapping is shown in Fig. 3.

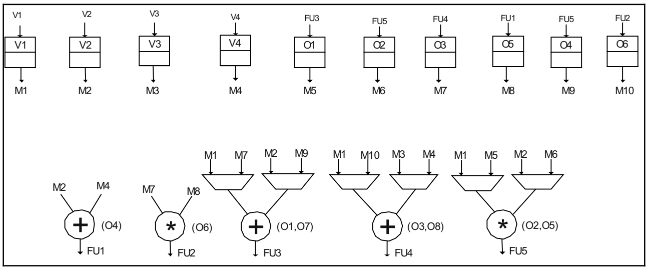


Fig. 3. Result for Distributed Memory Mapping

4.2 Multi-read/Multi-write Port Memory Mapping

The second mapping variation targets memory banks with each *M* read and *N* write ports. This technique, *multi-port memory mapping*, allows more variables accessed in the same time step to be allocated in each memory bank. However, the complexity of allocating variables to specific ports becomes evident. For each port, the implemented technique assigns a subset of mapped variables while ensuring that there are no conflicts between the variables' read and write access times and that the number of read and write ports do not exceed *M* and *N,* respectively. The assignment of a variable occurs in one step: the target memory bank and the target port within that memory bank are selected.

Fig. 4 shows the distributed memory mapping algorithm. VarList is a list of all variables in the scheduled CDFG where each variable has a write_time and a list of read_times. The algorithm allocates for each memory bank a list of write ports and a list of read ports to keep track of the variable accesses for each of the ports. The procedure iterates through all variables in VarList trying to fit variables that do not conflict with each other in the same memory.

The algorithm shows how the variables are mapped to ports as part of mapping the variables to memories. First, a search is performed to find one write port where all variables assigned to it do not conflict with the write time of the current variable. Then, for every read time of the current variable, a search is performed to find one read port with no read time conflict. If the necessary ports are found, the variable is assigned to the memory. This process repeats for as many memories as needed until all variables in VarList are assigned.

```
While (VarList <> Ø) {
 List Memory;
 Memory is an empty list;
 For (all variables in VarList) {
   V= next variable of VarList;
   List V_ReadPorts;
   Flag Valid_Write;
   V_ReadPorts is an empty list;
   For (all write_ports of Memory) {
      W = next write port of Memory;
      Valid_Write = false
      If (Write_time of V not found in W.write_timelist) {
          Valid_Write = true;
          Break out of for loop;
      }//end if
   }// end for loop
   If (Valid_Write) {
     For (all Read_times of V) {
          Rtime = next Read_time of V;
          For (all read_ports of Memory) {
            R = next read_port of memory;
            If (Rtime not found in R.read_timelist) {
              Add R to V_ReadPorts;
            }//end if
          }//end for loop
      }//end for loop
      If (# of ports in V_ReadPorts = # of Read times of V) {
            Add Write_time of V to W.write_timelist;
            Add All Read_times of V to respective read_timelists
                              of ports in V_ReadPorts;
            Add V to Memory;
            Remove V from VarList;
      }//end if
   }//end if
 }//end for loop
 Add Memory to list of all Memories;
}// end while loop
```

Fig. 4. Multi Port Memory Mapping Algorithm

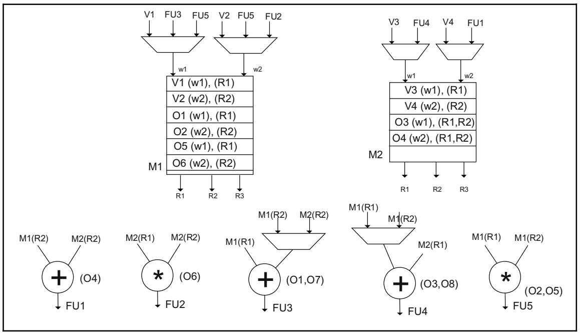


Fig. 5. Result for Multi Port Memory Mapping

At the end of this algorithm, the number of multiplexers needed at the inputs and outputs of each memory can be readily determined. The output of this technique is illustrated in Fig. 5.

The general memory mapping approach presented in this section can be adapted to cater to single-port memory banks (banks with a single read/write port) and to multiple read/write-port banks (banks with bi-directional ports). The outputs of these two variations are shown in Fig. 6 and Fig. 7 respectively. The algorithms for these two variations as well as the one for the technique described in Section 4.1 are not shown since they can be deduced by refining the algorithm of Fig. 4. However, their results will be analyzed in the following section.

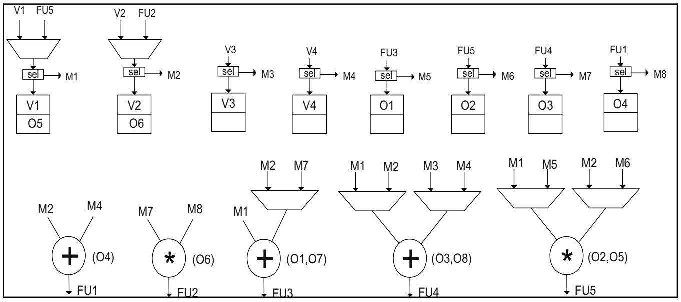


Fig. 6. Result for Single Port Memory Mapping

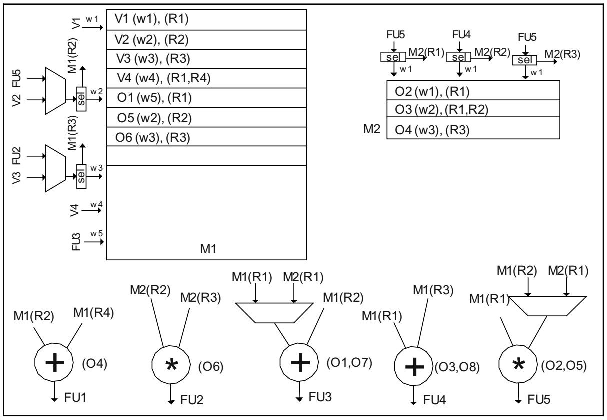


Fig. 7. Result for Multi R/W Port Memory Mapping

5 Results

The Discrete Cosine Transform (DCT) benchmark was used to validate the techniques implemented in this paper. Specifically, the equation of one of the 4x4 variables in the DCT matrix was implemented, and it consisted of 35 nodes (or operations). Area and speed results presented below were obtained from the Altera MAX+Plus II software and characterized for the Altera EP1K100FI484-2 device of the ACEX1K family [13].

5.1 Area Analysis

The results of the four techniques along with the technique presented in [1] were compared to the results obtained by the Left Edge register binding algorithm [10]. As seen in Table 1, all techniques decreased the overall area (in terms of logic cells) occupied by the design.

Table 1. Gain over Register Binding Using Left Edge Algorithm

+	x	# of Memories	MUX Area	Total Area	Register Area	MUX Area	Total Area	MUX gain	total gain
Dual Port Memory Mapping [1]					*Left Edge Register Binding*			*% Area Gain*	
1	1	20	400	1160	320	880	1960	55	41
4	16	20	752	12552	320	800	12920	6	3
Distributed Mapping (32 memories)					*Left Edge Register Binding*			*% Area Gain*	
1	1	32	688	1448	320	880	1960	22	26
4	16	32	608	12408	320	800	12920	24	4
Single R/W port Memory Mapping					*Left Edge Register Binding*			*% Area Gain*	
1	1	20	496	1256	320	880	1960	44	36
4	16	36	656	12456	320	800	12920	18	4
Multi R/W Memory Mapping (5 R/W ports)					*Left Edge Register Binding*			*% Area Gain*	
1	1	4	288	1048	320	880	1960	67	47
1	16	8	288	11998	320	864	12894	67	7
4	16	8	688	12488	320	800	12920	14	3
Multi Read Multi Write Port Memory Mapping (3 Read / 2 Write ports)					*Left Edge Register Binding*			*% Area Gain*	
1	1	10	304	1064	320	880	1960	65	46
4	16	10	784	12584	320	800	12920	2	3

Table 2. Dual Port vs. Distributed Dual Port Memory Mapping

		Dual Port Memory Mapping [1]			*Distributed Mapping (32 memories)*			*% Change*	
+	X	# of Memories	MUX Area	Total Area	# of Memories	MUX Area	Total Area	Change in MUX Area	Change in Total Area
1	1	20	400	1160	32	688	1448	-72	-25
1	16	20	592	12302	32	416	12126	30	1
2	1	20	400	1190	32	688	1478	-72	-24
2	16	20	736	12476	32	544	12284	26	2
3	1	20	400	1220	32	688	1508	-72	-24
3	16	20	832	12602	32	640	12410	23	2
4	1	20	400	1250	32	688	1538	-72	-23
4	16	20	752	12552	32	608	12408	19	1

The results of applying the distributed memory mapping technique, shown in Table 2, display a decrease in the multiplexer area when large resource bags are used. This gain over the dual port memory mapping technique [1] is due to the use of more banks. Since the allocated variables were distributed over more memories, fewer multiplexers were needed at memory inputs. Thus, the best results appeared with larger resource bags and reached up to 30 % gain in multiplexer area. Knowing that with the distributed technique fewer variables are allocated per memory and more memories are used, it is expected that more multiplexers will be required at the inputs of the functional units. Accordingly, with the smallest resource bags, the results show a loss (up to 72%) in the total multiplexer area. With more functional units available, more operations are executed in parallel and fewer multiplexers are required.

5.2 Speed Analysis

Preliminary speed estimates were obtained for the memory mapping techniques through the use of the timing analyzer capability of the MAX+Plus II software and the reported maximum clock speed was observed. The targeted device has a flip-flop delay of 0.8ns and a memory delay of 2.8ns, but the effectiveness of using memories instead of registers despite this additional memory latency will be justified in the following analysis. Table 3 shows the speed performance of simple designs that compare use of registers versus use of embedded memory banks. The two used designs only differ in their implementation of the storage components, where the registers were solely interchanged with memory banks with a one-to-one register to memory correspondence. The same design was implemented for 1, 8, 16, 32, and 64 bit operands corresponding to the rows in the table. As the variable sizes increase, the delay overhead of using memory accesses vanishes. When single-bit variables are used, the registers outperform the memories since the placement and routing of logic is constrained to a very compact section of the device. On the other hand, with growing variables, the registers tend to spread onto multiple logic cells whereas memories handle variable width operands (up to 16 bits in this device) within the same physical block, thus limiting additional interconnection delays.

Table 3. Delays of Register vs. Memory Designs

Number of Bits	Delays in Register Designs (ns)	Delays in Memory Designs (ns)	% Gain of Memory over Register
1	4	9	-125
8	16.9	19.8	-17
16	36.7	35.7	3
32	76.6	71.9	6
64	145.3	129.8	11

Thus, besides the difference between registers and memory banks, the functional units and multiplexers used in both designs were the same. Therefore, the delay gain in the memory design over the register design is only attributed to the distribution of the registers over multiple logic elements.

An important factor in the acquired delay values is the device used: with higher RAM density FPGAs, such as the Stratix II, it is expected to obtain better delay results. This is due to the fact that memory banks are closer to the logic elements, and hence, the difference in placement delays between register binding and memory binding should diminish.

In order to compare the effectiveness of the binding techniques, a simple design was selected, and storage allocation was performed with both register binding as well as memory mapping. The memory mapping outperformed the register binding by an area gain of 35% while a speed loss of 16%. Although the speed loss was considerable, the preliminary speed performances shown in Table 3 suggest that with larger bitwidths, the delay overhead disappears. Moreover, an initial timing analysis of the DCT benchmark shows a speed loss of 19% for 8-bit data compared to about 200% loss for 2-bit data. Due to the limitation of the software used, designs with larger bitwidths did not place/route on the largest supported device; however, the results obtained so far are promising.

6 Conclusion

This paper elaborated on the possibilities of using FPGA embedded memory banks to allocate storage of variables in a design. Several memory specifications were considered and corresponding memory binding algorithms were developed. From the types of memory structures contemplated, single port memories having only one R/W port, proved to be not viable with scarce resources and caused high interconnection area and used large numbers of memories; however with enough resources, variables were distributed over a larger number of memories and thus gave an area gain over the dual port memory mapping technique [1].

Encouraging results were obtained for delay estimates obtained in memory mapping. The overhead in time delay introduced due to register accesses being replaced by memory accesses tends to disappear as the design becomes bigger and wider signals are used.

Future work involves improving routing and placement costs caused by the implemented memory mapping techniques. For example, there is need to implement techniques to reduce wastage of unused bits in memory words by wisely selecting the memories' width and depth configurations. Furthermore, when memory mapping is used, careful consideration should be given to modifying the designs' controllers in order to generate memory addresses while not affecting the design's speed. Finally, delay analysis is required using higher-density devices or devices from other families and manufacturers.

References

[1] H. Al Atat and I. Ouaiss, "*Register Binding for FPGAs with Embedded Memory*," Proceedings of the Twelfth IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press, April 2004. (*accepted for publication*)

[2] I. Ahmad and C. Y. Roger Chen, "*Post Processor for Data Path Synthesis Using Multiport Memories*," Proc. IC-CAD'91, pp.276-279, 1991.

[3] M. C. McFarland, A. C. Parker, and R. Camposano, "*Tutorial on high-level synthesis*," Proc. 25th Design Automation Conf., 1988, pp. 330--336.

[4] G. De Micheli, "*Synthesis and Optimization of Digital Circuits,*" McGraw-Hill, New York, 1994.

[5] D. Gajski, N. Dutt, A. Wu, and S. Lin, "*High-Level Synthesis*," Kluwer Academic Publishers, Boston, 1992.

[6] Young-Long Lin, "Survey Paper: Recent developments in high-level synthesis," ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 1, January, 1997, pp 2-21.

[7] L. Stok, "*Interconnection Optimisation during Data Path Allocation*," Proc. EDAC, pp.141-145, 1990.

[8] T. Kim and C. L. Liu, "*Utilization Multiport Memories in Data Path Synthesis*," 30th ACM/IEEE Design Automation Conference, pp.298-302, 1993.

[9] C.-J. Tseng and D. P. Siewiorek, "*Automated Synthesis of Data Paths in Digital Systems*," IEEE Trans. CAD, vol. CAD-5, no.3, pp.379-395, July 1986.

[10] F.J. Kurdahi and A.C. Parker, "*REAL: A Program for Register Allocation*," Proceedings of the 24th Design Automation Conference, pp. 210-215, 1987.

[11] P. Ellervee, M. Miranda, F. Catthoor, and A. Hemani, "*Exploiting Data Transfer Locality in Memory Mapping*," 1999.

[12] M. Balakrishnan et al., "*Allocation of Multiport Memories in Data Path Synthesis*," IEEE Trans. CAD, vol. 7, no. 4, pp.536-540, April 1988.

[13] Altera Corporation, *"http://www.altera.com"*

[14] M. Luthra, S. Gupta, N. D. Dutt, R. Gupta, and A. Nicolau, "*Interface Synthesis using Memory Mapping for an FPGA Platform,*" 21st Intl. Conference on Computer Design, 140-145, October 2003.

Real Time Optical Flow Processing System

Javier Díaz, Eduardo Ros, Sonia Mota, Richard Carrillo, and Rodrigo Agis

Departamento de Arquitectura y Tecnología de Computadores, E.T.S.I. Informática,
Universidad de Granada, Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain
{jdiaz, eros, smota, rcarrillo, ragis}@atc.ugr.es

Abstract. We describe an optical flow processing system that works as a *virtual* motion sensor. It is based on an FPGA device; this enables the easy change of configuring parameters to adapt the sensor to different motion speeds, light conditions and other environment factors. We call it virtual sensor because it consists on a conventional camera as front-end and a processing FPGA device which embeds the frame grabber, the optical flow algorithm implementation, the output module and some configuring and storage circuitry. To the best of our knowledge this paper represents the first description of a fully working optical flow processing system that includes accuracy and processing speed measurements to evaluate the platform performance.

1 Introduction

Optical flow algorithms have been widely described in the literature. Some authors have addressed a comparative study of their accuracy on synthetic sequences [1]. Their evaluation using real world sequences is difficult to address because the real optical flow of such sequences is unknown. We have chosen to implement a classical gradient model based on work done by Lucas & Kanade (L&K) [1, 2]. Several authors have pointed out the good trade-off between accuracy and efficiency of this model that is an important factor to decide which approach is more suitable to be implemented as a real time processing system. For example, in [1] L&K's algorithm provides very accurate results. Liu et al. [3] evaluate the efficiency vs. accuracy trade-off of different optical flow approaches and outline that L&K is a good candidate. Finally, McCane et al. [4] also give L&K a good score and conclude that this approach requires an affordable computational power. This has motivated some other authors to focus on the L&K algorithm [5, 6].

In this paper we describe the hardware implementation of the L&K algorithm. There are other authors that have described recently the hardware implementations of optical flow algorithms [7, 8, 9], but most of them do not provide results to evaluate the performance of the system, i.e. the accuracy and the computation speed. This approach is a fully working system at conventional camera frame rates of 30 Hz, with images sizes of 320x240 pixels. To the best of our knowledge, this is the first description of such a system and therefore represents the state of art in this area.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 617–626, 2004.

2 Optical Flow Model

Although the original algorithm was proposed as a method to estimate disparity map in stereo pair images [2], we have implemented the Barron's description of L&K algorithm that can be applied to optical flow computation [1]. Several modifications have been added to improve the hardware implementation feasibility. Instead of using temporal FIR filters, we have implemented IIR filters as described in [10]. A FIR approach is also feasible in the used prototyping platform because it includes four independent memory banks. But a IIR approach is much more easy to configure for different time constants (only a single coefficient needs to be modified). On the other hand, the FIR approach requires variable memory resources and memory accesses depending on the constant of the temporal filter.

Other modification is adopted to provide estimations when the aperture problem appears. In this situation, as described in [11], we can add a small modification that allows to provide an estimation in the maximum gradient direction.

In the following equations, we describe briefly the computations in which is based the L&K approach. We will refer to these computational stages when describing the system architecture. A detailed description of the L&K model is provided in [1,2]. The algorithm belongs to the gradient based techniques that are characterized by gradient search performed on extracted spatial and temporal derivatives. Making the assumption of constant luminance values across the time, L&K method constructs a flow estimation based on first order derivatives of the image. By least squares fitting, the model extracts the motion estimation based on the hypothesis of similarity of velocitiy values in a neighbourhood of a central pixel. *W(x)* is a window that weights the constraints with higher weights near the centre of the spatial neighbourhood Ω.

The known solution to this problem is:

$$\vec{v} = \left[A^T W^2 A\right]^{-1} A^T W^2 \vec{b} \tag{1}$$

$$\begin{aligned} A &= \left[\nabla I(x_1),....,\nabla I(x_n)\right] \\ W &= diag\left[W(x_1),....,W(x_2)\right] \\ \vec{b} &= -\left[I_t(x_1),....,I_t(x_n)\right] \end{aligned} \tag{2}$$

An inherent limitation of these models appears in blank wall or aperture problem situations. In these cases, the problem has no solution (matrix A^TW^2A is not invertible) and the model can not provide any motion estimation. For this reason, we have added a small constant α to the matrix diagonal according to [9] that allows us to estimate the normal velocity field in situations where 2-D velocity can not be extracted due to the lack of contrast information. Therefore, the term of equation (1) is computed with expression (3).

$$\mathrm{A^T W^2 A} = \begin{bmatrix} \sum\limits_{x\in\Omega} \mathrm{W^2 I_x^2} + \alpha & \sum\limits_{x\in\Omega} \mathrm{W^2 I_x I_y} \\ \sum\limits_{x\in\Omega} \mathrm{W^2 I_x I_y} & \sum\limits_{x\in\Omega} \mathrm{W^2 I_y^2} + \alpha \end{bmatrix} \tag{3}$$

Summarizing, we have to compute the 2x2 matrix of equation (3), its inverse and the 2x1 matrix indicated in expression (4).

$$B = A^T W^2 \vec{b} = \begin{bmatrix} -\sum_{x\in\Omega} W^2 I_x I_t \\ -\sum_{x\in\Omega} W^2 I_y I_t \end{bmatrix} \quad (4)$$

The Gaussian smoothing is the pre-processing stage before the computation of the image derivatives in the matrices elements of equations (3) and (4). It reduces image noise and generates higher correlation between adjacent pixels. Typically, Gaussian spatio-temporal filters of 2 pixels variance plus a temporal derivative of 5 pixels are used. All the temporal operations require storage of 15 images for the entire process. This is hardly affordable in embedded systems; therefore, as indicated in [10], a more efficient implementation can be implemented using IIR temporal recursive smoothing and derivative filters. In this way, the temporal storage requirement is reduced to 3 frames, and the computation time improved, at cost of a slightly reduced accuracy. For an extensive discussion about how to design the IIR filters see [10].

3 Hardware Implementation

Nowadays software real-time computation of simple optical flow algorithms for small images is possible due to the outstanding computational power of the PCs. The drawback is that is difficult to adapt these systems to be used as embedded solutions. In the presented approach the motion computation chip can be regarded as part of a smart sensor. Alternatively, several hardware technologies can be used to implement an embedded system. The use of specific integrated circuits (ASIC) can achieve sufficient power to allow real-time computation but they are an expensive option. DSPs represent a valid alternative but if we need considerable computational power, the DSP solution is not powerful enough and a multiprocessor scheme needs to be designed [12]. The solution we propose is based on the use of programmable logic circuits (FPGAs). These circuits allow us to design a customized DSP circuit in a single chip of high computational power due to an intensive use of their intrinsic parallelism and pipeline resources. As we will show in later sections, the solution we propose uses this technology to implement a real-time hardware device capable of working as a PC coprocessor or smart sensor in embedded applications.

For our design we have used the RC1000-PP board from Celoxica [13] and Handel-C [14] as the hardware specification language. This board is connected to the PC by a PCI bus and it can be used as hardware accelerator board or as prototyping board. It contains a 2 million gates Virtex-E FPGA and four 2 MB SRAM memory banks accessible in parallel.

3.1 System Implementation Overview

The efficient implementation of the algorithm onto a FPGA device requires the efficient exploitation of the intrinsic processing parallelism of this kind of device. We use segmented pipeline architecture as shown in Fig. 1.

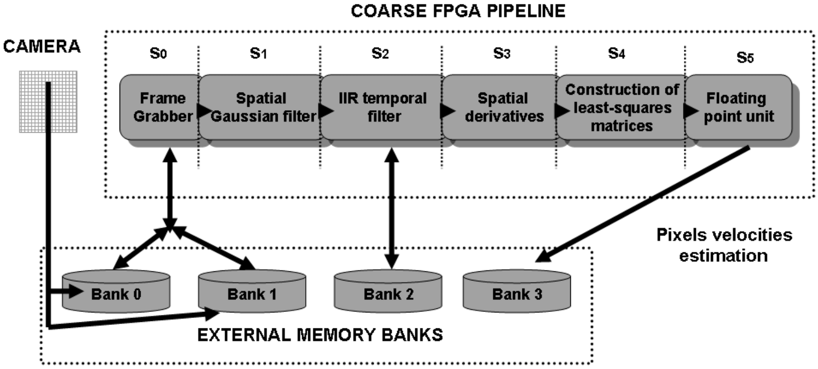


Fig. 1. Coarse pipeline processing architecture.

The basic computational stages in Fig. 1 can be summarised as follows:

S_0. Frame-Grabber receives the pixels from the camera and stores them in one of the external memory banks using a double-buffer technique to avoid temporization problems.

S_1. Spatial Gaussian filters smoothing stage.

S_2. IIR temporal filter affords temporal derivative and spatio-temporal smoothed images.

S_3. Spatial derivatives stage.

S_4. Construction of least-square matrices of equations (2) and (3).

S_5. Custom floating-point unit. Final velocity estimation need the computation of a matrix inversion, which requires a division operation. At this stage the resolution of the incoming data bits is significant and expensive arithmetic operations are required. Fixed point arithmetic becomes then too expensive and therefore we have designed a customized floating-point unit.

The computation bit-depth increases throughout the pipeline structure. For example, for a high precision system, with low degradation, we use 8 bits in the two first stages, 12 bits in the third and fourth stages, 24 in the construction of least-square matrices and 25 bits for the floating-point unit although a less hardware expensive approach has been tested with good results. The computation of the least-square matrices (S_4) is the most expensive stage in computational resources. Different parallelism strategies can be adopted at this point.

Basic parameters of the pipeline structure are latency (L) and the maximum number of cycles (MNC) required in the longest stage, which is the limiting factor of the computing speed. The circuit scheme gives us a basic relationship between the MNC and the system frequency clock (f_{clk}) to know the computing speed in pixels per second (pps), i.e. $\mathbf{pps=f_{clk}/MNC}$.

Due to the expensive requirements of the stages 5 and 6, the following subsection focus on their implementation and architectural design strategy.

3.2 Least Square Matrices Construction

This is a critical stage where the trade-off between efficiency and cost can be widely studied. Equations (2) and (3) require the generation of five products: I_x^2, I_y^2, I_xI_y, I_xI_t, I_yI_t. Then we have to make a weighted sum in a window (W) over a neighbourhood of size w_x by w_y. Due to memory limitations we save the I_x, I_y, and I_t values instead of the five crossed products. Therefore, the operations to do are: a) products computation for all the elements in a neighbourhood. We need to do 5 x w_x x w_y multiplications. b) Row convolutions operation. We have 5 x w_y convolutions to do and c) Column convolutions operation. It requires the computation of 5 convolutions.
The scheme of these operations can be seen in the Fig. 3.

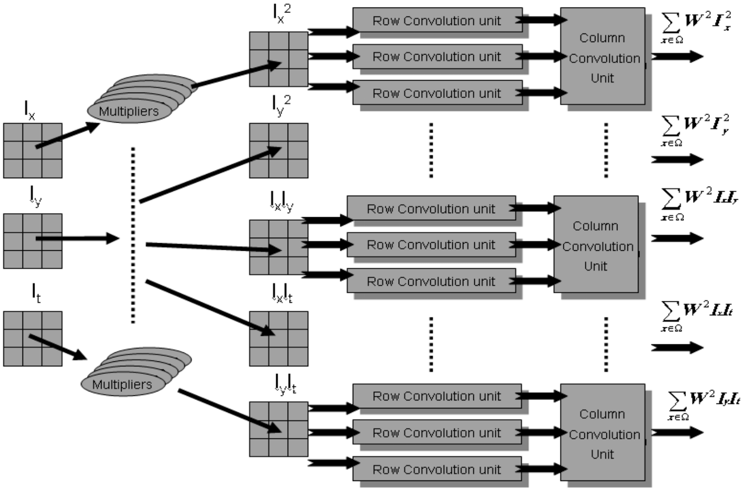


Fig. 3. Leas squares matrices circuit builder for a 3x3 neighbourhood.

This is an important stage where we can bias the trade-off between efficiency and hardware cost. The important parameters to choose are: neighbourhood weighted sum area, number of multiplication units and number of row-column convolution units.

For example, if we use a 3x3 neighbourhood, we can use between 1 to 45 multipliers, 1 to 15 row convolutions unit and 1 to 5 column convolution units. This choice allows us to compute the weighted sum values in one clock cycle with a highly parallel hardware unit or to compute it in a sequential way. Results using different configurations are shown in section 4.

3.3 Final Velocity Calculation Using a Custom Floating Point Unit

At this stage the expression (1) is computed. Until now, the arithmetic operations have been done using integer or fix point arithmetic with truncation operations. Convolution operations work well with this representation but when the bit depth is too high, a floating point data representation become better suited for hardware implementation. This is done with a customized superscalar floating point unit whose architecture is illustrated in Fig. 4. Since at the previous stage, a high bit-depth (24 bits) is used to preserve the computation accuracy, it is a very expensive stage in terms of hardware resources. Therefore it is a critical stage that highly affects the accuracy vs. processing speed trade-off.

This stage computes the inverse of a matrix and the multiplication of a (2x2) matrix by a (2x1) vector. This calculus involves the basic arithmetic operations: subtraction, multiplication and division. The hardware structure of the unit developed to compute this is shown in Fig. 4.

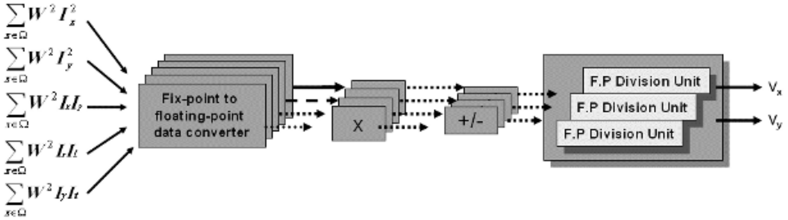

$\sum_{x\in\Omega} W^2 I_x^2$
$\sum_{x\in\Omega} W^2 I_y^2$
$\sum_{x\in\Omega} W^2 I_x I_y$
$\sum_{x\in\Omega} W^2 I_x I_t$
$\sum_{x\in\Omega} W^2 I_y I_t$
Fix-point to floating-point data converter
X
+/-
F.P Division Unit
F.P Division Unit
F.P Division Unit
V_x
V_y

Fig. 4. Floating-point unit scheme

When arithmetic operations are done with large bit depth, the signal delays associated to carry lines degrades the global system performance decreasing the maximum system frequency. To avoid this, pipeline arithmetic operators or sequential iterative operators can be used. The first one allows us to make the computation in few (1 or 2) clock cycles after a given latency at an expensive cost in terms of hardware resources. The second option takes several clock cycles therefore, degrading the MNC of the system, but allows us to use the same hardware for each iteration. We define a system which uses 1 cycle floating point hardware circuits because this works at the desired maximum clock frequency (without becoming the limiting stage) for all the operations but the division, because it is a difficult operation. We have used a hardware sequential divisor instead a pipelined divisor that needs 21 cycles to compute the division of 25 bits floating numbers therefore, the MNC is too high and it highly limits the system pipeline performance. The chosen solution uses up to 3–ways division units and, depending on the system performance required, we can synthesize more or less ways. Each floating-point unit needs: one to five fix-points to floating point converter units; one to six 25 bits floating point multipliers; one to three subtractors; one to two divisor units. If *n*-ways divisor scheme is used, then we use *n* to *2n* divisor units. Results using different configurations are shown in section 4.

4 Hardware Resources Consumption Study

The system has been designed in a very modular way. The parallelism and the bit accuracy at the different stages can be easily modified. Due to the high level of abstraction that Handel-C provides [14] it is easy to manage the parallelism of the computing circuits and the bit-depth at the different stages. In table 1 is summarized the hardware resources of the different stages using a XCV2000E-6 Virtex FPGA for a concrete implementation (called HSHQ in the following section).

Table 1. Detail sub-circuits hardware requirements for a Virtex XCV2000E-6. Note that the sum (% of the device in the first column) is larger than 100%, this can be explained because these data have been obtained by partial compilations and the synthesis tool makes a wide use of the available resources. When the whole design is compiled it consumes 99% of the device.

	Number of slices / (% of the device) / equivalent gates	Computing cycles	ISE maximum Clock frequency (MHz)	Memory requirements / (% of the device)
Spatial Gaussian (17 taps)	220 / (1%) / 270,175	8	29.2	16 / (10%)
IIR filter	134 / (1%) / 51,971	7	38.5	3 / (1%)
Spatial derivative convolution	287 / (1%) / 121,296	7	28.0	7 / (4%)
Least square matrices construction	15,288 / (79%) / 642,705	10	20.3	24 / (15%)
Superscalar floating point unit	5,720 / (29%) / 90,993	10	17.4	0

The last two stages have the larger MNC values. Note that a lower MNC are possible for other stages but there is no reason to improve them due to the other existing limiting stages. The results of the Xilinx timing analyser are not always accurate. In fact, it can underestimate the speed at which a circuit can run, leading to the place and route tool to take much longer than it needs to; i.e. the maximum frequency allowed by the system has been experimentally measured and it is 10-20 MHz higher than the very conservative results given by ISE. This arises because the analyser looks at the static logic path, rather than the dynamic one (see [15]) and because of that we measure experimentally the maximum working frequency.

One important topic is the system configuration possibilities. We have evaluated several configurations to explore different trade-offs between accuracy, hardware cost and computing speed. In all these configurations we have used the same basic architecture but with different parallelism levels. Table 2 summarises the main properties of the different configurations. The ones using a 5x5 average window of the least-square-matrix neighbourhood we call high quality (HQ) approaches, and the ones using a 3x3 window, medium quality (MQ). Other modifiable parameters are the smoothing and spatial derivative filter sizes. HQ and MQ models include 5-pixel derivative filters and 9-pixel Gaussians. A low cost (LQ) version uses 3-pixel derivatives and a Gaussian filter of the same size.

If we fix the optical flow quality of the system, another factor to take into account is the performance vs. hardware cost trade-off. If the system works with maximum parallelism the MNC is 10. Lower cost approaches are possible if we reduce the parallelism level, thus increasing MNC. For example, we implemented a high-speed (HS) version with MNC=10 cycles using a three-way division unit and maximum parallelism. A slower version was implemented reducing the parallelism. We call this version medium speed (MS). Finally, we implemented a low-speed (LS) version. Table 2 summarises the performance of the systems and hardware costs.

Table 2. Performance and hardware cost of different configurations in a Virtex 2000-E FPGA (2 million gates and 640 Kbits of internal memory). (Kpps → kilopixels per second, Fps → frames per second). All the performance values were measured using a system clock frequency of f_{clk}=27MHz. These measurements (Kpps and Fps) are underestimations because the computing time measured also include data transmission to the prototyping board.

Version	% device occupation	% on-chip memory	Kpps	Image resolution	Fps (f_{clk}=27M Hz)	Max. f_{clk} (MHz)
HSHQ	99	17 31	177 6	160x120 320x240	95 24	35
HSMQ	65	16 31	177 6	160x120 320x240	97 24	35
MSMQ	43	16	625	160x120	33	35
LSLQ	36	8	400	120x90	38	35

It is important to note that in our experiments data transmission to the prototyping board using PCI bus takes about 40% of the total processing time, and therefore higher frame rates are expected using a direct connection between the camera and the FPGA. For instance, as explained in section 1, the theoretical bit-through of the HSHQ is 2700Kpps. This topic is widely discussed in [16].

Until now, we have shown the system flexibility and the trade-off between number of gates and system performance. Other important topic is the scalability at the level of functional units. All our results make the assumption that only one computational unit is used. Local image processing algorithm can take advantage of the FPGA splitting possibilities. We can synthesize some computational units in the same FPGA or in several of them and compute larger images in real time. If a memory buffer is used, it is straightforward to assign a small area to each computational unit and run it in parallel. The computational power is then increased by a factor equal to the number of computational units running in parallel. Within the pipeline computing structure, the scalability principles have been used in the floating point unit design where we have implemented a three ways superscalar division unit. This has been done to reduce the number of cycles required by this stage from 21 to 7, therefore obtaining a well balanced pipeline computing architecture.

5 Performance Evaluation

As commented in the introduction, the accuracy of the optic flow computation of real world sequences is difficult to assess because the real flow of these sequences is unknown. Therefore to evaluate the accuracy of our design that depends on the bit-depth of the different stages, we have adopted the test scheme and the synthetic sequence from the comparative study done by Barron et al. [1]. The results using the HSHQ approach are summarized in Table 3.

In the first row of Table 3 is compared the accuracy of the L&K algorithm computed by a standard PC using double precision variables and adopting the IIR filters using the error measure proposed in [17,18]. The second row includes the performance obtained by our hardware implementation. It can be seen that the accuracy is reasonable taking into account that fixed point variables and restricted bit depths are used in this approach.

Table 3. Yosemite sequence results using Fleet angle error measure [17,18].

Model	Average Error	Standard deviation	Density %	Parameters
LK IIR software vs. real flow	15.91 °	11.5 °	100	λ_{min}=0, σ_{xy}=0.8, τ=2, α=1
Hardware implementation vs. real flow	18.30 °	15.8 °	100	λ_{min}=0, σ_{xy}=0.8, τ=2, α=1

We also have compared the performance of the software and the hardware implementations using sinusoidal grating sequences. We used different stimulus frequencies (f_0=0.02 and f_0=0.05) and velocities (V=0.25 ppf and V=1 ppf). With these tests the hardware obtained results very similar to those of the software approach (less than 5% of error in the speed calculation). The software implementation (standard C) runs at a 30 fps of 160x120 pixels on a AMD 1800+.

6 Conclusions and Future Work

The system described here shows how an optical flow estimation circuit can be implemented using an FPGA platform as a specific purpose DSP to achieve real-time computation. The paper describes a scalable architecture that can work with large image data at video-frame rate.

Table 3 summarises the results of a comparison between the software and hardware results using the Yosemite sequence test and unthresholded results. It can be seen that the performance of the hardware is only slightly worse (2.48° increment of error) than the software version with data precision of 64 bits. The results of the hardware implementation described in this paper are in the range of other software approaches considered in the study of Barron et al. [1]. Therefore, the performance of the hardware is of reasonable quality provided that it computes in real time (at a speed of 1776 Kpps).

In the future, we plan address two main goals. The first one is to study the bit-depth needed for different applications. Although this hardware approach is being used with real-world sequences with satisfactory results, it uses a high depth that can be reduced significantly. The second goal is to use a multiscale computation to detect faster motion properly. Classical gradient models estimate velocities well for speeds slower than one pixel per frame but faster motion produces temporal aliasing. The basic solution consists in computing motion at higher frame rates (that needs special and expensive cameras). Alternatively, using multiscale approaches the same cameras can be used.

Acknowledgments. This work has been supported by the V EU research framework funds through the European Projects ECOVISION (IST-2001-32114).

References

[1] J. Barron, D. Fleet, S. Beauchemin: Performance of optical flow techniques. Internat. J. Computer Vision, Vol. 12, nº.1, pp 43-77, 1994.

[2] B. Lucas & T. Kanade: An iterative image registration technique with an applications to stereo vision. Proc DARPA Image Understanding Workshop, pp. 121-130, 1984.

[3] H. Liu , T.H. Hong , M. Herman , T. Camus and R. Chellappa: Accuracy vs. Efficiency Trade-offs in Optical Flow Algorithms. Computer Vision and Image Understanding. Vol. 72 , Issue 3 (Dec) pp. 271 – 286, 1998.

[4] B. McCane, K. Novins, D. Crannitch and B. Galvin: On Benchmarking Optical Flow. Computer Vision and Image Understanding. Vol. 84, pp 126–143, 2001.

[5] S. Baker and I. Matthews: Lucas-Kanade 20 Years On: A Unifying Framework. International Journal of Computer Vision, Vol. 56, nº.3, pp. 221 – 255, March, 2004.

[6] Gamal, A. El, Optical Flow Estimation using High Frame Rate Sequences, *Proceedings of the 2001 International Conference on Image Processing*, Vol. 2, pp 925-928, 2001.

[7] P. Cobos, F. Monasterio: FPGA implementation of the Horn & Shunk Optical Flow Algorithm for Motion Detection in real time Images. Proc of the XIII Design of Circuits and Integrated Systems Conference, pp. 616-621, 1998.

[8] P. Cobos, F. Monasterio: FPGA implementation of Camus correlation Optical flow algorithm for real time images. Proc of Int. Conf. on Vision Interface, pp. 7-9, 2001.

[9] S. Maya-Rueda, M. Arias-Estrada: FPGA Processor for Real-Time Optical Flow Computation. Lecture Notes in Computer Science, Vol. 2778, pp. 1103-1016, 2003.

[10] D. J. Fleet, K. Langley. Recursive filters for optical flow. IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol. 17, N. 1, pp. 61-67, 1995.

[11] E P Simoncelli and E H Adelson and D J Heeger. Probability distributions of optical flow. IEEE Conf on Computer Vision and Pattern Recognition, Mauii, Hawaii. June 1991.

[12] T. Rowekamp, M. Platzner, and L. Peters: Specialized Architectures for Optical Flow Computation: A Performance Comparison of ASIC, DSP, and Multi-DSP. Proc ICSPAT'97, 1997.

[13] www.celoxica.com

[14] Handel-C language referent manual. Celoxica 2003.

[15] Celoxica application note AN 68 v1.1: Timing analysis. Timing Analysis and Optimisation of Handel-C Designs for Xilinx Chips.

[16] D. Benitez, Performance of reconfigurable architectures for image-processing applications. J. of Systems Architecture: the euromicro journal, vol 49 (4-6), pp. 193-210, 2003.

[17] D.J Fleet, and A. D. Jepson: Computation of Component Image Velocity from Local Phase Information, International Journal of Computer Vision, Vol. 5, N.1, pp. 77-104, 1990.

[18] D. J. Fleet, Measurement of Image Velocity. Engineering and Computer Science. Kluwer Academic Publishers, 1992.

Methods and Tools for High-Resolution Imaging

Tim Todman and Wayne Luk

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2AZ
{tjt97,wl}@doc.ic.ac.uk

Abstract. Film and video sequences are increasingly being digitised, allowing image processing operations to be applied to them in the digital domain. For film in particular, images are digitised at the limit of available scanners: each frame may contain 3000 by 2000 pixels, with 16 bits per colour channel. We investigate the consequences of working with these high-resolution images on FPGAs. We consider template matching and related algorithms, and derive a performance model to establish bounds on performance and to predict which optimisations may be fruitful. An architecture generator has been developed which can generate optimised implementations given image resolution, the FPGA platform architecture, and a description of the image processing algorithm.

1 Introduction

Film and video sequences are increasingly being digitised, allowing image processing operations to be applied to them in the digital domain. For film, high resolution digital scanners have been developed, giving 3000 by 2000 pixels per frame, and 16 bits per colour channel. For video, digital formats are increasingly common: DV, mini-DV, High Definition Television(HDTV). Like film, HDTV can require large resolutions – the Society of Motion Picture and Television Engineers (SMPTE) has defined several high-resolution standards.

These high-resolution formats contain much more data than conventional video. Each image in the SMPTE 374M standard (1920 by 1080 pixels, 30 bits per pixel) contains some 7.42MB, compared to 1.17MB for VGA video (640 by 480, 24 bits per pixel). Digitised film frames are even larger: 45.8 MB for 3000 by 2000 images with 64 bits per pixel, and higher-resolution scanners are being developed. Large images are used in medical applications: for example, in chest radiography a 2048 by 2048 format, with 1024 shades of grey (10 bits per pixel) [7] has been used; the Digital Mammography Database contains images up to 5200 by 4000 pixels [2].

In this paper, we consider the use of FPGAs for processing of large-format images. Our target application is template matching, a convolution of two images together which has many applications in its own right, and as a component of larger algorithms.

This paper makes the following contributions:

- We map a generic template matching algorithm to hardware, showing the memory requirements and the effect of different numbers of memory banks on the performance.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 627–636, 2004.

- We create a performance model, illustrating the effects of caching on the performance.
- We propose an architecture generator which takes the basic platform architecture and image sizes and allows the user to create an optimised configuration.

Many researchers have implemented template matching and related algorithms on reconfigurable hardware, as far as we are aware, none have done so for large resolutions.

Gause et al. implement Summation of Absolute Differences (SAD) correlation, compiling to a systolic array implementation [4]; they apply this correlation to template matching at HDTV resolutions. Their implementation resembles our work for the case when all of the template is cached on the FPGA; however they do not consider caching just part of the template on the FPGA. Weinhardt and Luk [8] infer shift registers and memories for loop nests. Diniz and Park [3] also infer small memories to reduce the bandwidth required to off-chip memory, as well as parameterizable external memory interfaces for FPGAs [6]. All of these papers assume that the memories fit into the FPGA along with the rest of the algorithm. We handle cases when these caches do not fit. Our methods complement these prior approaches, and we would like to explore integrating these techniques.

The rest of this paper is organised as follows. The next section describes the template matching algorithms we use, and shows a straightforward mapping to reconfigurable hardware that is used as a base for comparisons with more sophisticated techniques in later sections; we also develop the performance model for this basic mapping. Next, we develop several caching techniques and extend the performance model to suit. The next three sections describe an architecture generator, show various specialisations of the basic algorithm, then give some results for actual hardware implementations. Finally, we conclude and suggest ideas for future work.

2 Mapping to Reconfigurable Hardware

We consider the problems of mapping image processing algorithms to FPGA platforms allowing for such large images. We study template matching, a form of convolution, which is the basis of many image processing functions, for example correlation, edge detection and blurring, and is used as part of other algorithms such as image registration. We consider convolutions between two images, which we refer to as image 1 (I_1) and image 2 (I_2), producing a resulting image Ic. In terms of template matching, I_1 is the template to match in I_2. These images may optionally have associated masks M_1 and M_2 respectively, of the same width and height. We consider convolutions of the following form:

```
for $y = 0$ to $H_2 - H_1$ do
  for $x = 0$ to $W_2 - W_1$ do
    $Ic_{y,x} = f_3($
            $\sum_{i=0}^{H_1-1} \sum_{j=0}^{W_1-1} f_{12}(M1_{i,j}, I1_{i,j}, M2_{y+i,x+j}, I2_{y+i,x+j})$,
            $\sum_{i=0}^{H_1-1} \sum_{j=0}^{W_1-1} f_{11}(M1_{i,j}, I1_{i,j})$,
            $\sum_{i=0}^{H_1-1} \sum_{j=0}^{W_1-1} f_{22}(M2_{y+i,x+j}, I2_{y+i,x+j}))$
    end for
end for
```

where image 1 is of height H_1, width W_1; image 2 is of height H_2, width W_2; $H_2 > H_1$ and $W_2 > W_1$ and f_3, f_{12}, f_{11}, f_{22} are pure functions (that is, they have no internal state and their results depend only on their parameters). The resulting image Ic has height $H_2 - H_1 + 1$ and width $W_2 - W_1 + 1$. Note that image subscripts represent coordinates; thus $I1_{i,j}$ means the pixel at $I1$'s ith row and jth column.

This form covers many kinds of correlations and convolutions, for example:

- the SAD correlation of two images would have no M_2, $f_3(a, b, c) = a$, $f_{12}(m1, i1, m2, i2) = m1 \times \text{abs}(i2 - i1)$ and null f_{11} and f_{22}.
- the normalised correlation between I_1 and I_2 would have $f_3(a, b, c) = a/\sqrt{b \times c}$, $f_{11}(m1, i1) = m1 \times i1$, $f_{22}(m2, i2) = m2 \times i2$ and $f_{12}(m1, i1, m2, i2) = m1 \times i1 \times m2 \times i2$.
- a gaussian blur would have no masks, image 1 as the gaussian kernal and image 2 as the image to be blurred.

This form of the algorithm has the following properties:

- All pixels of the convolved image are independent; they may be calculated in parallel or in any order.
- Likewise, each pair of inner and outer summations can be interchanged.

We use these properties in our caching schemes in the next section.

We choose a pipelined design, allowing us to use optmimised hardware libraries such as multipliers and square root, as needed for different functions f_{11}, etc. We show later that as long as the cycle time of the design is less than the memory access time, the memory access is the limiting factor: the algorithm is limited by speed of input and output. Each memory bank can be written to at most once per cycle. Figure 1 is a block diagram of our hardware's datapath. It shows each image (I_1, I_2, M_1, M_2, I_c) mapped to a separate memory bank. This assumes that all of each image fits in one bank, and that there are at least five external memory banks on the FPGA platform.

For each pixel of I_c, we must read all of I_1 and M_1, and an I_1-sized area of I_2 and M_2. If the data bit-width is d bytes, and the memory wordsize in bytes is s, this takes a total of:

$$(d/s) \times (H_2 - H_1 + 1) \times (W_2 - W_1 + 1) \times H_1 \times W_1$$

cycles.

Table 1 summarises the number of cycles taken by this basic mapping, for film and HDTV image resolutions and 1-, 4- and 8-byte per pixel image formats. Note that the memory word-size $s = 4$ for these results:

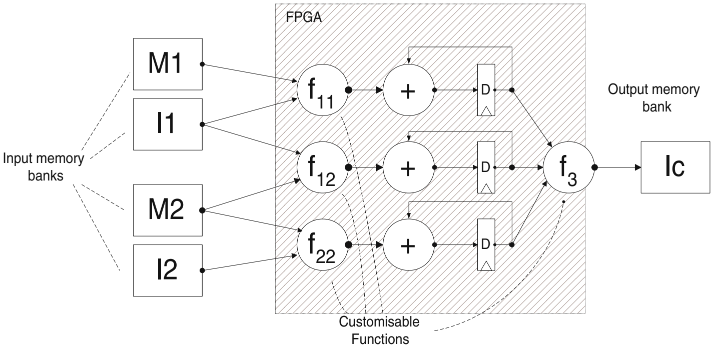


Fig. 1. Datapath of our design. Parts inside hatched area are on FPGA device; outside blocks are external RAM banks. Round blocks are stateless, pure functions; D-type flip-flops represent registers used to accumulate summations. Functions f_{11}, f_{12}, f_{22}, f_3 set what kind of template matching algorithm is performed.

Table 1. Time taken in seconds to run the straightforward mapping on a platform with 100MHz RAM, showing times for film (4000 by 3000) and HDTV(1920 by 1080) resolutions, with two sizes of template.

D	H_2	W_2	H_1	W_1	cycles	time
1	4000	3000	32	32	3.02E+09	30.17
4	4000	3000	32	32	1.21E+10	120.79
8	4000	3000	32	32	2.42E+10	241.81
1	1920	1080	32	32	5.07E+08	5.07
4	1920	1080	32	32	2.03E+09	20.31
8	1920	1080	32	32	4.07E+09	40.66
1	1920	1080	64	64	1.93E+09	19.34
4	1920	1080	64	64	7.74E+09	77.37
8	1920	1080	64	64	1.55E+10	154.79

The first three rows show times for a small image 1, such as a 32x32 gaussian filter, with image 2 at digitised film resolution; the second three rows repeat this with image 2 at HDTV resolution. The last three rows show repeat the second three rows for larger image 2. These results show that the basic scheme may take a long time for large images.

In the next section we develop our performance model to account for caching.

3 Caching

In the previous section, we developed a formula for the number of cycles taken by a straightforward implementation of the algorithm. We now:

- extend our performance model to account for caching.

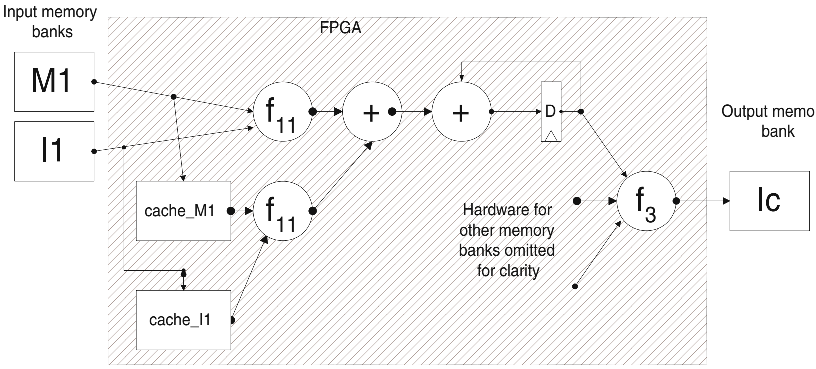


Fig. 2. Datapath of our design, showing caches *cache_M*1 and *cache_I*1 on for the memory banks containing $M1$ and $I1$ respectively; other memories, functions and accumulators removed for clarity. Data read from memory are cached to avoid re-reading. An extra instance of function f_{11} processes cached data; an extra adder sums the data from external memory with the cached data.

– use the performance model to establish a lower bound on the time taken assuming perfect caches of infinite and bounded size.
– use the performance model to calculate the performance of practical cache sizes and configurations.

The straightforward implementation repeatedly reads parts of the images and masks, suggesting that we can save time by caching these parts using internal FPGA RAMs. We extend the performance model to account for two mutually-exclusive caching schemes: perfect caching, where each pixel need only be read once; caching several columns or rows of I_1 and I_2.

Figure 2 shows a simple cached version of the basic datapath in fig. 1. The memories containing masks and images are read as before, but the values they yield are cached in on-chip memories. These caches can then be read simultaneously with the main memory, and save re-reading values that have already been read.

3.1 Perfect Caching

For perfect caching: the algorithm needs to read all of $I1$, $M1$, $I2$ and $M2$. We assume that all the images and masks are read in parallel. As $H_2 > H_1$, and $W_2 > W_1$, the time to read $I2$ will dominate the time to read $I1$. Assuming the result image, Ic can be written at the same time, the overall time is just the time to read image 2, given by:

$$(d/s) \times (H_2 \times W_2)$$

Table 2 shows the time for the examples using this perfect caching:

Table 2. Repeat of table 1, adding times for perfect caching; the final column shows the speedup achieved.

					No Cache		Perfect Cache		
d	H_2	W_2	H_1	W_1	cycles	time	cycles	time	speedup
1	4000	3000	32	32	3.02E+09	30.17	3.00E+06	0.030	1005.81
4	4000	3000	32	32	1.21E+10	120.79	1.20E+07	0.120	1006.55
8	4000	3000	32	32	2.42E+10	241.81	2.40E+07	0.240	1007.53
1	1920	1080	32	32	5.07E+08	5.07	5.18E+05	0.005	978.79
4	1920	1080	32	32	2.03E+09	20.31	2.07E+06	0.021	979.50
8	1920	1080	32	32	4.07E+09	40.66	4.15E+06	0.041	980.46
1	1920	1080	64	64	1.93E+09	19.34	5.18E+05	0.005	3730.73
4	1920	1080	64	64	7.74E+09	77.37	2.07E+06	0.021	3731.42
8	1920	1080	64	64	1.55E+10	154.79	4.15E+06	0.041	3732.33

Comparing the two tables, we note that: using perfect caching, only the size of I_2 bounds the total processing time; for our examples, three orders of magnitude speedup can be achieved using perfect caching, owing to the application being IO-bound.

We do not need to cache all of $I2$ to implement a perfect cache, because not all of $I2$ is required to produce any pixel of Ic; we use the scheme of Gause [4]. Figure 3 shows the minimum perfect cache size. Starting at the bottom left of Ic, we cache pixels of $I2$ and $M2$ in row-major order, simultaneously caching pixels of $I1$. When we reach pixel $I2_{H_1-1,W_1-1}$, we can produce $Ic_{0,0}$. We continue scanning, each read of $I2_{y,x}$ producing $Ic_{y-H_1+1,x-W_1+1}$. The shaded area of fig. 3 shows the area of $I2$ and $M2$ we need the cache to be $(H_1 \times W_1) + (H_1 - 1) \times (W_2 - W_1) = (H_1 \times W_2) - W_2 + W_1$ pixels in size. This is just for $I2$; we must also store the same area of $M2$ and all of $I1$ and $M1$. Because all the images are stored on the memory banks, and the outer loops of the algorithm are interchangeable, perfect caching can also run in column-major order; by analogy to the above formula for row-major order, the space required for column-major order is $(W_1 \times H_2) - H_2 + H_1$ pixels – column-major order uses less space for square $I1$ with any $I2$ where $H_2 < W_2$, which is true of many video formats.

Perfect caching is not practical for large images as it uses much on-chip memory: for HDTV resolution (I_2 of size 1920 by 1080), with I_1 of size 32 by 32, a perfect cache needs $32 \times 1080 - 1080 + 32 = 33512$ pixels. A large FPGA, such as the Xilinx XC2V8000 can cache that many pixels in an eight-bytes-per-pixel image format in its dedicated RAM resources (block SelectRAM), but cannot cache an image 1 more than 45 pixels tall. Moreover, to generate a pixel of Ic every cycle, we must do the entire summations of the algorithm in that cycle. This requires $H_1 \times W_1$ additions and the same number of instances of f_{11}, f_{12} and f_{22}.

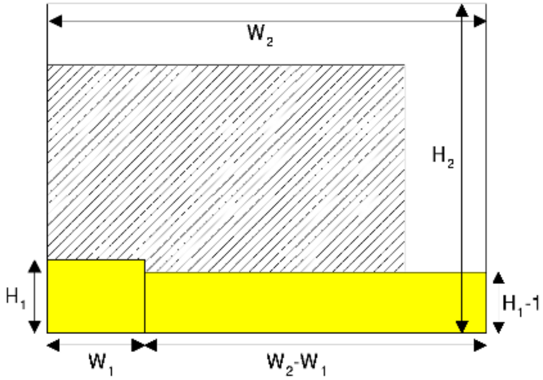


Fig. 3. Size of cache for perfect caching, shown by shaded area. Ic shown as hatched box (extends under shaded area). Outer bounding box is the size of $I2$.

3.2 Practical Caching

If a perfect cache is impractical, some speedup can still be obtained from using smaller caches. We investigate the effect of caching one or more H_1-height columns of $I1$, $M1$, $I2$ and $M2$.

Consider the effect of caching N such columns. Once the cache is initialised with the first N columns, only $W_1 - N$ columns of $I1$, $M1$, $I2$ and $M2$ need be read to produce the next pixel of Ic, assuming that the cache is read in parallel with being updated. Since only $W_1 - N$ columns need be read per pixel, the total number of reads per row of Ic is $H_1 * W_1$ for the first pixel, plus $H_1 \times (W_1 - N)$ for each of the remaining $W_2 - W_1$ pixels. The total number of cycles taken is thus:

$$(d/s) \times (H_2 - H_1 + 1) \times ((H_1 \times W_1) + (W_2 - W_1) \times (H_1 \times (W_1 - N)))$$

Compared with the case without caching, a cache with $N = W_1/2$ will make half as many reads, hence it will run twice as fast. N can be usefully increased until $N = W_1 - 1$; beyond that point the extra cache does not contribute to the calculations for the current pixel. Substituting $N = W_1 - 1$ into the expression above shows that the speedup in this case is roughly W_1.

Like the perfect cache mentioned above, an extra instance of f_{11}, f_{12} and f_{22} are required for each cache read that can occur in parallel with the memory read. Thus for $0 < N <= W_1/2$, one extra instance of each are required, for $W_1 + 1 < N <= 2W_1/3$, two extra instances are required, and so on.

4 Specialisations

So far, we have considered the implementation of the general case of our template matching algorithm. The general case allows one or more of the masks, and all but one of the customisation functions f_{11}, f_{12}, f_{22} and f_3 to be absent. In this section, we consider specialisations of the general purpose algorithm, and how these may yield further opportunities for optimisation. We list several specialisations, then explore one in more detail:

- Image $I1$ may be constant for a number of different image $I2$s.
- If one or both masks are absent, the memory banks that would have been used can be saved.
- Mask $M1$ always multiplied with $I1$: the host application can premultiply the mask and image, saving a memory bank.
- Small bitwidth for masks: some applications use boolean mask types; in this case, the entire mask can be cached on the FPGA as a preprocessing step, saving a memory bank.
- Small $I1$: recognising a small input image would allow us to use some of the existing methods such as [8].
- Sharing memory banks between images: sometimes, Ic and $I1$ will fit in the same memory bank. Ic is only written every $H_1 \times W_1$ cycles in the straightforward algorithm.
- Use of saturating arithmetic: some applications can tolerate the use of saturating arithmetic, which clamps a value to a maximum level if calculations exceed that level.
- Compression of input images: this can also be done by the host, perhaps allowing a modest increase in the size of image that can fit in one bank.

We deal with a constant $I1$ in more detail. This situation can happen in several ways: $I1$ is being used as a filter, for example a Gaussian blur; one template is being matched to many other images – for example in factory automation or security applications.

If $I1$ is constant, the hardware configuration can take advantage of this. Instead of caching the image values in registers, they can be encoded in constant multipliers and adders, taking much less room than the caches and variable multipliers used in our caching designs, so more of the image can be cached on the FPGA. This runtime constant propagation could be built into the architecture generator we describe in the next section.

5 Architecture Generator

In this section we propose an architecture generator that, given image sizes, bit widths and number of memory banks, generates several cache configurations, reports the number of cycles taken by each, splitting images across banks if enough memory banks are available.

The proposed architecture generator produces designs in Celoxica's DK [1] tool which instantiate Xilinx Coregen blocks. It generates designs using a specified pipeline length. It is up to the user to compile the resulting circuits and verify that they fit on the available hardware, and that the clock cycle time exceeds that of the memory. Currently, the generator does not support automatic

Table 3. Device utilisation for various column caches for Sum of Absolute Differences correlation, H_1 = 16, 32, 64, using Xilinx XC2V4000 device.

	H_1 =16		H_1 =32		H_1 =64	
Number of columns	% FF	% LUT	% FF	% LUT	% FF	% LUT
(no cache)	1	8	1	8	1	8
1	3	13	7	16	13	22
2	7	18	13	24	27	35
3	10	23	20	31		
4	13	28	27	39		

tradeoffs between pipeline length and available cache size – we also require the user to do this manually.

We adopt an architecture generator approach, because although languages like Handel-C offer powerful macros, allowing programs to be parameterised for bit widths and number of memory banks, such parameterised programs are not always easy to use.

An architecture generator can:

- handle quirks like pipelined RAM, priority queues to write to memory.
- validate input parameters ($H_2 > H_1$, $W_2 > W_1$, $d = 1, 2, 4, 8$).
- store architectural descriptions (currently: number of memory banks, pipelined memories).
- generate combined mask / image banks (concatenated, or multiplied together if the algorithm allows it).
- generate multiple cache configurations.

The architecture generator takes as input the parameters of the template matching algorithm: H_2, H_1, W_2, W_1, d, s, list of memory banks and the custom functions f_{11}, f_{12}, f_{22} and f_3. As output it produces several designs: a straightforward design, to use as a base for comparisons, various column caches and a perfect caching design, along with calculations of the number of clock cycles taken. Users can then synthesise these designs and choose which best suit their needs.

6 Results

Figure 5 shows utilisation of Xilinx Virtex XC2V4000 for 0- to 4-column caches for $H_1 = W_1 = 32$, SAD correlation.

As expected from section 3.2, the utilisation rises fairly linearly with the number of columns in the cache.

7 Conclusion

In this paper, we mapped a customisable template matching algorithm to hardware, focussing particularly on large image resolutions and bitwidths, which do not allow the entire template image to be cached on the FPGA. We developed a performance model, allowing us to set an upper bound on the performance available from caching, and to analyse the performance available from more practical caching designs. Finally, we proposed an architecture generator, to take the parameters of a template matching algorithm and generate various caching designs, to allow the user to explore the design space.

We would like to explore several possible future extensions to this work. Firstly, the ideas in this paper could be extended to three-dimensional image processing. Our generic algorithm could be extended to three dimensions fairly easily, but designing suitable caches is likely to be more challenging [5]. Secondly, we would like to combine our approach with the related work mentioned earlier. Thirdly, we would like to use our methods to implement the Fast Fourier Transform (FFT) method for template matching. The FFT has a lower computational complexity than our convolution method, but has a butterfly memory access pattern, making caching rather different. Finally, we would like to explore runtime constant propagation, as mentioned in the section on specialisation.

References

1. Celoxica DK1 website, `http://www.celoxica.com/products/tools/dk.asp`
2. Digital Mammography Database website, `www.wiau.man.ac.uk/services/MIAS/MIASfaq.html`
3. Diniz P and Park J, "Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Engines", in *Proceedings FCCM*, IEEE, 2000.
4. Gause J, Cheung PYK and Luk W, Reconfigurable Shape-Adaptive Template Matching Architectures, in *Proceedings FCCM*, IEEE, 2002.
5. Nikolaidis N, Pitas I, *3-D Image Processing Algorithms* John Wiley, 2001.
6. Park J and Diniz P, "An External Memory Interface for FPGA-Based Computing Engines", in *Proceedings FCCM*, IEEE, 2001.
7. Kurt Rossman Laboratories web site, `www-radiology.uchicago.edu/krl/imgquali.htm`
8. Weinhardt M and Luk W, "Memory Access Optimisation for Reconfigurable Systems", *IEE Proc. Comput. Digit. Tech.*, IEE, Vol 148, No. 3, May 2001.

Network-on-Chip for Reconfigurable Systems: From High-Level Design Down to Implementation*

T.A. Bartic[1], D. Desmet[1], J-Y. Mignolet[1], T. Marescaux[1], D. Verkest[1**], S. Vernalde[1], R. Lauwereins[1***], J. Miller[2], and F. Robert[2]

[1] IMEC, Kapeldreef 75, 3001 Leuven, Belgium
{bartic,desmetd,mignolet}@imec.be
[2] Université Libre de Bruxelles

Abstract. In order to use Networks-on-Chip as communication infrastructure for heterogeneous, reconfigurable Systems-on-Chip, a set of tools are needed that would allow for an evaluation of the performance of a particular network, and a fast implementation of the system. In this paper we present two models that can be used in the design and implementation of the platform and of its applications. The first model is written in synthesisable VHDL, and it is highly parameterizable allowing a fast network implementation. The second one is a cycle-accurate SystemC model that allows a fast exploration of the design space. The models offer complementary information and help the platform and the application designers to make the best trade-offs. We present how the two models can be used for platform optimization and implementation and for application mapping, using a motion JPEG decoder as a case study. We analyze the system performance as a function of the different design parameters and we present the implementation results for the reconfigurable platform that we have built.

1 Introduction

As the complexity of Systems-on-Chip (SoCs) continues to rise, design reuse is more and more regarded as the only way that allows designers to keep pace with the technological developments[1]. In this context, the design of the infrastructure insuring the communication between the different system components (microprocessors, memories, dedicated hardware blocks, reconfigurable hardware) becomes a key element in system design. For platforms containing reconfigurable blocks, fixed communication resources and standard interfaces are the only way to avoid difficult run-time routing problems. Networks-on-Chip (NoCs) are gaining support as the system-wide communication resource, connecting together the major subcomponents of the system (further referred as IPs)[2,3,4]. The reason is that due to their parallel structure, networks can sustain higher communication bandwidths and have much better scaling properties than traditional solutions like buses.

[*] Part of this research has been funded by the European Commission through the IST-AMDREL project (IST-2001-34379) and by Xilinx Labs, Xilinx Inc. R&D group.

[**] also Professor at Vrije Universiteit Brussel, also Professor at Katholieke Universiteit Leuven

[***] also Professor at Katholieke Universiteit Leuven

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 637–647, 2004.

Platform designers have to dimension the communication according to the application domain for which the platform is developed. Besides the application domain, the designer has to consider also the size, the cost and the power requirements while minimizing the development time. Consequently, the network capacity, latency and services have to be dimensioned considering all these constraints. On his side the designer can play with a large number of parameters in designing the network. NoC designs have been proposed by Guerrier et al. [5] with the SPIN network, Rijpkema et al. with Aethereal [6], Kumar et al. [7]. Although all these designs are scalable, the networks size can be increased only by increasing the number of nodes while keeping a fixed, regular topology. We believe however that only one topology can not suit all applications. Moreover, support for irregular topologies is important as they allow to make better trade-offs and permit a smooth transition from the present, mainly bus-based designs. Recently, Saastamoinen et al. [8] proposed a network design that supports several parameterization options including topology. Their approach is to build a library of components that can be combined in order to realize the different networks. However, for the time being arbitrary topologies are not supported.

Application designers need to find the best mapping of a given application onto the platform. In designing programmable or reconfigurable platforms, like the one presented in [9], the estimation can be based on the needs of typical applications, but it can not cover all possibilities. Therefore, the designer will need a high-level model of the network to map the application onto the reconfigurable platform and to assess its performance. The usual design methodology consists in simulating the application, determining the communication needs of the different application blocks, separately simulating the network with a network simulator and tuning the network such that it satisfies the application requirements. There are many general network simulators available, for instance OMNet++ [10] and NS-2 [11], both written in C++. We used OMNet++[12] as the starting point for our network model. Sun et al. [13] have recently used NS-2 for Network-on-Chip simulation, showing how high-level network models can be used to tune the different network parameters. However, a network model that could be simulated together with the application blocks would allow a much faster, more accurate and convenient exploration of the different mapping solutions.

We can therefore identify three different design problems. The design of the platform for a specific class of applications, the mapping of a certain application to a given platform or a mixed platform-application design where the platform is optimized for a certain application. Thus, we have developed two NoC models, a high-level one and a synthesisable one, in order to support all possible scenarios and to realize a short development cycle. This paper describes and compares the two models, discusses the simulation times, and presents a case study illustrating how the two models can be used in a mixed platform and application design.

This paper is organized as follows: Section 2 presents the VHDL network model; Section 3 explains the SystemC network model; Section 4 compares the two models; Section 5 presents the mapping of a motion JPEG decoder on a reconfigurable tile architecture interconnected through our Network-on-Chip, discusses the impact of the different network design decisions on the total system performance and presents some implementation results; finally in Section 6 some conclusions are drawn.

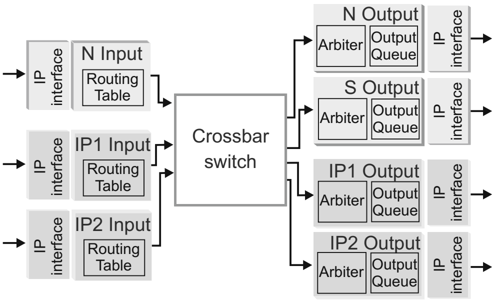


Fig. 1. VHDL network router structure. The number of input ports can be different from the number of output ports, and multiple IPs can be connected to the same router.

2 Network-on-Chip – The VHDL Model

As argued in the introduction, different networks will be required for different application domains. Once the network parameters are determined through simulation, a flexible, easy to customize VHDL network model provides a fast implementation. If the parameters have to be slightly changed to support more applications, the implementation can be also quickly adapted. Our Network-on-Chip, further referred as ICN, was designed to be a very flexible packet-switched network, with routers allowing a variable number of input and output ports, which in turn allows networks of varioustopologies and capacities to be easily built [14]. The network router structure is shown in Figure 1.

The network is implemented as a synthesisable VHDL component for which the following parameters can be specified:

- **Network Topology** – It is specified as a graph of routers, IPs and links. This notation allows us to easily build networks with irregular topologies, with a variable number of nodes and variable number of IPs connected to the same node.
- **Routing algorithm** – It is specified as a routing table and can be customized at link level.
- **Link Width** – It determines how many bits are sent per clock cycle, and therefore it has a strong impact on the network throughput.
- **Output Queue Size** – The ICN uses virtual cut-through switching, therefore when the packets are blocked they are locally buffered in the router. It also uses output queuing, meaning that the buffers are placed in the output, after the switching element. The larger the queue the higher the throughput, but also the total router area.
- **Maximum Packet Size** – The size of a packet is not fixed but it has to be limited, such that the output queue can be dimensioned.

3 Network-on-Chip – The SystemC Model

At system level, the designer will have to evaluate the performance of a given communication architecture in the early stages of the design cycle. If the communication between

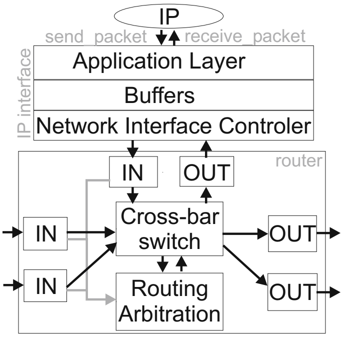


Fig. 2. SystemC network node structure

the IPs is provided by a network, many parameters can influence the overall system performance, both network related (topology, routing algorithm, buffer sizes . . .), as well as application mapping related (number of logical communication channels mapped on the same physical link). Therefore, we have developed a high-level model of the network where these parameters can be easily changed, and that can be co-simulated with the IP models, allowing fast design space exploration.

The high-level network model was written in SystemC, which is an open environment that can be linked to other simulation environments. The network model is pin and cycle accurate although the internal structure differs from the real implementation. The node structure is shown in Figure 2. The IPs connect through an IP interface. The application layer of the interface offers high-level services to the application like *send packet* and *receive packet*. It also makes the adaptation between the simulation language of the IP model and SystemC. For instance the application used in Section 5 is written in OCAPI-xl [15] and required a special interface to make the co-simulation with the SystemC network model possible. The interface contains also buffers and connects to the router through a Network Interface Component (NIC) that implements the same low-level handshaking protocol as the VHDL model.

The router consists of a common routing and arbitration block, a crossbar switch, input and output controllers. The model parameters are:

- **Network Topology** – The topology is defined by a graph of all routers, IPs and links. Variable network sizes as well as irregular topologies are fully supported.
- **Routing algorithm** – Implemented as a look-up table at router level.
- **Switching technique** – One can chose virtual cut through or wormhole switching.
- **Maximum Packet Size** – In the case of virtual cut through switching the output buffers have to be able to hold at least one maximum sized packet.
- **Input Queue Size and Output Queue Size** – The SystemC model allows to specify not only the parameters for the output queue but also for the input one. To emulate the VHDL model the size of the input port buffers is set to zero.
- **Link Handshaking Number of Cycles** – This is an extra parameter, specific to the SystemC model that determines the delay in clock cycles between the request and the acknowledgment signals implementing the handshaking protocol. It allows to make the SystemC model cycle level equivalent with the hardware implementation.

The SystemC model generates two outputs: a signal waveforms file and a packet trace file. Since the model is cycle-accurate, the waveforms can be compared to outputs of the VHDL simulation, and they should be identical. The trace file records for every packet the path it followed through the network, and the timestamps when the packet left each node.

4 Models Equivalence and Simulation Performance

As the two models are not structurally identical we have designed a testbench to checks the equivalence in terms of functionality and the accuracy of the VHDL and SystemC models. The test scenario consists of three parts that grow incrementally in complexity. The first part checks the basic functionality and the flow control when the packets are never blocked in the network. The second part checks the flow control in the blocking case, but only one packet is buffered. The third part is the most complex, it checks the behavior when multiple packets are blocked, buffers are full and other packets wait for the same output. The scenario is repeated with different packet lengths and for different relative timings between the packets.

For this testbench we chose to instantiate a 3x3 mesh network with one IP per node, XY routing algorithm, 16 bits wide bi-directional links, 544 bytes maximum length packets and output buffers that can hold up to 3 packets. The testbench runs for about 400 kcycles. The VHDL model simulated with ModelSim SE 5.7 reaches about 110 cycles/s on a Pentium III machine. The SystemC model reaches 33 kcycles/s, a 300 times speed up compared to VHDL.

We have estimated also the simulation speed in the case when the IPs are blocks of a motion JPEG decoder, which is a much more complex and therefore a more realistic design. The decoder was modeled in OCAPI-xl [15], from which we generated VHDL. The decoder was divided into four blocks, and the communication between them was implemented using the ICN. In this way we have created two simulation models, a VHDL one combining the ICN and the decoder blocks and a similar SystemC – OCAPI-xl model that is explained in more detail in Section 5. The simulation speed of the motion JPEG decoder alone (OCAPI-xl only) was 6.3 kcycles/s. The SystemC – OCAPI-xl co-simulation reached 5.4 kcycles/s. The simulation of the VHDL model was timed at 18.5 cycles/s, again 300 times slower than the high level model. It can be seen that at equal complexity the high-level model performs at least 2 orders of magnitude better, which makes it more suitable for system level design.

5 Motion JPEG Decoder Mapping onto a Reconfigurable Platform

This section goes through all the steps of designing and implementing a tile based reconfigurable platform interconnected by a Network-on-Chip, and mapping an application on it. The purpose is to show what kind of information can be obtained from the simulation model and how simulation can be used to provide information about the network performance for different network parameters. As the application has only an illustrative purpose we have selected a simple, easy to understand motion JPEG decoder modeled in OCAPI-xl [15]. We have decided to partition the application in four tiles: a send tile that reads an input file and streams it to a second tile that performs the Huffman decoding and dequantization (Huffman tile). The third tile implements the IDCT and the rest of the data processing (IDCT tile) and sends its data to the last tile, which writes it to an output file. We have chosen to split the application in to four tiles because it is a number that is already high enough to have a realistic traffic volume but yet not too high, so that the results can be easily understood.

5.1 Platform Design and Performance Appraisal

The goal of the platform designer is to choose an interconnect network that offers the best compromise between performance and size for a certain application domain. The application designer has to find out the best mapping of the application blocks to the platform, such that the application reaches the required performance and makes best use of the available resources. For a reconfigurable platform such as *Gecko*^{2}, the communication requirements change in a dynamic way and depend on what applications are running on the platform at a given moment. Simulation is thus essential, in choosing the right ICN and in evaluating the performance of each application in the context of the whole system. In this section we will do the exercise only for one application, but in general the designers will have to repeat it ideally for all applications that will run on the platform, considering also their interaction when they are run concurrently.

The ICN SystemC model provides information over all the packets that travel through the network and allows us to determine the total travel time of a packet from the sender to the receiver. As an example, Figure 3 plots the end-to-end delay of each packet sent during the decoding of a 5 SIF frames sequence, grouped per logical communication channel. For this simulation, the motion JPEG decoder tiles have been mapped as 4 IPs, each connected to one router of a 2x2 mesh network. The IPs are mapped in such a way that the packets have to travel only one link to reach destination and each link is dedicated to one communication channel only.

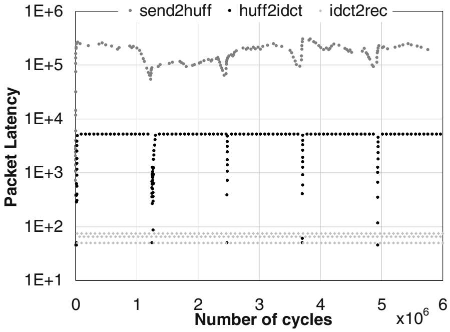


Fig. 3. Packet transmission delay, per communication channel, for a 5 SIF frames sequence

The data rate on the different links is very different. Between the sender and the Huffman tile there are a lot less packets than between the Huffman and the IDCT tile. Also the execution speeds of the different tiles are quite different. The packets from the sender and the Huffman tiles are blocked and have very long delivery delays, while the packets from the IDCT tile are almost immediately consumed. The frame based periodicity of the decoding algorithm is also clearly visible. It can also be noticed that for the second frame the packets sent by the Huffman tile have very small delays compared to the packets of the same channel but for the other frames. It turns out that this frame is very rich in details, which causes the Huffman tile to run a lot slower, at a speed comparable to the IDCT tile, and therefore the packets are not blocked any more but are immediately processed. From this experiment we have concluded that our application runs much slower than the network and therefore the effect of network parameter variations will be hidden by the long execution time of the IPs. We have therefore decided to use different clocks for the network and for the motion JPEG decoder. In all the experiments that follow the IPs have been clocked 5 times faster than the network.

System performance – Network topology. There are multiple network design parameters that can influence performance. We will evaluate the impact of most of them for our motion JPEG decoder. First and the most obvious one is the network topology. The choice in topology has a direct impact over the network area as well. Considering that our decoder has four tiles there are several obvious choices shown in Figure 4. We will consider four different topologies: a 2x2 mesh, 4x1 torus, balanced binary tree and a 4x4 crossbar. The four tiles are considered as four IPs connected by bi-directional links, one for each router. Considering the particular nature of our application we could consider also 3 more topologies: a 3x1 mesh, 3x1 torus and a 3x3 crossbar switch. In this case we take advantage of the fact that the first tile only sends data and the last one only receives data and therefore they can share the same bi-directional connection.

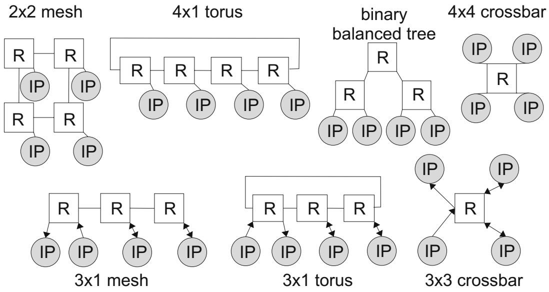


Fig. 4. The network topologies considered for the MJPEG decoder.

The goal of the simulation is to make sure that the network does not constitute a communication bottleneck and to assess the performance loss. The different topologies have been simulated using wormhole switching routers with 2 flits output buffers and packets of maximum 64 flits. The impact of the different network topologies on the system performance is shown in Figure 5. The number of frames per second is calculated assuming that the design runs at the same frequency as the design without the ICN clocked to reach 30 frames/s. In this way the loss in performance due to the network can be rapidly assessed. When the application runs on our platform, the total number of simulation cycles is larger than in the case of the monolithic implementation, therefore for the same clock frequency the decoding speed is lower.

This experiment tells us two things. First, the penalty for the network is not negligible. However, as it will be shown further, it is the wormhole switching that is responsible for the slow down, because similar networks using virtual cut through switching achieve much better performances. The reason is twofold, there are many extra cycles lost in handshaking when packets are blocked and the buffers are smaller. It turns out that similar performance degradation is observed also in the decoder implementation without the ICN if the size of the internal buffers is reduced.

Secondly, there is not much difference between the different topologies. This result was to be expected for such a simple scenario. What it is rather surprising is that the tree network provides the fastest solution while the crossbar, the slowest. One would have expected rather the opposite considering the complexity of the two. It seems however that the amount of available buffer space is more important than the tile proximity, as will be confirmed by further experiments. In the tree network the channel from the Huffman to the IDCT tile passes through three routers and an equal number of output buffers. It can also be noticed that the torus and mesh topologies give identical results, as expected, and that there is no difference between the cases when the first and the last tile are connected to the same router or to different ones. This result is particularly interesting when comparing the 2x2 mesh with the 3x1 one. Although in the last case the packets have to travel back over two links there is no performance penalty. These results are explained by the absence of conflicts in the network and by the pipelined nature of our application.

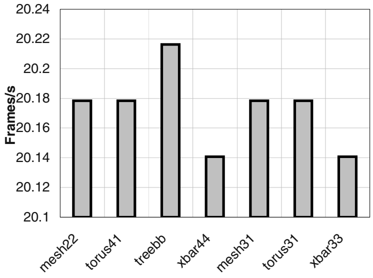


Fig. 5. Impact of the network topology on system performance relative to the design without ICN, decoding 30 frames/s.

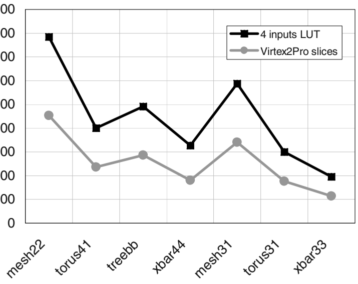


Fig. 6. The size of the different network topologies implemented on a Virtex2Pro FPGA.

Since we can conclude that for this application the topology has little influence on performance, we can start analyzing the area taken by the different implementations. To this end we have used the VHDL model, and we have synthesized, placed and routed the different networks onto a Virtex2Pro FPGA. The results are shown in Figure 6. For the case of wormhole switch routers the size of output buffers can be kept to a minimum of

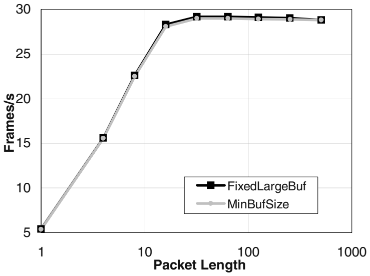


Fig. 7. Impact of the maximum packet length and output buffer size on system performance.

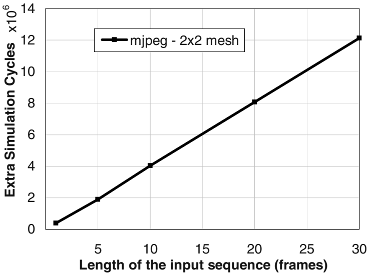


Fig. 8. Extra simulation cycles vs the length of the input sequence.

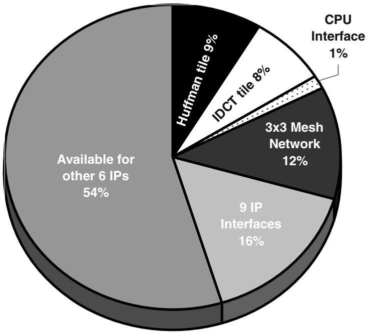


Fig. 9. Relative size, in slices, of the different components of the *Gecko*^{2} platform, implemented on a Virtex II 6000 FPGA.

two flits. For virtual cut through switching their size will depend on the maximum packet size and will have a considerable contribution to the total router size. The buffer size has to be considered on top of the numbers shown in the figure. The figure shows that mesh networks have the largest area, followed by the tree the torus while the crossbar is the smallest.

System performance – Maximum packet length and buffer sizes. As suggested by the topology experiments the size of the output buffers seems to have a large influence on the performance. Another related parameter is the maximum size of a packet.

The results of the experiments investigating the influence of these two parameters are presented in Figure 7. The network topology was kept fixed, a 2x2 mesh with virtual cut through routers, and the results show the decoding speed that would be reached if running the design at the same frequency as the implementation without ICN decoding 30 frames/s. The maximum packet size was varied from 1 to 512. In the first case packets have only one flit of data and three header flits and performance penalty is very large as expected. The performance improves dramatically as the packet size increases, and the decoder reaches almost the same speed as the decoder without the ICN. For very large packet sizes the trend changes, this behavior being explained by the fact that there is a balance between the amount of data transferred in one packet, and thus the packet overhead on one hand, and the packetization delay on the other hand.

The size of the output buffers plays a role as well. Bigger buffers help but the differences are not spectacular. Figure 7 shows two sets of data both taken in the same conditions except that we used minimum size buffers for the first one and fixed size buffers, 512 flits large, for the second experiment. The relative advantage of larger buffers diminishes therefore for larger packets, such that for packets of 512 flits there is only one experimental point. This is a more realistic scenario than having buffers that are a multiple of the maximum packet size, as for very large packet sizes the amount of required resources would grow prohibitively large. For this application and in the absence of traffic conflicts, there is however very little advantage to using large buffers, and the optimum packet length is situated around 32 flits.

System performance – Length of the input sequence. All the above mentioned experiments have been carried out with a fixed length input sequence. Therefore one could ask how much of the speed decrease is latency and how much is start delay. To this end we have simulated with sequences of different lengths, for a 2x2 mesh network with wormhole switching routers, and the results are shown in Figure 8. The figure shows the extra simulation cycles required by the simulation to complete for the ICN implementation compared to the non-ICN one as a function of the length of the input sequence in frames. The results show that the performance loss is for the most part latency and not so much start delay.

5.2 System Implementation Results

This section presents the implementation results of the motion JPEG decoder on our *Gecko*^{2} [9] reconfigurable platform. The platform consists of a central CPU and a Virtex II 6000 FPGA. Linux operating system runs on the CPU and manages the entire platform. The current implementation has nine tiles, out of which eight can be used by user IPs. One tile is always used by the interface with the central CPU. The communication is insured by a 3x3 mesh network. Figure 9 shows the implementation results for the case when two IP tiles have been used to map the Huffman and the IDCT blocks of the motion JPEG decoder. For the decoder, the CPU interface plays the role of the sender and of the receiver. The figure shows the relative sizes, in slices, of the different components of the *Gecko*^{2} platform. The VHDL of the decoder was automatically generated from the OCAPI-xl model. The network and the nine IP interfaces take about 28% of the total amount of slices. The two tiles of the decoder take 9% and respectively 8%. About 54% of the FPGA remains available for implementing six other IPs. Although it might seem that the communication infrastructure uses a lot of hardware resources, it offers a very high communication bandwidth and management services for a relatively complex system that can consist of up to eight IPs.

6 Conclusions

Designing a Network-on-Chip that meets the requirements of a complex, heterogeneous, reconfigurable system is a challenging task. It is therefore important to have appropriate simulation models that can assist the designer through all the different design phases. As the trade-offs that have to be made during the process are very complex, the models need to be as flexible as possible. This paper has showed that our NoC models allow the designer to do fast and accurate simulations while exploring the system design, and generate a corresponding implementation in a short time. Applying the whole design flow to a motion JPEG decoder, we have found that for this application the parameters with the highest impact on performance were the maximum packet length and the switching technique. The implementation results on a Virtex II FPGA show that our network design can provide the required communication resources for a large heterogeneous, reconfigurable system at an acceptable cost.

References

1. J. A. Rowson and A. Sangiovanni-Vincentelli. Proc. DAC, pp. 178–183. ACM Press, 1997.
2. W. J. Dally and B. Towles. Proc. DAC, pp. 684–689. ACM Press, 2001.
3. A. Jantsch and H. Tenhunen. Networks on chip, pp. 3–18. Kluwer Academic, 2003.
4. L. Benini and G. De Micheli. Comp. 35(1):70–78, 2002.
5. P. Guerrier and A. Greiner. Proc. DATE conf., pp. 250–256. ACM Press, 2000.
6. E. Rijpkema, K. G.W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P.Wielage, and E. Waterlander. Proc. DATE conf., 2003.
7. S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. berg, K. Tiensyrj, and A. Hemani. Proc. of IEEE Comp. Soc. Symp. on VLSI, 2002.
8. I. Saastamoinen, D. Siguenza, and J. Nurmi. Networks on chip, pp. 193–213. Kluwer Academic, 2003.
9. V. Nollet, T. Marescaux, D. Verkest, J. Mignolet, and S. Vernalde. Proc. DAC, 2004.
10. http://www.omnetpp.org/.
11. http://www.isi.edu/nsnam/ns/.
12. A. Vargas. European Simulation Multiconference, 2001.
13. S. K. Yi-Ran Sun and A. Jantsch. Proc. IEEE Norchip Conf., 2002.
14. T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Proceedings Systems on Chip Conference, 2003.
15. http://www.imec.be/ocapi.

A Reconfigurable Recurrent Bitonic Sorting Network for Concurrently Accessible Data

Christophe Layer and Hans-Jörg Pfleiderer

University of Ulm,
Department of Microelectronics,
Albert-Einstein-Allee 43,
89081 Ulm, Germany
{christophe.layer, hans-joerg.pfleiderer}@e-technik.uni-ulm.de

Abstract. This paper presents the hardware realization of a recurrent scalable sorting network based on Batcher's bitonic algorithm, which is very suitable for concurrently accessible data. Firstly, preserving the time complexity of the original bitonic sorter, the recurrent network yields a lower area complexity by reducing the communication within the network and minimizes the cost in terms of hardware resources. Secondly, an enhancement of the input registers allows the reuse of the same architecture for different input widths, where the role of each comparator level is redistributed over the network. Finally, the implementation of such a sorter has been realized in an FPGA (Field Programmable Gate Array) and shows how applications treating data block-wise can benefit from this architecture.

1 Introduction

Sorting is one of the most common task performed by a computer in both parallel and sequential computing systems. One of the best known sorting network is Batcher's bitonic merger and sorter [3], discovered in 1968. This sorting network has the capability of sorting N keys in $O(\log^2 N)$ time with $O(N \cdot \log^2 N)$ comparators. Though the bitonic sorting network has a very regular architecture, it is not optimal in terms of area and depth (or in other words delay time). Some of the fastest sorting network known, also dealing with various input widths, are described by Knuth in [5]. Ajtai et al. have presented in [2] an algorithm working in $O(\log N)$ delay time for a sorting network processing N keys with a complexity in $O(N \cdot \log N)$. Furthermore, the algorithm of Leighton presented in [8] for sorting N keys in $O(\log N)$ time with N processors is not appropriate for hardware realizations and is much slower in practice than other parallel sorting algorithms. None the less, Batcher's bitonic sorter remains the most suitable for actual implementations in hardware or in multi-processor systems.

Following these observations, section 2 reminds the principles and gives an overview of a regular bitonic sorter and merger. Section 3 presents the recurrent bitonic sorting network including a direct feedback to the input registers. Section 4 shows how the recurrent sorting network can be transformed into a

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 648–657, 2004.

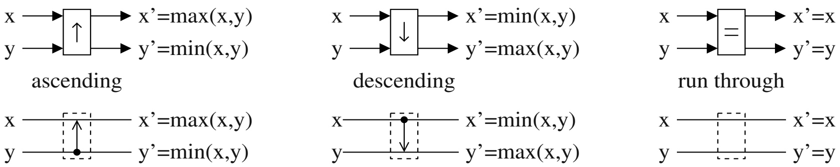


Fig. 1. The basic set of comparing switches including the ascending comparator, the descending comparator and the neutral element. Their respective representation in a Knuth diagram is given under each of them

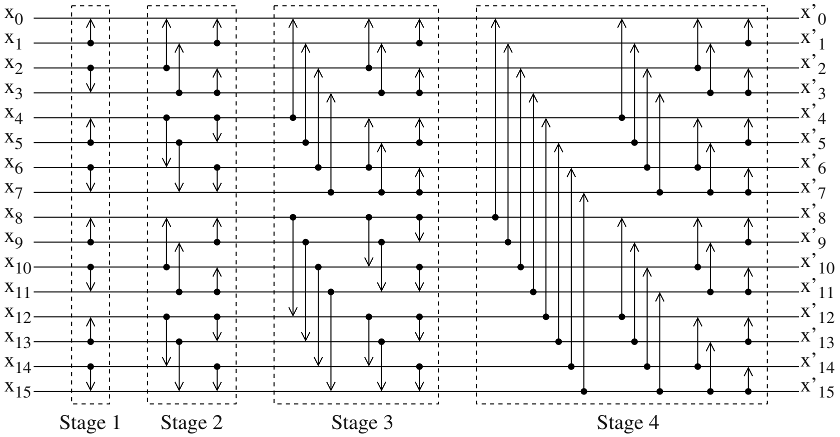


Fig. 2. Representation of the bitonic algorithm in a Knuth diagram for the sorting of 16 keys. The sorter includes 10 levels grouped in 4 stages and 80 switch comparators

scalable and reconfigurable sorter including partially fulfilled multiplexer levels at the beginning and the end of the basic structure. Section 5 presents the reconfigurable recurrent bitonic sorting network in its typical System on Chip (SoC) environment as well as the results of the FPGA implementation of its architecture regarding various input widths.

2 Batcher's Bitonic Sorter and Merger

Batcher's bitonic sorter and merger [3] is based on a comparison network scheme in which many compare and exchange operations are performed in parallel. It is an important algorithm facilitating efficient parallel implementations because the sequence of comparisons is not data-dependent. Described in figure 1, the switch comparator is the basic element of such a network, which allows two input keys to be compared and output in ascending or descending order.

A conventional comparator network for a bitonic sorter can be represented in a Knuth diagram [5], using both ascending and descending comparators, as seen in figure 2. In the represented sorting network with $N = 16$ keys, the

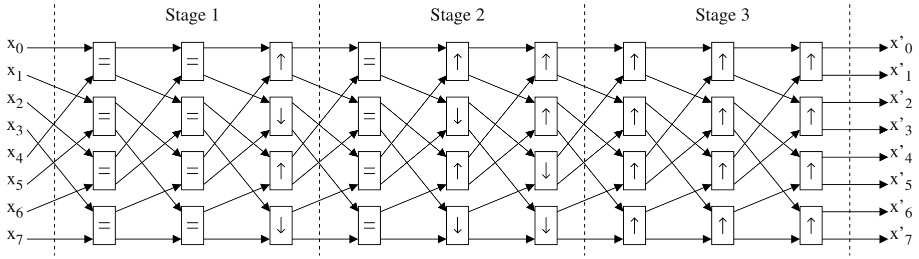


Fig. 3. A regular representation of a bitonic sorter for $N = 8$ keys including $\log_2 N = 3$ stages and using ascending, descending and run-through switch comparators

enable	up_dn	x'	y'
0	–	x	y
1	0	min(x,y)	max(x,y)
1	1	max(x,y)	min(x,y)

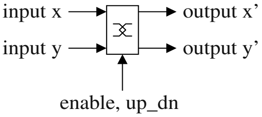


Fig. 4. Functionality and truth table of the controlled switch comparator

$N = 2^n$ horizontal lines correspond to the inputs and the arrows to the switch comparators. This network consists of n successive merging phases i with $1 \leq i \leq n$, in which pairs of sorted sequences of length 2^{i-1} are presented in oppositely sorted order and merged, according to the bitonic principle [3].

Stone showed that a sorting network for N keys with $N = 2^n$ can be regularly constructed by repeating a perfect shuffle pattern [11], as illustrated in figure 3. This sorting consists of n stages of n steps each where simultaneous operations are performed on 2^{n-1} pairs of adjacent elements.

It appears clearly that this network structure can be simplified to a single column of $N/2$ special switch comparators executing a predefined sequence of operations, at the cost of $\log_2 N$ basically neutral but necessary steps (the ones performed by the comparators switched in run-through mode). For this reason, we created a new type of switch comparator, as explained in the next section, to permit the realization of a recurrent bitonic sorting network.

3 Recurrent Bitonic Sorting Network

3.1 Controlled Switch Comparators

In oder to reduce the regular bitonic sorter seen in figure 3 to only one butterfly network followed by one level of toggle logic, we need a switch comparator performing the three different kinds of operations described above, depending on its place p in the network, with $1 \leq p \leq N/2$.

Figure 4 shows the functionalities of the comparator forming the basic element of the recurrent bitonic sorting network with the help of a truth table. The controlled switch comparator is based on the normal switch comparator and includes a few add-on logic gates allowing two new functionalities:

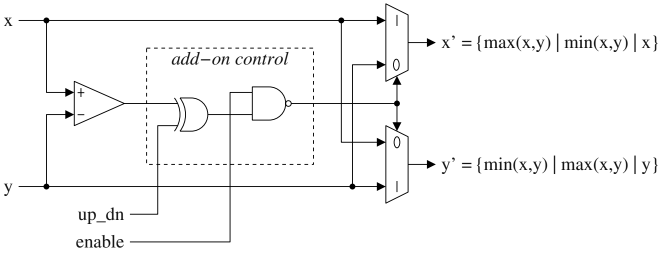

x
y
add–on control
+
−
up_dn
enable
1
0
0
1
x' = {max(x,y) | min(x,y) | x}
y' = {min(x,y) | max(x,y) | y}

Fig. 5. Add-on complementary logic with control signals `up_dn` and `enable` to transform the conventional exchange comparator into a controlled switch comparator

enable function: when the `enable` input is not selected, the switch comparator is transparent and acts as a run-through gate: $x' = x$ and $y' = y$.

up/down function: when the `enable` input is selected, the switch comparator redirects its inputs according to the level on the `up_dn` signal. When `up_dn` is at a high level, the switch element is set in ascending mode: $x' = \max(x, y)$ and $y' = \min(x, y)$. When `up_dn` is at a low level, the switch element is set in descending mode: $x' = \min(x, y)$ and $y' = \max(x, y)$.

Figure 5 shows the internal realization of such a controlled switch comparator. Two supplementary gates are necessary to decide whether a switch of the input is performed at the output. Moreover, it is safe to say that the size of the add-on control logic which does not depend on the width of the input keys x and y is negligible compared to the size of the comparator and of the swapping multiplexers. The main difference considering the design of the recurrent bitonic sorting network remains the additional control signals which have to be generated by an external dedicated controller.

3.2 Regenerated Butterfly Network

A butterfly network is build symmetrically by connecting regularly the half of the inputs to each second output, from $i = 1$ to $N/2$ and mirrored from $i = N$ to $N/2 + 1$. The following program shows the regularity of the mapping of a butterfly network for N keys or $N/2$ switch comparators in only a few lines of code, where the description of the control signal is deliberately omitted.

```
butterfly_mapping : for I in 0 to (N/2 - 1) generate
    SWITCH_COMPARATOR port map(
        INPUT_X  => NET_IN(I),
        INPUT_Y  => NET_IN(N/2 + I),
        OUTPUT_X => NET_OUT(2*I),
        OUTPUT_Y => NET_OUT(2*I +1) );
end generate;
```

(Extract of VHDL code for the realization of a butterfly network for N keys based on the generation of $N/2$ controlled switch comparators in a N keys feedback 1:1 network)

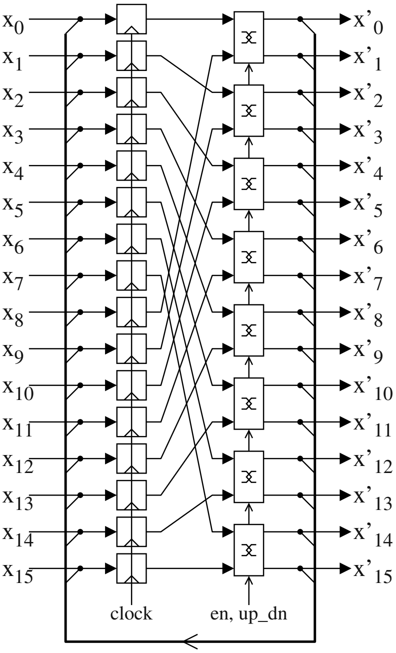


Fig. 6. A 16-keys recurrent bitonic sorting network

Figure 6 shows the result of the implementation of the previous VHDL code, by adding a level of registers and mapping directly the entries of the registers to the outputs of the controlled switch comparators over the feedback network. This sorter sorts N keys in $O(\log^2 N)$ time with only $N/2$ elements.

In the recurrent network, an additional external controller is necessary to generate the control signal described in figure 4. The `enable` and `up_dn` signals are supervised during the sorting procedure which lasts exactly $(\log_2 N)^2$ clock periods for the sorting of N keys. Let i be the iteration step in the sorting sequence where $1 \leq i \leq (\log_2 N)^2 = n^2$ and s the current stage, as described in figures 2 and 3. Depending on the position p of each comparator with $1 \leq p \leq N/2$, a sequence of length n is initiated at each re-setting of the `enable` signal to control `up_dn`. Let t be a time index over this sequence, with $1 \leq t \leq n$.

$$\texttt{enable} = f(i, n) = \operatorname{sign}\big(i - (n-1) \cdot s - 1\big), \quad \text{with } s = \lceil i/n \rceil \tag{1}$$

$$\texttt{up\_dn} = f(p, t, n) = \left\lceil \frac{2^n - p}{2^t} \right\rceil \bmod 2 \tag{2}$$

These equations describe mathematically the value of the control signals and are not meant to be implemented this way in hardware. The sequences of values can be read directly from figure 3 for the example of $N = 8$ keys, where `enable` equals 0 for each switch comparator marked with “=” otherwise 1, and `up_dn` equals 1 for “↑” and 0 for “↓”, otherwise it can be ignored.

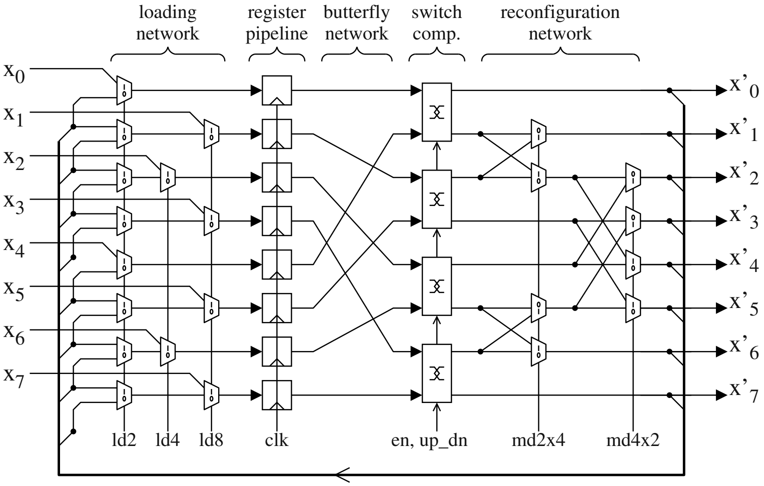


Fig. 7. Internal architecture of an 8 keys reconfigurable recursive bitonic sorter including the loading network, the butterfly network, the reconfiguration network, one level of toggle logic and one level of pipeline registers

Considering that the first run-through comparisons can be disregarded, it is possible to modify the controller so that it skips the insignificant steps of the first phase to accelerate the whole sorting. The final remaining time for this model is then expressed in the following equation:

$$t_{\mathrm{BIT\_REC}} = (\log_2 N)^2 - \log_2 N + 1 \tag{3}$$

While the time complexity of the sorter in $O(\log^2 N)$ is the same as the original bitonic sorting network, but its complexity in terms of logic gates and comparators has been drastically reduced from $O(N \cdot \log^2 N)$ to $O(N)$.

4 Scalability and Reconfiguration

The recurrent bitonic sorting network presented in the previous sections is scalable in the sense that it is adaptable to any kind of bus width, as long as the number of key to sort is a power of two ($N = 2^n$). Slightly diverging realizations of redistributing networks for sorting were found in the literature where the main idea was either to use special design environments to verify the lengths of intermediate wires [4] or to resort to the parity strategy to reduce the communication within the sorter [7].

Figure 7 gives the example of a reconfigurable recurrent bitonic sorting scheme adaptable to different widths N/k, with $k = 1, 2, 4 \ldots N/2$. For this reason, the depicted network includes two additional internal subnetworks, compared to the regular recurrent bitonic sorter seen in figure 6:

The loading network permits a partial loading of a new set of keys and a reuse of the comparators as if they were pipelined. The equivalent architecture remains the one of k recurrent bitonic sorters for N/k keys. The total amount of multiplexers needed can be calculated with the following equations:

$$\mathrm{Mux}_{LN}(N) = \sum_{i=1}^{N/2} \frac{N}{i} = 2 \cdot (N-1) \qquad (\text{with } N = 2^n \geq 4) \tag{4}$$

The reconfiguration network is controlled by an external register setting the complete network in a lower configuration, so that the feedback loop completes the design as if the k sorters of N/k keys were totally independent. The total amount of multiplexers needed can be estimated with the following equations:

$$\mathrm{Mux}_{RN}(N) = (n-2) \cdot 2^{n-1} = (\log_2 N - 2) \cdot \frac{N}{2} \qquad (\text{with } N = 2^n \geq 4) \tag{5}$$

Table 1. Configuration table of the scalable recurrent bitonic sorting network. The example is given for a $N = 8$ key sorter according to figure 7 with divider k varying

div	mode	reload	at loading:	settings:	input	output
(k)	(keys)	period	ld$_2$ ld$_4$ ld$_8$	md$_2$ md$_4$	keys	keys
1	1×8	7 clk$^{(\dagger)}$	1 1 1	0 0	x_{0-7}	x'_{0-7}
2	2×4	2 clk	1 1 0	1 0	$x_{0,2,4,6}$	$x'_{1,3,5,7}$
4	4×2	1 clk	1 0 0	1 1	$x_{0,4}$	$x'_{3,7}{}^{(*)}$
k	$k \times \frac{N}{k}$	$\frac{1}{k}(\log_2 \frac{N}{k})^2$	$\mathrm{ld}_i \Leftarrow \frac{N}{k} \geq i$	$\mathrm{md}_i \Leftarrow k \geq i$	$\{\mathrm{x}_{k \cdot i}\}$	$\{\mathrm{x'}_{k \cdot i+k-1}\}$

Based on figure 7, the example of an $N = 8$ key sorting network with the parameter k varying from 1 to the theoretical limit $N/2$ is given in table 1. The first line marked $^{(\dagger)}$ corresponds to a special case in which the first run-through comparisons can be disregarded by the controller, see eq. 3 for explanations. The penultimate line $^{(*)}$ of the table is here just for showing the regularity of the network but is useless because only one step is necessary for comparing 2 keys. For the same reason, the rightmost crossing nodes in the reconfiguration network part of figure 7 are not meant to be in an 8 keys recursive sorter, but their aim is to show the distribution of the $N/2$ multiplexer on each level. The last line shows the generic case regarding the divisor k for the N keys input bus.

This leads to the generic representation of the bitonic scheme, as seen in figure 8. It shows a global loop endorsing the direct feedback of N keys within the sorter and all the necessary elements for making it reconfigurable and adaptable to different widths. Due to its regularity and symmetry, it is possible to use the same bus as input and output to feed the network or read the results from the outside. This feature is important in the realization of systems including a single master-slave bus for the data exchange, as shown in the next section.

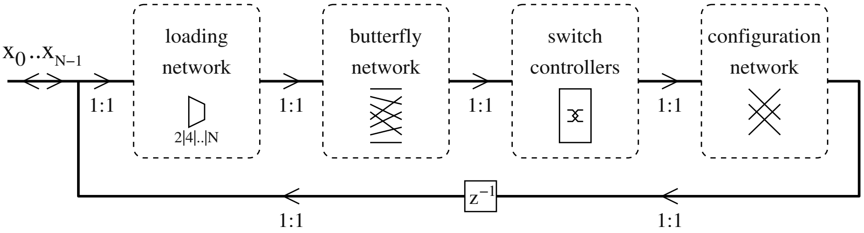


Fig. 8. Block schematics of a generic reconfigurable recurrent bitonic sorting network. Note that the register level (z^{-1} box) can be placed anywhere in the loop

5 Hardware Implementation

The reconfigurable recurrent bitonic sorting network can be seen as a generic sorter core or as a piece of IP (Intellectual Property) with configuration parameters for any custom use. Typical applications for such sorters are for instance ATM switches [1] [10] or the acceleration of graphics algorithms for rendering and ray tracing [12], where the amount of data to be ranged in varies significantly. In each case, a particular attention is given to producing compact layouts.

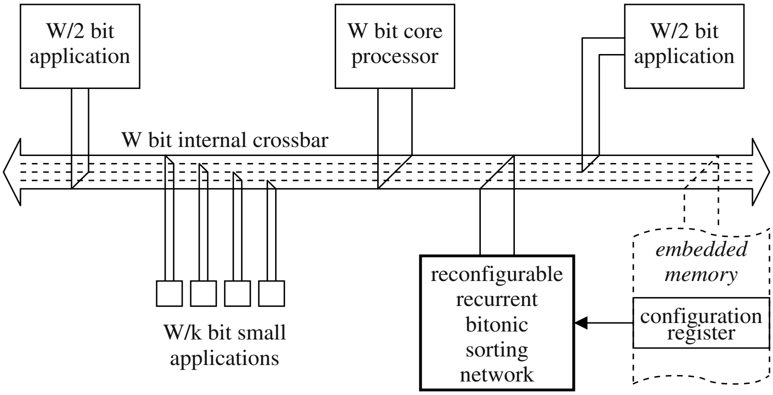


Fig. 9. A reconfigurable recurrent bitonic sorting network within a typical SoC environment. All the components are connected to the system bus with different widths. The W bit wide internal crossbar handles in parallel k keys with a length of W/k bit

Figure 9 shows a reconfigurable recurrent bitonic sorting network within a typical SoC environment. The system is built around a large internal bus allowing the communication and data exchanges between the elements. Though the processor core remains the main part of the chip, the other blocs are application specific hardware accelerators providing a significant performance increase. In this example, W is the width in bit of the internal crossbar which is basically a power of two, and k the degree of parallelism, or in other words the number of keys which can be handled in parallel in the sorting unit. Most of the highly parallel system architectures like superscalar or vector processor based architectures [6] include many processing elements which can be connected to each other over different widths, performing tasks involving various data types.

The design of the reconfigurable recurrent bitonic sorter core is based on a single butterfly network and a small regular reconfiguration network, as seen in figure 8. To produce an efficient sorting network, it is necessary to implement the butterfly network on the chip (in this case an FPGA) carefully by placing thoroughly each element of the design so that the amount of cross connections is minimized. This task can be performed efficiently by resorting to manual place and route tools. Four input widths have been implemented in the sorter presented in the previous section on a Xilinx XCV2000E of the Virtex-E FPGA series to allow us to evaluate area and speed performance.

Table 2. Implementation of the reconfigurable recurrent bitonic sorting network in a Xilinx Virtex-E XCV2000E FPGA with different input widths (16 to 128 keys)

N (keys)	Max. Period (ns)	Max. Speed (MHz)	Area (Slices)	FPGA usage
16	11.958	83.626	684	3%
32	12.088	82.727	1227	6%
64	12.142	82.359	2694	14%
128	13.588	73.594	5390	28%

Table 2 summarizes the results of the synthesis of the four architectures, considering N keys having a length of 24 bit. We can appreciate the size of the designs in terms of look-up tables (LUT) and register pairs. The Xilinx synthesis tools report the area in “slices”, where a Virtex-E slice contains two LUT and two registers. As the number of keys N is doubling, so is the area of the architecture. The maximal speed of the design remains not significantly affected by this increase for N up to 64 keys.

6 Conclusion and Outlook

The bitonic sorter presented here is an efficient, compact and scalable parallel sorting network. When it is used in a recurrent architecture where only one or a few comparator levels are present, this sorter is very appropriated in area critical applications. The use of a feedback network for a parallel redistribution thus influences the mapping and the overall flexibility of the architecture but minimizes the costs in terms of hardware resources.

We showed that the reconfigurability of this sorter allows its utilization in diverse applications requiring sorting tasks and increases the flexibility of shared system buses in SoC designs. Moreover, hardware reuse within the reconfigurable recurrent bitonic sorting network delivers an optimal compromise between delay time and chip area.

References

1. Ahmadi, D., Denzel, W.E.: A Survey of Modem High-Performance Switching Techniques. IEEE Jal. on Selected Areas in Comm., vol. 7, no. 7 (1989) 1091–1103
2. Ajtai, M., Komlos, J., Szemeredi, E.: An O(N log N) Sorting Network. Proceedings of the 15th Annual Symposium on Theory of Computing (1983) 1–9
3. Batcher, K.E.: Sorting Networks and Their Applications. Proceedings of the Spring Joint Computer Conference, AFIPS, vol. 32 (1968) 307–314
4. Claessen, K., Sheeran, M., Singh, S.: The design and verification of a sorter core. Proceedings of the 11th Advanced Working Conference on Correct Hardware Design and Verification Methods, vol. 2144 of LNCS, Springer-Verlag (2001) 355–369
5. Knuth, D.E.: The Art of Computer Programming, vol. 3, Sorting and Searching. Chapter 5.3: Optimum Sorting. 2nd Edition, Addison Wesley (1997)
6. Kozyrakis, C., Patterson, D.: Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. Proceedings of the 35th International Symposium on Microarchitecture, ACM/IEEE (2002) 283–293
7. Lee, J.D., Batcher, K.E.: Minimizing Communication in the Bitonic Sort. IEEE Transactions on Parallel and Distributed Systems, vol. 11, no. 5 (2000) 459–474
8. Leighton, F.: Tight Bounds on the Complexity of Parallel Sorting. IEEE Transactions on Computers, vol. 34, no. 4 (1985) 344–354
9. Nakatani, T., Shing-Tsaan, H., Arden, B., Tripathi, S.: K-Way Bitonic Sort. IEEE Transactions on Computers, vol. 38, no. 2 (1989) 283–288
10. Sharma, N.K.: Modular Design of a Large Sorting Network. Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks, IEEE (1997) 362–368
11. Stone, H.S.: Parallel processing with the perfect shuffle. IEEE Transactions on Computers, C-20 (1971) 153–161
12. Westermann, R., Ertl, T.: Efficiently Using Graphics Hardware in Volume Rendering Applications. Proceedings of the 25th International Conference on Computer Graphics and Interactive Techniques, vol. 32, no. 4 (1998) 169–179

A Framework for Energy Efficient Design of Multi-rate Applications Using Hybrid Reconfigurable Systems

Sumit Mohanty and Viktor K. Prasanna

University of Southern California, CA, USA
{smohanty, prasanna}@usc.edu

Abstract. Hybrid reconfigurable systems integrate DSPs and general purpose processors with an FPGA fabric. These systems may support features such as efficient start-up and shut-down, dynamic voltage scaling, and reconfiguration, that are exploited for energy-efficient application design. Duty cycle is the proportion of time during which a system is operated. Multi-rate applications consist of tasks that execute at different rates. Designing an energy-efficient hybrid reconfigurable system with duty cycle specification that implements a multi-rate application using devices with multiple operating states presents several challenges such as modeling, rapid performance estimation, and efficient design space exploration. We present a design framework that addresses these challenges. Using our framework, we illustrate the design of two energy efficient systems: automated target detection and adaptive beamforming.

1 Introduction

Due to dramatic increase in the number of portable and embedded applications, systems implementing signal processing applications which were formerly limited by the available computation power are currently limited by the available energy [11]. Examples of such systems include mobile base stations for software defined radio [4] and target detection and tracking systems [13]. A system consists of an application and a target hardware that implements the application. Our target applications, implemented by the systems discussed above, are multi-rate signal processing applications that can be modeled as data flow graphs [7]. The input is assumed to be a data stream consisting of several frames of data with a specified frame rate. The target hardware is a hybrid reconfigurable system that integrates instruction set architecture (ISA) based processors [5,12] with a reconfigurable fabric. The tradeoff considered here is high throughput and relatively higher average power dissipation using the FPGAs vs. comparatively lower throughput and power efficient solution using the general purpose processors. A hybrid reconfigurable system can be a single chip device such as the Xilinx Virtex-II Pro [15] or a multi-chip device such as, for example, an Actel ProASIC [1] and a TI DSP [14].

Given a target application, design of a hybrid reconfigurable system involves (device) selection of suitable processing components (FPGAs and ISA-based

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 658–668, 2004.

processors) and memory. Following device selection, the mapping between individual application tasks and processing components, appropriate operating state for each mapping, and the schedule of execution are identified based on the performance requirements specified as an input. The designer also needs to identify appropriate hardware/software partitioning onto those platforms. In addition, other capabilities that play a significant role, especially for energy efficient design, are reconfiguration, dynamic voltage scaling, choice of low-power operating states, and device activation scheduling based on the duty cycle specification [10]. Duty cycle is the proportion of time during which a system is operated. Such specification allows modeling of a period of execution as alternate active and inactive phases. Energy dissipation (e.g. due to leakage current), especially for systems with low duty cycle, during the inactive phases can contribute significantly to the overall energy dissipation of the system. Therefore, the tradeoff between the performance cost of shutting down and starting up a device and the performance cost of remaining idle needs to be considered during system design [2]. In addition, an application is defined as multi-rate if the constituent tasks execute at different rates. For example, an adaptive beamformer processing up to 105 mega samples per second may need to update the weight coefficients only once every second [3]. Therefore, tasks performing weight coefficient evaluation and update execute once every second. On the other hand, the tasks that process each data sample are executed 105×10^6 times every second (see Section 5.2).

In this paper, we present a framework that supports energy efficient design of multi-rate applications based on duty cycle specifications using a hybrid reconfigurable system. Our primary focus is system design problems with performance requirement of minimizing energy while sustaining a given input rate. Furthermore, the task rates affect the quality of the output [3]. Our framework can also be used to evaluate the target hardware(s) in terms of the task execution rates supported for multi-rate applications and range of input rates sustained based on latency requirements. The framework is based on a hierarchical design technique [10] which involves modeling and a two step design space exploration (DSE).

The paper is organized as follows. Section 2 discusses related work. The modeling approach is discussed in Section 3. Section 4 discusses the design framework. Two examples demonstrating the framework are presented in Section 5. We conclude in Section 6.

2 Related Work

There are a number of research and commercial initiatives that address the design of efficient systems.

The SPADE (System Level Performance Analysis and Design Space Exploration) methodology evaluates embedded system architectures at multiple abstraction levels to perform DSE [8]. DSE involves quick evaluation of a large number of designs using coarse-level architecture models followed by the use of

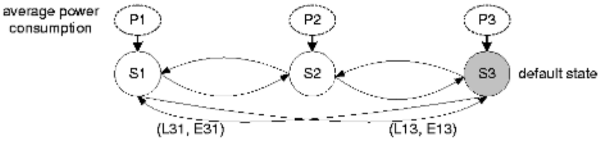


Fig. 1. Modeling multiple operating states

more accurate and detailed models as the number of designs reduces. In contrast, we use a hybrid approach by integrating optimization heuristics and high-level performance estimation. Because of optimization heuristics, we can rapidly evaluate a much larger design space prior to estimation (or simulation) based design space exploration. Additionally, our augmented data flow (DF) model for applications enables performance analysis of multi-rate applications based on operating state transition costs when the target devices support multiple operating states.

Among the commercial tools, Xilinx System Generator for Simulink provides a high-level interface for application design using pre-compiled libraries of signal processing kernels [15]. However, this tools starts with a single conceptual design. Design space exploration is performed as part of implementation or through local optimizations to address performance bottlenecks identified during synthesis. Additionally, these tools [15] are specific to a set of target devices and cannot be used for device selection problem.

Benini et al. discuss various techniques for dynamic power management such as turning off system components or moving system components to a low performance operating state [2]. However, to the best of our knowledge, there are no tools available that take into consideration both multi-rate applications and duty cycle specifications during system design to reduce energy dissipation.

3 System Modeling and Design Problems

A set of models are used in the design framework to capture the system specification (application, target hardware, mapping, performance, etc.).

3.1 Modeling Multiple Operating States

We model a candidate devices based on the operating states supported by the device. An operating state can be a configuration for an FPGA or a voltage/frequency setting for devices supporting voltage/frequency scaling. In addition, given two operating states A and B, we assume that the transitions from A to B and B to A are associated with transition costs (latency and energy).

Our model is based on an augmented finite state machine (FSM). Figure 1 shows a sample model for a device with 3 operating states. Each node in an FSM represents one operating state. Each pair of nodes is connected with a pair of directed edges. Each edge correspond to a state transition from the state represented as the source node to the state represented as the destination node. Each edge is also associated with a 2-tuple (TL, TE) where TL is the latency cost and TE is the energy dissipation during the transition. Each operating state is associated with an estimate of average power consumed while idling ($P1$, $P2$, $P3$ in Figure 1). The model also indicates a default state (shown in gray).

3.2 Multi-rate Application Modeling and Mapping Specification

To model the application, initially, we create a (data flow) DF model of the application assuming that all the tasks will execute at the same rate. The rate is assumed to be 1 which refers one execution of a task per input frame. Rate of 1 is the maximum rate possible for any task. Afterwards, we associate each task with their respective rates. A rate of r refers to one execution per r input frames. Therefore, given a multi-rate application, for some input frames certain tasks will not be executed. In such cases, we assume that the tasks preceding and succeeding the task not being executed are connected (Figure 2) and thus the dependencies between the tasks are maintained.

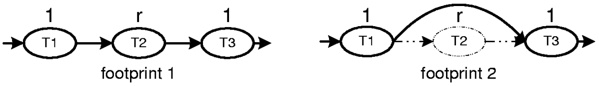


Fig. 2. Multi-rate application modeling

Based on the serial order ($1st$, $2nd$, $\cdots$, nth) of the frame being processed, the data flow graph will change as it needs to model only the active tasks. We define each distinct DF model for the application as a *footprint*. Figure 2 shows two footprints of the same application with three tasks. Each footprint is analyzed by our framework to identify the most energy efficient design per footprint while performing a duty cycle based design space exploration. A *master footprint* is defined as the footprint of the application when all tasks are active.

In addition, each application task is associated with a set of alternatives. Alternatives refer to the mappings of application tasks on different target devices operating in different operating states. For example, there will be two alternatives if a task can be mapped on a general purpose processor and a DSP. If each device has two operating voltages, then there will be four alternatives instead.

3.3 Modeling Duty Cycle

Duty cycle is the proportion of time a system is active. Therefore, based on the duty cycle specification, system execution can be modeled as alternate active and inactive phases. Duration of each phase depends on the input rate and the latency required to process a single input frame. Input rate can be variable. Our design framework needs the input rates to be statically defined. Variable input rate is specified as an ordered set of 2-tuples (Ir, Nf) where Ir refers to the input rate in Hz and Nf refers to number of frames processed at the above rate. Given two consecutive 2-tuples, (Ir_i, Nf_i) and (Ir_{i+1}, Nf_{i+1}), application execution is modeled as "process Nf_i frames at the rate of Ir_i Hz followed by Nf_{i+1} frames at the rate of Ir_{i+1} Hz". Our framework assumes that the given set of 2-tuples can repeat indefinitely to model the processing of large number of input frames. Similarly, maximum allowed latency to process an input frame can also be specified. If not specified, the framework derives the constraint based on the given maximum input rate.

3.4 Constraints

The constraints are divided into two types; *performance constraints* and *design constraints*. Performance constraints are of the form "metric < constant" where the metric can be latency or energy. The design constraints specify valid combinations of task mappings through pairwise relations between two mappings (e.g. task $A1$ implemented on device $B1$ operating in state $C1$ is not compatible with task $A2$ implemented on device $B2$ operating in state $C2$, or task X can be mapped only to device R). Similarly, design constraints also specify valid combinations of devices. As our focus is designing a hybrid reconfigurable system, given one FPGA, two DSPs, and two general purpose processors, one can specify a constraint that a valid target hardware is a combination of one FPGA, and one DSP or one general purpose processor only.

3.5 System Design Problems

The models discussed above define a design space. Due to large number of design choices such as target devices, operating states, mappings, device shut down or leave on, etc., the size of the design space can be very large (see Section 5.1). Traversal of such a large design space is not practical using a simulators or even high-level estimators (see Section 5). Additionally, to estimate the performance of a candidate design, during estimation, it is required to support duty cycle, multi-rate applications, and multiple operating states for the devices. Our framework addresses the above issues using a hierarchical approach towards DSE and an enhanced high-level performance estimator. The framework supports the two classes of system design problems (a) **device selection**: design a system by selecting a combination of devices from a given set of candidates while satisfying the performance and design constraints and (b) **hardware evaluation**: evaluate a target hardware in terms of the input rates or the range of task execution rates supported.

4 System Design Framework

The models discussed above form the basis for the system design framework. The framework supports user friendly visual modeling to specify multi-rate applications, target devices, duty cycle specifications, mappings, and constraints.

The system design framework is implemented using MILAN, a **M**odel-based **I**ntegrated Simu**LA**tio**N** environment [9].

4.1 Design of the Framework

MILAN is a model based integrated simulation environment for embedded system design and optimization through the integration of various simulators and tools into a unified environment [9]. Using the MILAN environment, the designer formally models the target application, target devices, and constraints through a graphical interface. The models are used to drive various tools and simulators integrated in MILAN. Additional details about the MILAN environment can be found in [9]. We extended the MILAN modeling paradigm to support the models discussed in Section 3. Specific extensions were support for multi-rate application modeling, devices with multiple operating states based on the augmented finite state machine model, and duty cycle specification. MILAN integrates a heuristic based design space exploration tool, DESERT [6] and HiPerE, a high-level performance estimator [10].

DESERT supports DSE when the design space is defined by an application modeled as a data flow graph with alternatives [10]. Alternatives are associated with a task and refer to different mappings of the task onto devices. Given a set of performance and design constraints, DSE using DESERT refers to identification of a set of designs that satisfy the given constraints. DESERT does not support devices with multiple operating states while performing DSE. We developed a technique that uses pseudo tasks and additional design constraints to enable DESERT perform DSE using devices with multiple operating states. This technique introduces a pseudo task between each pair of connected (via an edge in the model) tasks to model operating state transitions. The alternatives for this pseudo task are all possible operating state transitions among the devices. We specify a set of design constraints to ensure that appropriate state transition is chosen based on the mappings chosen for the pair of tasks. Thus, when DESERT performs DSE and selects the designs that meet the design constraints, correct operating state transitions are also automatically chosen. In addition, operating state transition costs also get added to the overall performance of each design while evaluating the designs against the performance constraints.

We enhanced HiPerE to support duty cycle specification, multi-rate applications, and devices with multiple operating states. Given a system design and the number of input frames to process, HiPerE estimates the overall latency and energy dissipation of the system. HiPerE uses the models discussed in Section 3 to extract the required performance estimates for task mapping, operating state transitions, etc. By default, HiPerE assumes that the devices are idle during inactive phases. However, HiPerE can be configured to shut down the devices during the inactive phase in which case HiPerE takes into account the start up cost as well for the next active phase.

4.2 Design Flow

The design flow begins with modeling of the application and the target devices. Target devices are modeled using the augmented finite state machine (FSM) model. The performance estimates for various components of the augmented FSM are derived from vendor provided data sheets. Application modeling involves application specification as a data-flow graph with alternatives. Rate of execution is also specified for each application task. The functional specification of the target system specifies the structure of the data-flow graph and the choice of implementations specifies the alternatives. Each alternative is mapped onto a target device operating in a specific state. Each mapping is associated with performance estimates in terms of latency and energy dissipation. Constraint specification follows modeling.

The design framework uses a hierarchical approach for design space exploration (DSE). The basic idea of hierarchical approach is a two step DSE [10]. Step-I uses a heuristic based design space exploration technique to quickly evaluate a large design space and identify a set of design that satisfy the given performance and design constraints. Step-II uses a high-level performance estimator to evaluate the selected designs to identify the most energy efficient design. In our framework, Step-I is performed using DESERT and Step-II is performed using HiPerE (Figure 3). The key to fast DSE using the hierarchical approach is a rapid DSE in Step-I using an approximated model followed by exhaustive search based on a more detailed model [10].

In Step-I, we do not consider the duty-cycle specification and the multi-rate specification of the application while performing DSE. We only consider the multiple operating states of the target devices, the design constraints, and the latency constraint based on the minimum input rate to be sustained. The designs rejected by Step-I are the ones which do not satisfy the given design constraints and the latency constraint based on minimum input rate to be sustained. Therefore, our approach guarantees that none of the rejected designs would have satisfied the given performance requirement even when duty cycle and multi-rate are considered. DSE in Step-I using DESERT is fast. Our experience with DESERT shows that we can prune a design space with $10^4 \sim 10^5$ designs in order of minutes on a typical uniprocessor system [10]. If the number of designs is too large, we can use energy constraint to further evaluate the designs. The energy constraint can be relaxed (tightened) if too few (many) designs are selected.

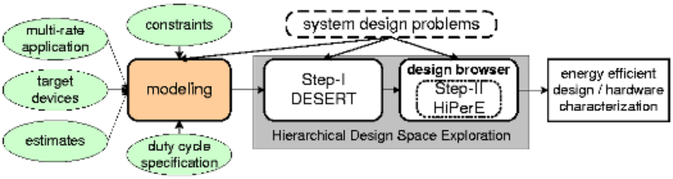


Fig. 3. System design approach

In Step-II, the selected designs are evaluated using HiPerE based on the duty cycle specification and the multi-rate aspect of the application to identify design(s) based on the specific system design problem being solved. Using HiPerE, a designer specifies the duty cycle and the rates of different tasks and evaluates all the designs. The most energy efficient design is selected manually. In addition, HiPerE provides automated support to evaluate designs for range of input rates and ranges of rates supported for each task.

5 Illustrative Examples

We use two problems to demonstrate the flow of our design framework. The first problem identifies an energy-efficient hardware for a target detection application [13] from a set of devices. Target detection is widely used in radar applications, surveillance videos, and sensor networks to predict position and velocity [13]. The second problem also solves the device selection problem for an LMS (Least Mean Square)-based MVDR (Minimum Variance Distortionless Response) adaptive beamforming algorithm [3] and evaluates the selected target hardware(s) in terms of the rate of weight coefficient update. LMS-based MVDR adaptive beamforming algorithm is used in the base station for software defined radios to design smart antennas [3].

5.1 Personnel Detection Application

The application consists of 5 tasks (Figure 4) [13]. Tasks shown in gray have rates higher than 1. The hardware needs to be selected from a set that consists of Xilinx Virtex-II Pro, Actel ProASIC^{PLUS}, Intel PXA 255, PowerPC 405, and TI C6711 DSP. Application modeling involved specification of the application as an augmented data flow graph (Figure 4). Individual target devices were modeled using the augmented FSM model. Possible mapping choices for each task were also indicated. For example, inverse and whiten needs to be computed using floating point arithmetic for which Actel ProASIC^{PLUS} is not a suitable choice (due to a smaller number of gates). Such requirements are indicated by not allowing inverse and whiten to be mapped onto ProASIC^{PLUS} while modeling. Additionally, the 4 valid device combinations are, 1) Virtex-II Pro only, 2) ProASIC + DSP, 3) ProASIC + PXA 255, and 4) ProASIC + PPC 405. The rate of CVM computation and inverse was specified as 2 and input rate was specified as 0.5 Hz. The latency constraint was specified as ≤ 2 seconds. We also consider only DSP as the fifth choice for performance comparison between DSP and hybrid reconfigurable system. Modeling effort required for this experiment was approximately 4 hours. Modeling effort does not include simulation or estimation necessary to estimate performance of mapping and state transitions. However, as our framework supports model reuse, simulation performed off-line can be reused during modeling.

Following modeling, we performed design space exploration using DESERT and HiPerE. Through several iterations using DESERT, energy constraint of

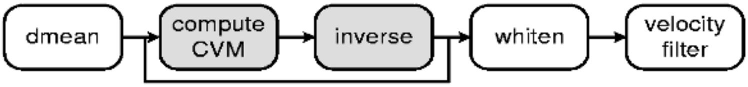


Fig. 4. Application model for Personnel Detection

$\leq$ 860 mJ was chosen to have DESERT select 16 designs as output of Step-I. The size of the initial design space was approximately 73, 000. Once 16 designs were identified by DESERT, we used HiPerE to perform Step-II to identify the best design that meets the duty-cycle requirements and dissipates the minimum energy. From the 16 designs, we chose 3 designs with different target hardwares for analysis (Table 1).

Table 1. Results for device selection

Designs	Scenario 1		Scenario 2	
	Latency	Energy	Latency	Energy
Virtex-II Pro only	247.33 ms	114.06 mJ	17895 ms	13938.1 mJ
TI DSP only	614.57 ms	670.11 mJ	18286 ms	9692.7 mJ
TI DSP + ProASIC	496.50 ms	538.18 mJ	17166 ms	8652.9 mJ

In Scenario 1, the system processes only one frame (no start up or shut down cost included) and in Scenario 2, the system processed 10 input frames (Table 1). Note that though the Virtex-II Pro based design is the most energy-efficient for Scenario 1, it is the least energy-efficient for Scenario 2. This is due to high energy dissipation during inactive phases. In both the scenarios, the hybrid reconfigurable system out-performs the DSP based system. Based on the above analysis, we selected TI DSP and ProASIC^{PLUS} as the target hybrid reconfigurable system.

DSE using our framework was performed using a PC with a 848 MHz Pentium III. The use of an optimization heuristic in Step-I allows us to evaluate a design space of size 73, 000 in less than a minute. Step-II, evaluating 16 designs, also took about 2 minutes. On the other hand, if we use HiPerE (which runs significantly faster than low-level simulators), it takes approximately 10 hours to estimate the performance of all the designs and a tedious manual comparison of all the estimates to identify the most energy efficient design.

5.2 Adaptive Beamforming for Software Defined Radio

The LMS-based MVDR algorithm consists of three steps. The first step is the calculation of filter output. The second step is the calculation of input signal power, correlations between the signals, normalization, and the calculation of error signals. The third step is the update of the weight coefficients of the adaptive filter. The incoming data rate is approximately 105×10^6 samples per second [3]. This provides us with the latency constraint of ≤ 9.5 ns.

Typically, the rate of weight coefficient update is once every second [3]. However, the rate of update depends on the stability of the environment and the application requirement. For example, an environment with high interference and path-loss might require a higher rate of update. Therefore, in addition to device selection, we also use the framework to identify the maximum rate of update that can be supported without affecting latency. The set of devices considered includes two Xilinx Virtex-II Pro devices (xc2vp2 and xc2vp20), one Xilinx Virtex-II (xc2v1500), Intel PXA 255, PowerPC 405, and TI C6711 DSP. Most of these devices are also used by the system design problem discussed in Section 5.1. Thus, we were able to reuse the models of the devices defined in MILAN. The size of the design space was 243.

HiPerE was configured to power down components when idle if it reduces the overall energy dissipation. Table 2 shows the designs selected by our framework. Scenario 1 refers to coefficient update rate of once every 105×10^6 samples. In Scenario 2, the coefficient update rate is the maximum that can be sustained by a design. We evaluated all the designs based on their performances when the system executes for a period of 1 second. For each design, Table 2 shows the energy dissipation per scenario. The last column shows the maximum update rate that can be supported.

An additional advantage of using our framework is the savings in design time due to reduction in the simulation time. Cycle-accurate simulation of the processors and RT-level simulation of the FPGAs are time consuming. For example, simulation of the task filter using the DSP simulator takes approximately 15 minutes and using ModelSim and XPower takes about 30 minutes. Therefore, simulating all the 243 designs would take 100+ hours. Using our approach, we could identify the results in minutes. There are some overheads associated with computing the performance estimations of different mappings while modeling. However, in the worst case, the overhead will be 5 simulations (one simulation of the three stage application per device) as opposed to 243 simulations.

Table 2. Candidate designs and max update rates supported

No.	designs	total energy dissipated (mJ)		max rate
		Scenario 1	Scenario 2	supported
1	two xc2vp2	921.6	1113.1	8
2	xc2vp2 + PowerPC	816.6	755.6	12×10^3
3	xc2vp2 + PXA 255	682.6	839.6	11×10^3
4	xc2vp2 + TI DSP	490.6	1960	1.5×10^3
5	xc2v1500	792.3	852.2	8
6	xc2vp20	968.6	1019.6	8

6 Conclusion

Our framework implements a hierarchical approach for hybrid reconfigurable system design based on duty cycle specification and multi-rate applications using devices with multiple operating states. DSE using our framework is faster than DSE using only low-level simulators or high-level estimators. In addition, we provide a user friendly interface for modeling and DSE. We depend on the vendor provided data sheets and third-party simulators to estimate various model parameters (e.g. task mapping costs, operating state transition costs, etc.). The quality of estimates affects the quality of the output of our framework. In addition, our framework does not perform synthesis for which we rely on the design tools provided by respective vendors [1,15].

References

1. Actel ProASIC^{PLUS} `http://www.actel.com/`
2. L. Benini, A. Bogliolo, and G. De Micheli "A Survey of Design Techniques for System-level Dynamic Power Management," IEEE Tran. on VLSI Systems, Volume: 8 Issue: 3, June 2000.
3. M. Devlin, "Product Focus DSP: How to Make Smart Antenna Arrays," Xcell Journal Q1 2003.
4. C. Dick, "FPGA: Enabling the Software/Reconfigurable Radio," Advanced Radio Technologies, 2003.
5. Intel PXA 255 Processors. `http://www.intel.com/design/intelxscale/`
6. A. Ledeczi, J. Davis, S. Neema, and A. Agrawal, "Modeling Methodology for Integrated Simulation of Embedded Systems," ACM Transactions on Modeling and Computer Simulation, January 2003.
7. E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," Proc. of IEEE, Vol. 75, Sep. 1987.
8. P. Lieverse, P. Van Der Wolf, K. Vissers, and E. Deprettere "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems," JVLSI Signal Processing for Signal, Image and Video Technology, Nov. 2001.
9. Model-based Integrated Simulation. `http://milan.usc.edu/`
10. S. Mohanty and V. K. Prasanna, "A Hierarchical Approach for Energy Efficient Application Design Using Heterogeneous Embedded Systems," Compilers, Arch., and Synthesis for Embedded Sys., 2003.
11. J. Ou, S. Choi, and V. K. Prasanna, "Energy Efficient Hardware/Software Co-Synthesis on Platform FPGAs," Journal on Embedded Systems, June 2004.
12. PowerPC 405 Embedded Cores. `http://www.ibm.com/`
13. P. Singer, "The Optimal Detector," SPIE Conf.: Signal and Data Processing for Small Targets, 2002.
14. TI TMS320 Series DSPs. `http://dspvillage.ti.com/`
15. Xilinx Virtex-II Pro and Xilinx System Generator for Simulink (Matlab). `http://www.xilinx.com/`

An Efficient Battery-Aware Task Scheduling Methodology for Portable RC Platforms[*]

Jawad Khan and Ranga Vemuri

Department of ECECS, University of Cincinnati, Cincinnati, OH 45221-0030
{jkhan, ranga}@ececs.uc.edu

Abstract. In this paper we present a simple yet efficient methodology for battery-aware task execution on FPGAs in portable Reconfigurable Computing (RC) platforms. We divide the reconfigurable area on an FPGA into several fixed reconfigurable slots called *Configurable Tiles*. We then schedule real-time tasks onto these tiles. Various schedules using different number of tiles are calculated off-line. These schedules along with their execution times are then sent to a run-time scheduler which dynamically decides, which schedule is the most battery efficient. By varying the number of tiles used for scheduling tasks, we can vary the battery usage and lifetime. We tested the methodology by running it on several different task graph structures and sizes, and report an average of 14% and as high as 21%, less battery capacity used, as compared to non-optimal execution. Finally, we present a case study where we implement a real-time face recognition algorithm on the iPACE-V1 [6] platform using the proposed methodology and observed 1.3 to 3.3 times improvement in battery life-time.

1 Introduction

The battery is a finite and perhaps the most precious resource in a portable system: When opportunities of charging it become limited its value becomes even more elevated. Most of the work on battery-aware task scheduling is in the area of embedded systems where dynamic voltage and frequency scaling policies are used extensively [5]. The system can have a single processor where tasks are executed with varying voltage/frequency assignments [11,14] or the system can be a mix of heterogeneous processing units such as general purpose processors, ASICs and analog circuits etc [10]. RC based mobile systems are not only capable of supporting the traditional methods for improving battery-life reported in [5], but also it is possible to change the hardware implemented to achieve battery savings. It has been shown that [12], reconfigurable approach is extremely energy efficient for DSP and communication processing applications. The methodology presented in this paper is targeted to portable reconfigurable platforms having

[*] This work is sponsored in part by The Ohio Board of Regents Ph.D. Enhancement Program and The Dayton Area Graduate Studies Institute (DAGSI) and The Air Force Research Laboratory(AFRL) Research Program under contract number IF-UC-00-07.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 669–678, 2004.

batteries as their main source of power. The surface of the FPGA is divided into several virtual *Configurable Tiles* and real-time task graphs are scheduled onto these tiles. This is a unique methodology because it tries to dynamically match the number of tiles being used to the application requirements, with the goal of efficient battery usage. The results indicate that even without exploiting the traditional techniques [5], the methodology proposed is capable of delivering significant battery savings.

The rest of this paper is organized as follows: Section 2 briefly discusses the battery model used and the target architecture. Section 3 discusses the methodology and the scheduler algorithms. We discuss the experiments used to verify the effectiveness of the methodology and their results in Section 4. Finally, a real-time face recognition application implementation using the methodology proposed is discussed in Section 5.

2 Battery Model and Target Architecture

Battery Model: Several factors affect the battery behavior, including the rate of discharge, profile of the discharge currents, rest periods and temperature etc. Therefore, batteries are highly non-linear devices. *Rated Capacity* of a battery is defined as the capacity of the battery (in mAh) under a nominal constant current discharge and is reported by the manufacturer. It is observed that higher rates of discharge tend to reduce the rated capacity significantly (rate capacity effect) and reducing discharge rates between heavy discharge periods allows the battery to regain some of its lost capacity (recovery effect) [9].

Battery models are essential for studying the effects of battery-aware methodologies because in many cases, it is extremely cumbersome to validate the usefulness of a technique using a large number of actual battery-discharge experiments. Rakhmatov et al. developed a variable load analytical model, based on the laws of chemical kinetics [13], which takes into account the rate capacity effect and the recovery effect. We have chosen to use this battery model because of its high accuracy and low computational complexity. It was shown in [14] that the maximum prediction error for various constant current and variable current discharge profiles was less than 4%. The authors used an actual Li-Ion battery and a simulated battery called DUALFOIL [1] for these experiments. This battery model requires two parameters (α and β) to be estimated by conducting constant current discharge experiments [14]. Intuitively, α represents the battery capacity and β captures the non-linearities in the battery behavior; the higher the value of β the more the battery behaves like an ideal source and the effect of non-linearities is less visible. The values of α and β used in this paper are 40375 ($mAmin$) and 0.275, respectively.

Equation 1 describes the battery model.

$$\sigma = \sum_{k=0}^{n-1} I_k \left(\Delta_k + 2 \sum_{m=1}^{10} \frac{\left(e^{-\beta^2 m^2 (T - t_k - \Delta_k)} - e^{-\beta^2 m^2 (T - t_k)}\right)}{\beta^2 m^2} \right) \tag{1}$$

The value of σ gives the amount of charge lost by time T, which is the length of a current discharge profile having n distinct discharge intervals. I_k, t_k

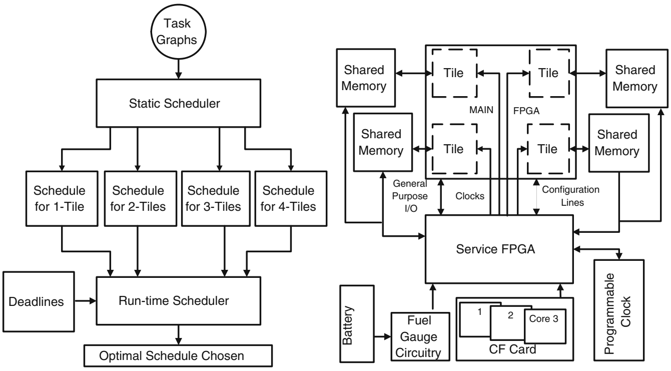


Fig. 1. (a) Methodology Overview, (b) Target Architecture

and Δ_k are the the current drawn from the battery, start time and duration of the discharge for kth interval respectively. The battery lifetime is estimated by evaluating Equation 1 for increasing values of T and stopping where $\sigma \approx \alpha$: At this point the value to T is taken as the battery lifetime. For further details please refer to [13,14].

Target Architecture: The target architecture for this methodology is shown in Figure 1-b. The *MAIN FPGA* is used to execute user tasks. The *Service FPGA* acts as the system controller and is responsible for configuring the system upon startup and bringing it to a known state. This FPGA also configures the tasks on the *MAIN FPGA* using an appropriate schedule generated by a static-scheduler (discussed in Section 3). This FPGA has access to a Compact FLASH card which stores all the bitstreams and data for various tasks to be executed. Most modern FPGAs, including Xilinx Virtex, Virtex II, Virtex II-PRO [4] and Altera Stratix and Stratix-II [2] have several useful features, such as high logic capacity, dedicated clock networks and on-chip memories, which can be used for implementing this battery efficient methodology. Various components of this methodology are the following:

1. **Configurable Tiles:** We divide the *Main FPGA* into several coarse grained, virtual, fixed sized *Configurable Tiles* (from hereon simply called: *Tiles*) as shown in Figure 1-b. No two tiles can overlap and it is assumed that the communication between two tiles is via a shared memory. However, the two major limitations on maximum number of tiles which can be implemented in an architecture are the availability of dedicated clock lines and memory bandwidth. Using several independent memory banks can improve the memory bandwidth.
2. **Clock Architecture:** Each tile has a dedicated clock line *Tile Clock*, which is a copy of a global system clock. Most modern FPGAs (Virtex-II, Startix-II) have a maximum of sixteen independent, low-skew global clock networks, which can be used as *Tile Clocks*. These tile clocks can be turned on and off to control the participation and power consumption of each tile during the

execution of a task-graph: this is also called clock gating. When a particular *Tile Clock* is turned off, the power consumption of the tile becomes negligible.

3. **Memory Subsystem:** The tiles are interfaced to fast shared local memories which are used to store application data. Memory management is performed by cooperation between the *Service FPGA* and the *Micro-tasks* (Explained in Section 3) A memory management scheme similar to [7] is used.

3 Battery-Efficient Methodology

Task Representation: We define a real-time task in the form of a Directed Acyclic Graph (DAG), simply called a "Task Graph" from hereon. In [8], authors use a set of different task-graph types for studying various scheduling algorithms for multiprocessors. The nodes in a task graph represent *Micro-tasks*. A *Micro-task* is a computational structure which is synthesized and implemented in the FPGA in the form of a partial-bitstream: It need not be acyclic in nature. In fact, all the control and loop structures are implemented within a node. If there is loop structure which cannot be implemented within a node then the loop is unrolled in space, using multiple identical nodes. The edges in a task graph convey the precedence constraints upon the nodes for execution. Each node in the task graph also contains information about the average current consumption, execution time, node type and in case of a leaf-node, a non-zero deadline.

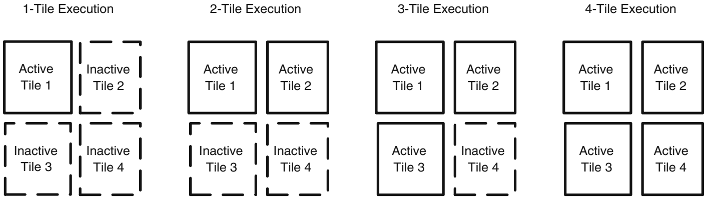


Fig. 2. Different Execution Modes

3.1 Scheduling Algorithms

The main idea of this methodology is to dynamically match the number of tiles being used to the application requirements, with the goal of efficient battery usage. For a four tile system various execution modes using different number of tiles are shown in Figure 2. For example, the 1-tile execution mode corresponds to the case where only a single tile is used to execute any particular task-graph. Alternatively, a 4-tile execution would use all four tiles available for executing a task-graph.

A top-level diagram of the methodology is shown in Figure 1-a. This methodology uses two schedulers: a *StaticScheduler* and a *RuntimeScheduler*. The *StaticScheduler* and *RuntimeScheduler* algorithms are given in Figure 3. In order to save battery-power, the *StaticScheduler* is executed at compile-time at an energy

```
$BatteryEfficientMethodology(T)$
Definitions:
$T$ = the task to be executed
$\mathcal{M}$: Total Number of tiles in the system
$\mathcal{N}$: is the maximum number of tiles to be used for scheduling
$T_d$ = deadline for $T$
$S_n = \{s_1, s_2, ..., s_n\}$: set of schedules for T corresponding to each of the $n$ tiles used for execution $(1 \leq n \leq \mathcal{M})$
$S = \{S_1, S_2, ..., S_{\mathcal{M}}\}$: set of schedule-sets $S_n$ representing each of the $\mathcal{M}$ execution-modes: 1-tile, 2-tiles etc
$E_S = \{E_{S_1}, E_{S_s}, ..., E_{S_{\mathcal{M}}}\}$: set of execution times representing each of the $\mathcal{M}$ execution-modes
$S_c$: Task $T$ chosen to be executed using $c$ tiles where $c \in \{1, 2, .., \mathcal{M}\}$
Inputs: T
Begin
for $q = 1$ to $\mathcal{M}$ do
    $S_q, E_{S_q} = StaticScheduler(T, q)$
end for
$RunTimeScheduler(S)$
End $BatteryEfficientMethodology$
```

```
$RunTimeScheduler(S)$
Inputs: $S$
Begin
$(S_c, c) = ChooseTilesToUse(S, E_S, T_d)$
Configure $T$ on the FPGA using $S_c$ schedule
Wait until $T$ has finished execution
End $RunTimeScheduler$
```

```
$ChooseTilesToUse(S, E_S, T_d)$
Inputs:$S, E_S, T_d$
Outputs:$S_c, c$
Begin
for $k = 1$ to $\mathcal{M}$ do
    if $E_{S_k} < T_d$ then
        Return$(S_k, k)$
        Break
    end if
end for
End $ChooseTilesToUse$
```

```
$StaticScheduler(T, \mathcal{N})$
Definitions:
$I_p, \Delta_p$: current consumption and execution time of the node $p$
$D$: A dummy node where $I_D = I_{sleep}$
$Start_p, End_p$: Execution start time and end time of node $p$
$\mathcal{R}$: is a set of all nodes $p$ in $T_j$ s.t. Parents$(p, T) = 0$
Inputs:$T, \mathcal{N}$
Outputs:$S_{\mathcal{N}}, E_{S_{\mathcal{N}}}$
Begin
for $i = 1$ to $\mathcal{N}$ do
    $V_i = 0$
end for
$\mathcal{R} = RootNode(T)$
$V_{old} = 0$
while $T$ is not empty do
    $E = 0$
    for $i = 1$ to $\mathcal{N}$ do
        if $R \neq NULL$ then
            Choose a node $p \in T$ s.t. $I_p = max(I_k)$ $(\forall k \in \mathcal{R})$
            $s_i = p$, $Start_p = V_{old}$, $End_p = Start_p + \Delta_p$
            if $End_p > E$ then
                $E = End_p$
            end if
            Remove $p$ from $T$
        else
            $s_i = D$, $Start_D = V_{old}$, $End_D = E$
        end if
    end for
    $V_{old} = E$
    for each node $p$ scheduled in this iteration, where $p \neq Dummy$ do
        $\mathcal{R} = u$ where $u$ is a child of $p$ and Parents $(u, T) = 0$
    end for
end while
Return$(S_{\mathcal{N}}, E_{S_{\mathcal{N}}})$
End $StaticScheduler$
```

Fig. 3. Battery-Efficient Task Scheduling Methodology

rich platform. It schedules a given task graph on different number of tiles. *StaticScheduler* also calculates the total execution time of any task graph. If T is a task to be executed on the portable RC platform then the *BatteryEfficientMethodology* finds an execution mode which would result in battery-efficient execution. If $\mathcal{M}$ is the maximum number of tiles in the system then the *StaticScheduler* produces $\mathcal{M}$ schedules of any task graph by scheduling it on $1, 2, 3 \ldots \mathcal{M}$ number of tiles respectively. A simple list based scheduling algorithm is used to implement *StaticScheduler*. It was shown in [14] that in the absence of any precedence constraints the upper bound on battery capacity used is defined by a profile which uses a strictly increasing current and the lower bound is defined by a profile using a strictly decreasing current. Therefore, in the list of nodes ready to be scheduled, the node with the highest current consumption is scheduled to be executed first. If two or more nodes are to be executed simultaneously in a time step, then the *StaticScheduler* waits until all nodes have finished execution before starting a new time step.

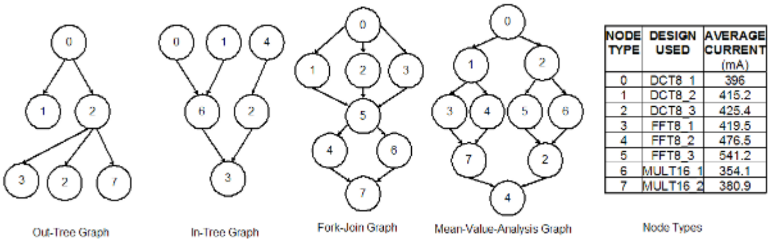
These schedules along with their execution times are used by the *RuntimeScheduler* which then selects the execution mode which uses the least amount of battery capacity and meets the deadline for the task. We calculate the battery consumption and battery lifetime by using the battery model discussed in Section 2.

Each tile reconfiguration poses a battery cost penalty as well as execution time penalty. Therefore, it is important to minimize the number of reconfigurations as much as possible. If the task execution time granularity is of the order of milli-seconds then the penalty associated with reconfiguration is extremely high: it takes 100 msec to fully reconfigure a Xilinx Virtex XCV300 FPGA [4] and we found by experiments on iPACE-V1 platform, that the associated battery cost is 0.0125 mAh. It is to be noted that the methodology presented here can be implemented in conjunction with some of the traditional techniques (Voltage and Frequency Scaling) [5] to further enhance the battery efficiency.

4 Experiments and Results

Task Graphs Used in Experiments: In [8], authors use a set of different task-graph types for studying various scheduling algorithms for multiprocessors including Out-Tree, In-Tree, Fork-Join and Mean-Value-Analysis. An illustrative example of each of these graph types is shown in Figure 4. These task-graph types represent a high-level task structure of commonly encountered parallel applications. For each type we generated five different task-graphs where the size varied from 20-100 nodes with increments of 20. The out-tree and in-tree graphs were generated using Task Graphs for Free (TGFF) [3], software. The number of out-going/in-coming edges from/to any node of the out-tree/in-tree graph was randomly varied from 2 to 7 and the number of in-coming/out-going edges was randomly varied from 2 to 4. Similarly, the fork-join and mean-value-analysis graphs were randomly created, however, TGFF was not capable of generating these graphs. In fork-join graphs the number of out-going and in-coming edges were randomly varied between 2 to 8 and for mean-value analysis these numbers were randomly chosen to be 1 or 2, respectively.

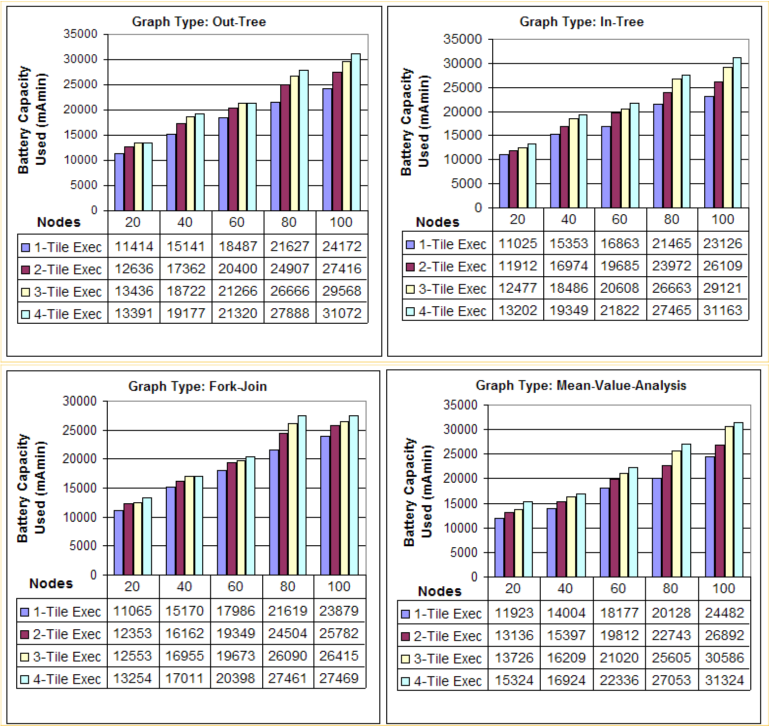
In order to test this methodology we used iPACE-V1 [6], shown in Figure 7-b. Each node in the task-graphs represented one of the seven different node types (Figure 4) which were implemented on the iPACE-V1 platform and their average current consumption was measured. The total current consumption of the system was used in profiling the current consumption of a task, which included the cumulative effect of all subsystems running on this system. The execution time for each node was chosen randomly as well as the deadline for the leaf-nodes such that it was possible to schedule the graph using at-least one of the schedules generated by the *StaticScheduler*. The results in this Section reflect an architecture which had four tiles only.



NODE TYPE	DESIGN USED	AVERAGE CURRENT (mA)
0	DCT8_1	396
1	DCT8_2	415.2
2	DCT8_3	425.4
3	FFT8_1	419.5
4	FFT8_2	476.5
5	FFT8_3	541.2
6	MULT16_1	354.1
7	MULT16_2	380.9

Fig. 4. Different Graph Types and Node Types Used

Experiment 1: Effect of the Number of Tiles Used for Scheduling Tasks: We wanted to see the effect of the number of tiles used for scheduling tasks, on the battery cost for each graph type discussed in Section 4. Therefore, we scheduled the graphs discussed in Section 4 on one, two, three and four tiles respectively. Four plots were generated and compared for each graph type and are shown in Figure 5. The battery is discharged at much higher rates when four tiles are being used instead of just one, consequently the the battery capacity used for 4-tile execution is more.



Graph Type: Out-Tree

Nodes	20	40	60	80	100
▨ 1-Tile Exec	11414	15141	18487	21627	24172
■ 2-Tile Exec	12636	17362	20400	24907	27416
□ 3-Tile Exec	13436	18722	21266	26666	29568
□ 4-Tile Exec	13391	19177	21320	27888	31072

Graph Type: In-Tree

Nodes	20	40	60	80	100
▨ 1-Tile Exec	11025	15353	16863	21465	23126
■ 2-Tile Exec	11912	16974	19685	23972	26109
□ 3-Tile Exec	12477	18486	20608	26663	29121
□ 4-Tile Exec	13202	19349	21822	27465	31163

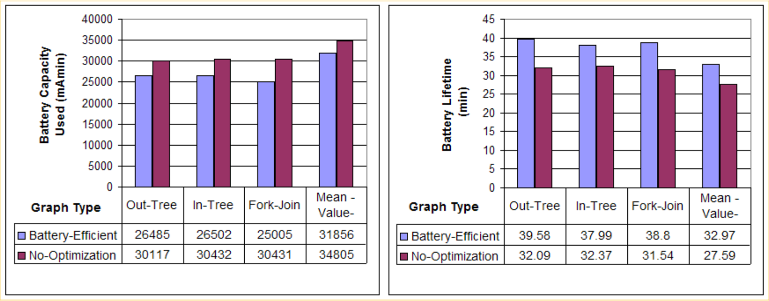
Graph Type: Fork-Join

Nodes	20	40	60	80	100
▨ 1-Tile Exec	11065	15170	17986	21619	23879
■ 2-Tile Exec	12353	16162	19349	24504	25782
□ 3-Tile Exec	12553	16955	19673	26090	26415
□ 4-Tile Exec	13254	17011	20398	27461	27469

Graph Type: Mean-Value-Analysis

Nodes	20	40	60	80	100
▨ 1-Tile Exec	11923	14004	18177	20128	24482
■ 2-Tile Exec	13136	15397	19812	22743	26892
□ 3-Tile Exec	13726	16209	21020	25605	30586
□ 4-Tile Exec	15324	16924	22336	27053	31324

Fig. 5. Effect of the Number of Tiles Used for Scheduling Tasks-Graphs



Graph Type	Out-Tree	In-Tree	Fork-Join	Mean - Value-
▫ Battery-Efficient	26485	26502	25005	31856
▪ No-Optimization	30117	30432	30431	34805

Graph Type	Out-Tree	In-Tree	Fork-Join	Mean - Value-
▫ Battery-Efficient	39.58	37.99	38.8	32.97
▪ No-Optimization	32.09	32.37	31.54	27.59

Fig. 6. (a) Battery Cost for Experiment-2, (b) Battery Life-time for Experiment-2

Experiment 2: Effect of Scheduling Multiple Real-time Tasks on the Battery Capacity and Lifetime: For each graph type discussed in Section 4 we generated 10 new graphs with 20 nodes each. All leaf-nodes were assigned a random deadline and each graph was scheduled using the *StaticScheduler* on one, two, three and four tiles respectively: The execution times along with the schedules for the four cases were then passed onto the run-time schedular. The run-time scheduler chose the schedule which matched the number of tiles being used to the application requirements, with the goal of efficient battery usage. The results for each graph type are shown in Figure 6-a, where the battery cost of the optimal execution is compared with the cost of the case where maximum surface area of the FPGA is used to ensure that the deadlines would be met. We report an average of 14.9% and as high as 21.7%, less battery capacity used, as compared to non-optimal execution.

Instead of looking at the battery capacity used, another way to characterize the effectiveness of the methodology proposed can be to find out how long the battery survives after all ten graphs are executed, for each of the four graph-types. Therefore, we discharged the battery at a constant rate of 500mA, after the last node was executed, until the battery was exhausted for both cases: optimized execution and non-optimized execution. The battery lifetime for each of the four graphs is shown in Figure 6-b. We observed an average of 20.8% and as high as 23.34% improvement in the battery lifetime when our battery-efficient methodology was used.

5 Case Study: Battery-Efficient Real-Time Face Recognition

This section provides the details and results of using the methodology proposed in this paper on a real-time implementation of the *Eigenface* algorithm [15] for face recognition using the iPACE-V1 platform. For details of the algorithm please refer to [15]. The task graph of the EigenFace algorithm is shown in Figure 7-a.

The most compute intensive part of this task graph is the multiplication of the Eigenfaces (EF) with the difference face (DF) and accumulation of the results (Nodes 1 to 4). As the number of Eigenfaces increases the accuracy of

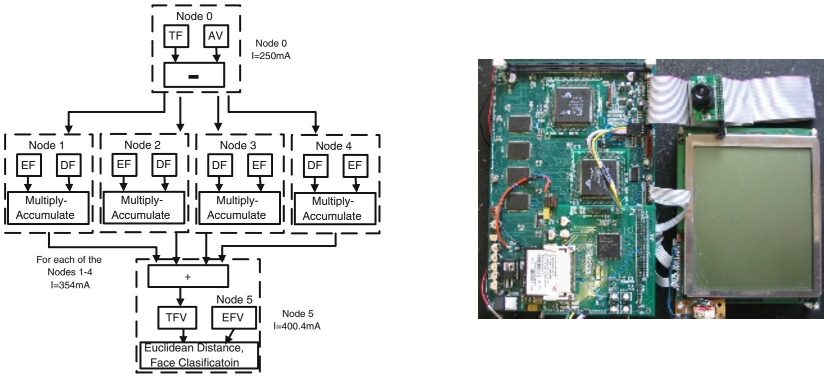


Fig. 7. (a) Eigenface Task-Graph, (b) iPACE-V1 Platform

Table 1. (a) EigenFace Algorithm run-times (b) Battery life-times Results

Max Tiles Used	10-EigenFaces Per Frame Exec. Time (msec)	20-EigenFaces Per Frame Exec. Time (msec)
1	40.12	72.00
2	23.47	36.72
3	17.92	25.62
4	15.15	20.08

	10-EigenFaces		20-EigenFaces	
Frame Rate (1/sec)	15	30	15	30
Deadline per Frame (msec)	66	33	66	33
Battery-Efficient Lifetime (minutes)	74.01	44.83	39.66	20.12
No-Optimization Lifetime (minutes)	22.32	22.32	14.64	14.64

the algorithm gets better but the execution time increases. We use two cases for executing this algorithm: using 10-Eigenfaces and 20-Eigenfaces for face recognition. Additionally we consider two frame rates for real-time execution: 15 frames per second and 30 frames per second. As the frame rate increases, the deadlines become more strict. For the system to remain real-time, the execution time of the task graph should be less than the deadline associated. The image size was chosen to be 208 × 208 pixels. The complete system was executed at 13Mhz.

Due to the *MAIN FPGA* size limitations, we mapped Nodes 0 and 5 to the *Service FPGA* and Nodes 1 through 4 were mapped to the *MAIN FPGA*: these were the nodes which occupied the four tiles in the main FPGA. Both FPGAs were configured once at the startup: The execution and participation of each node was coordinated by dynamic clock gating as explained in Section 3. The task graph shown in Figure 7-a was scheduled on one, two, three and four tiles respectively, with 10 and 20-Eigenfaces used for face recognition: Table 1-a shows the execution times for these two cases. When the frame rate was chosen to be 15 frames per second (fps) the deadline for execution of the task-graph was 66.7 msec and when the frame rate was 30 fps the deadline was 33.3 msec. As shown in Table 1-a for 10-Eigenface case the schedule using a single tile can meet the deadline for 15 fps execution. However, when the frame rate is increased to 30 fps, one has to use 2-tiles or more to meet the deadline. For 20-Eigenface case, even the deadline for 15 fps cannot be met by the schedule using one tile: Therefore, for 15 fps we use 2-tile schedule and for 30 fps we use 3-tile schedule for battery-efficient execution.

Table 1-b shows the battery life-time observed when 10 and 20-Eigenface cases were executed at 15 fps and 30 fps frame rates, respectively. The task graph was executed periodically with period equal to the frame rate, until the battery was exhausted. The battery life-time improvement of 1.3 to 3.3 times are reported as compared to the execution where no-optimization steps were taken and the task graph was executed using the maximum FPGA area.

References

1. P. Arora, M. Doyle, A. Gozdz, R. White, and J. Newman. Comparison between Computer Simulations and Experimental Data for High-Rate Discharges of Plastic Lithium-Ion Batteries. In *J. Power Sources*, page 88, 2000.
2. Altera Corporation. Stratix II Device Handbook, Vol 1 and 2, 2004.
3. R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *Proceedings of the 6th international workshop on HWSW codesign*, pages 97–101, 1998.
4. Xilinx Inc. Virtex-II Platform FPGA Handbook, Virtex-II-PRO Platform FPGA Handbook. www.xilinx.com, 2003.
5. Debashis Panigrahi Kanishka Lahiri, Sujit Dey and Anand Raghunathan. Battery-Driven System Design: A New Frontier in Low Power Design. In *ASP-DAC/VLSI Design 2002, Bangalore, India*, pages 261–268, January 2002.
6. Jawad Khan, Manish Handa, and Ranga Vemuri. iPACE-V1: A Portable Adaptive Computing Engine for Real Time Applications. In *LNCS2438 Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, pages 69–78, Berlin Heidelberg, 2002. Springer-Verlag.
7. Jawad Khan, Balasubramanian Sethuraman, and Ranga Vemuri. A Power-Performance Tradeoff Methodology for Portable Reconfigurable Platforms. In *ERSA'04: The International Conference on Engineering of Reconfigurable Systems and Algorithms, To Appear*. C.S.R.E.A. Press, June 2004.
8. Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. In *IEEE Transactions on Parallel Distributed Systems 7(5)*, pages 506–521, 1996.
9. D. Linden and T. B. Reddy. *Handbook of Batteries*. McGraw Hill, NY, 2002.
10. Jiong Luo and Niraj K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Design Automation Conference*, pages 444–449, 2001.
11. T. L. Martin. *Balancing Batteries, Power, Performance: System Issues in CPU Speed-Setting for Mobile Computing*. PhD thesis, Carnegie Mellon University, 1999.
12. Jan M. Rabaey. Reconfigurable computing: The solution to low power programmable dsp. In *Proceedings 1997 ICASSP Conference, Munich*, 1997.
13. D. Rakhmatov and S. Vrudhula. An analytical high-level battery model for use in energy management of portable electronic systems. In *ACM/IEEE International Conference on Computer Aided Design: ICCAD*, pages 488–493, 2001.
14. D. Rakhmatov and S. Vrudhula. Energy management for battery-powered embedded systems. In *ACM Transactions on Embedded Computing Systems, Volume 2, Number 3*, pages 277–324, 2003.
15. M. A. Turk and A. P. Pentland. Eigenfaces for recognition. In *Journal of Cognitive Neuroscience,3(1)*, pages 71–96, 1991.

HW/SW Co-design by Automatic Embedding of Complex IP Cores

Holger Lange and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 23,
D-38106 Braunschweig, Germany
{lange, koch}@eis.cs.tu-bs.de

Abstract. Complex SoC and platform-based designs require integration of configurable IP cores from multiple sources. Even automatic compilation flows from a high-level description to HW/SW systems can benefit from having access to reusable sophisticated hand-optimized IP blocks. This is especially the case in the domain of reconfigurable computers, which offer core integration directly into the custom datapath. This work proposes the Parametric C Interface For IP Cores (PaCIFIC) to allow the automatic embedding of complex IP cores in a high-level language such as C. PaCIFIC provides for formal description of IP behavior and interface characteristics as well as an idiomatic programming style natural for SW developers.

1 Introduction

In many current design styles such as systems-on-chip (SoCs), embedded systems and platform-based techniques involving hardware-software (HW-SW) co-design, a gap appears in the design flow at the interface between HW and SW. The individual HW and SW sub-flows (from RTL to layout and SW to binary code) themselves are quite mature with regard to tool support, but the interface between both requires significant manual effort to establish [1][10]. This applies even more strongly if the HW contains IP cores, as these often feature complex functionality and interfaces. The challenges the designer has to cope with include large system-specific parameter sets that span a huge space of possible combinations, not all of them legal. While configuration management is already well-explored [7][8][9], this paper concentrates on HW/SW interface design. With the gate capacity of configurable devices reaching into the millions by now, these issues are also becoming applicable to target platforms such as Adaptive Computer Systems (ACS). On the other hand, their configurability allows a much tighter integration of IP blocks into the system at the datapath level than the comparatively coarse-grained on-chip buses used in the ASIC world.

Two aspects play key roles in this kind of interface design. First, the interface functionality itself has to be partitioned between HW and SW realizations. Second, concrete interface mechanisms and protocols must be determined (e.g., physical connections, address ranges, transfer modes, device drivers, etc.). Both of these issues require the designer to explore a large design space, a time consuming and sometimes tedious task despite initial efforts at tool support [3].

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 679–689, 2004.

This work focuses on the latter aspect in the context of using an ANSI C language description to tightly embed, compose and interact with IP cores. This language has been chosen in context of our work of enabling fully automatic compilation from a high-level programming language to ACS applications. As a solution, we propose the Parametric C Interface For IP Cores (PaCIFIC). It establishes an automatic design flow presenting convenient, simple C interfaces (function prototypes) to a SW developer. Our approach hides the formal descriptions of IP- or platform behaviors and interface characteristics by encapsulating them together with other IP configuration data in a dedicated repository [9].

2 Related Work

Tomiyama et. al. [2] compare several Architecture Description Languages (ADL) and determine the characterizing properties to be behavior- and structure description. They demand an explicit behavior description of processors for better compiler generation. However, they consider synthesis-based ADLs or HW Description Languages (HDL) neither sufficiently easy-to-use nor flexible enough for this task. Balboa [3] is a HW/SW co-design framework for system models. It abstracts IP interfaces in a twofold intermediate layer consisting of a Component Integration Language (CIL) and the Balboa Interface Description Language (BIDL) providing automatic data type matching and interface generation. The IP behavior is implemented as C++ models. The CoWare N2C suite [10] contains a set of interface behavior descriptions expressed as prototypes or templates specialized in many detailed descendants. Despite their great number, the behavior descriptions are not universal and cannot replace a behavior description language. Handel-C [11] is an extension to the C language with explicit parallelism, HW data types and inter-thread communication channels based on the model of Communicating Sequential Processes (CSP) [13]. SystemC and Synopsys Behavioral Compiler both provide abstract interface modeling, but the HW-oriented constructs in SystemC have not been embraced by the SW-development community. Traditional HDLs, e.g., as used in the Behavioral Compiler, have gained even less ground there. Carloni et. al. [17] construct an interface mechanism based on latency insensitive protocols. Thronicke [7] and Zeller [8] present configuration management (CM) methods from HW and SW domains. At present, there seem to be no attempts to combine CM and ADLs, although this would appear advantageous when building systems of complex IP cores and SW.

3 Problem Description

Consider a scenario with two IP cores which should be arranged forming a pipeline. Assume that each core has one input and one output interface.

As shown in Fig. 1, the data path comprising both cores is supposed to be used from a SW description that also sources and sinks the data. A natural approach for plain SW would consider the two IP cores to be C functions, leading to the code in Fig. 2.

From such a code description, the HW pipeline shown in Fig. 1 should be automatically inferred. This requires additional information about the HW “functions" `compress` and `crypt`. The SW developer should not have to be aware of the actual mechanisms

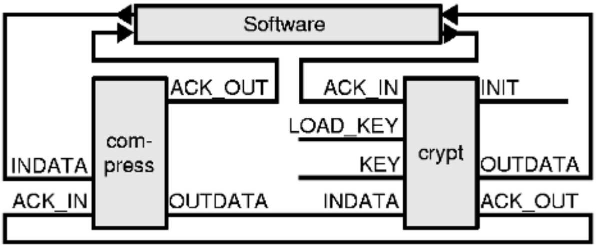


Fig. 1. Hardware pipeline used by software

```
int *indata, *outdata, *intermediate;
for (n = 0; n < 64; n++) {
    compress (indata++, intermediate);
    crypt (intermediate++, outdata++);
}
```

Fig. 2. IP cores as C functions

involved in realizing the structure. To this end, several issues must be addressed when dealing with HW embedded in a SW description:

- Recognition of IP cores
  Since C cannot distinguish between HW and SW, function calls aiming at IP core instances have to be detected somehow.
- Low-level interface control
  In contrast to HDLs, plain C has no notion of timing- or cycle-accurate execution schedules. Thus, for each IP core, interface parameters like signal timing, handshaking and bus arbitration must be provided in an external representation.
- Data transfer
  There are several ways to exchange data between SW and HW. IP cores are often programmed via register files. Thus, a Programmed I/O (PIO) mode is mandatory in this case. On the other hand, this is highly inefficient for the large data sets which are commonly processed by complex IP cores (video, networking). In these cases, Streaming I/O (SIO) mechanisms are generally employed, often assisted by rate matching and buffering using FIFOs. We will refer to such a setup as a stream engine. For each use of an IP core, the appropriate transfer method used has to be determined based upon data-traffic statistics and interface descriptions delivered by the IP provider.
- HW events
  Some transactions are initiated not by the SW, but by the IP core, e.g., the acceptance to process the next data block. Asynchronous events such as interrupts or error notifications are beyond the semantics of a C function. The functional synchronization, such as the indication of the current state of a HW function, must be realized, for example, to determine the end of a C function call (=IP core execution) and proceed with the rest of the program.

4 Proposed Solution

PaCIFIC consists of rules for an idiomatic programming style which must be used when embedding IP cores in a C source program, and interface control semantics which describe the interface behavior of an IP core (see Fig. 3). To this end, PaCIFIC includes a data model and a human-readable description language for the characteristics of individual IP blocks as well as entire platforms (not shown here). All components are realized as compiler passes that perform the necessary analysis and synthesis steps for both HW and SW. These steps access the PaCIFIC descriptions to discover idiomatic HW function calls in the C source program. As first practical realization, the Compiler for Adaptive Systems (COMRADE) [4][5] will act as the host compiler. PaCIFIC enables COMRADE to access and integrate IP cores too complex to be generated efficiently just from a SW description. For brevity, the details of the COMRADE integration will not be discussed here.

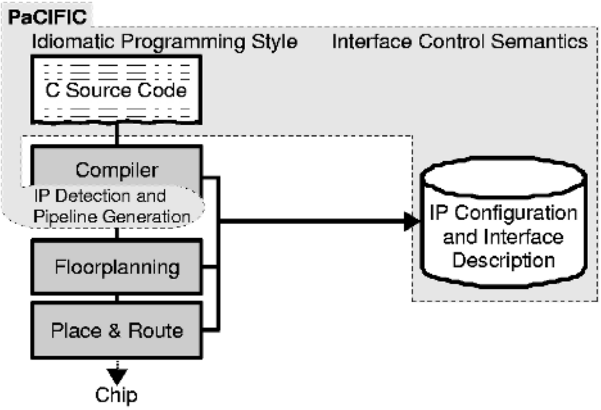

PaCIFIC
Idiomatic Programming Style
Interface Control Semantics
C Source Code
Compiler
IP Detection and Pipeline Generation
Floorplanning
Place & Route
Chip
IP Configuration and Interface Description

Fig. 3. Design flow with PaCIFIC

The data models and representations are based on the study of more than thirty commercial IP blocks that were classified using the attributes of the PaCIFIC interface template [12]. The aim was the capability to describe all of the IP cores' interface semantics with the existing attribute catalog. The majority of the evaluated cores belongs to the domains of multimedia and networking. The first cores generally presented a data path oriented interface, with the video or audio stream processing being the main task. In contrast, the networking IP cores employed a processor-based register interface. More complex IP blocks even use multiple different interfaces of both kinds.

5 Hardware Interface Description

The PaCIFIC interface description [12] is used to define the static properties for all IP interfaces as well as the dynamic flow of the interface protocols based on synchronous logic operating without a central flow control authority. As usual, properties are expressed as attributes and values. Some of the many defined attributes are:

- Identification (class, type, version, name).
- Auxiliary information (author, comments).
- Port definitions (transaction type, direction, width, associated clock, abstract data type, associated address, handshaking protocol, data traffic statistics, bus arbitration).
- External resources required by the IP core and their allocation modes (shared, exclusive, persistent). This might include external memories or special I/O requirements (e.g., access to multi-Gbps transceivers).

Port transaction types and handshaking protocols will be examined in more detail in Section 5.1. A fragment of an interface template for the `crypt IP` core is shown in Fig. 4 below:

```
interface crypt
  type: custom
  version: 1
  port INDATA
    transaction: data
    direction:   in
    width:       32
    sequence repeat: inf
      bigendian: 32 bit signed
    end sequence
    enableout: name ACK_OUT offset 0 latency 0
  end port
  port ...
...
end interface
```

Fig. 4. Part of PaCIFIC Interface for the crypt IP

The abstract data type defined by the `sequence` block of the example (here just a single scalar integer), plays a central role in the data exchange between SW and HW. It arranges the nature, order, and count (`repeat`) of the data items that are transferred over the port or bus. Every sequence block corresponds to formal parameter of the fictitious C function representing the IP core.

Interface *templates* can be used to group and reuse the same or similar interfaces in a fashion analogous to the classes and inheritance of object-oriented programming.

5.1 Interface Protocol Description

The fundamental interface flow control mechanism in PaCIFIC is a handshaking scheme which consists of an incoming and an outgoing signal per port. For an outgoing signal, the asserted state (selectable as high or low) means that the IP block is ready to consume data (on an input port) or that data is waiting to be fetched (on an output port). The incoming signal is the outgoing signal from the connected port at the other end of the communication. It is not necessary to specify both signals, a one-way handshake is possible as well as no handshake. A transaction is considered complete when all specified handshake signals are active at a clock edge. If both signals are specified, it is illegal to reset the first active signal before the second signal has been activated. For all handshakes, a time offset or initial latency with regard to another handshake may be specified. Additionally, an interrupt semantic can be selected for the handshaking signals. This is useful if no actual data transfer is required during the transaction, e.g., to indicate that data must be fetched from a mailbox register.

When such static interface properties no longer suffice to describe the characteristics of an IP core's interface, an enhanced version of the FLAME UCODE notation is employed [6] to describe dynamic behavior. A UCODE block is a list of statements most of which are executed sequentially. It represents the state machine of an interface controller. An excerpt of the UCODE statements is shown below:

- The `level` statement asynchronously sets ports to the values given as arguments of the form `port=value`.
- The `posedge` statement is similar to level, but operates synchronously with a rising clock edge.
- The continue statement takes three kinds of parameters: an optional `timeout: n`, optional `error: port=value` expressions, and normal `port=value` expressions which are interpreted as conditions. The first two branch to the `exception` block either if all error conditions are true or the timeout in clock cycles has expired. If no timeout or error occurs, the control flow is halted until all normal continue conditions are valid. As stated in [6], multiple conditions in the same continue statement are logically ANDed, multiple successive `continue` statements are ORed. The asynchronous `continue` statement can be synchronized by a following `posedge`.
- The `exception` block, if present, is located at the end of the UCODE block. It marks the branch target for all `error` and `timeout` clauses and puts the interface or IP block into a well defined error state. The normal control flow terminates, if the `exception` block or the end of the UCODE block is reached.
- The mandatory `transfer n name` block represents the transfer of n sequences to port name, with the nature of the sequence being defined in the PaCIFIC interface description. Without a sequence description on a port, `transfer` indicates n scalar transfers using the full port width. It acts as a loop in the UCODE control flow. Each iteration is triggered by the handshaking protocol defined for the port.

6 Software Interface Description

The last sections dealt with the HW realization of the interface to the IP cores. In this section, the corresponding SW mechanisms will be examined.

From the study of multimedia IP cores it is obvious, that a powerful data streaming service is needed to source and sink the data path interfaces of the IP. The stream engine fetches and stores data from respectively to shared memory, which is accessible to the

SW running on the CPU. The start address of the memory range to be streamed can be expressed as a pointer to C structures reflecting the composition of the sequences defined in a PaCIFIC interface.

In all cases, the IP cores also require programming (e.g., for initialization) using a register interface. This can be realized by simply mapping the registers into a SW accessible memory region (but not necessarily the main memory space).

To recognize the actual IP core embedding, and establish both communication methods, an idiomatic C programming style is required: Only two modes of instantiating IP cores from C are supported by PaCIFIC, but they are sufficient to cover all interface types under discussion.

First, there is the fully automatic interface generation, which results in the creation of read and write *primitives* for access to the ports of the IP core in both direct (register) and streaming fashions. This method works from the PaCIFIC interface definition, the IP designer (or more precisely, the author of the PaCIFIC description) does not have to provide any additional data. However, the SW has to explicitly call the primitives in the required order to actually get the IP core to perform the desired function.

Second, there are functions which atomically perform complex operations without requiring incremental prodding by a SW program. For the realization of these *monoliths*, the IP designer has to supply an algorithmic description of the control and data patterns that must be applied to the interfaces of an IP core for the required function. The monoliths are then generated automatically and their call resembles conventional C library functions (all individual control steps have been hidden and implemented automatically).

Note that threading models such as the POSIX one are compatible with PaCIFIC, enabling the parallel execution of HW and SW. This can be beneficial when calling data-intensive IP cores: E.g., while the HW is still running, the SW prefetches the next data block into memory and writes processed data to disk.

6.1 Primitives

Consider an input port indata without an associated address that is 32 bits wide (cf. example in Section 5). For this case, the C function `write_indata` is generated. It writes 32-bit integers (sequence `bigendian: 32 bit signed`) and terminates data dependently (`repeat: inf`):

```
void write_indata (int *data);
```

A best match approach is employed for mapping scalar hardware data to C data types. The *n* least significant bits of the next larger C type represent a hardware scalar of width n. Unrelated to the previous example, an output port with an associated address that delivers a sequence of composite data items (here mapped to the `struct comp`) results in the following function:

```
void read (int address, struct comp *data);
```

If the `repeat` value in a sequence definition equals one, SW wrapper functions may be used to eliminate the unwieldy pointer in favor of just passing scalar data. Primitives are most suitable for use with simple register- or memory-style interfaces.

6.2 Monoliths

Due to the strictly sequential semantics of C, it is not possible to directly describe pipelined accesses using primitives. However, this is achievable using monoliths. The example in Fig. 5 below reconsiders the compress-crypt scenario from Section 3 and describes the underlying control protocol for the behavior “encrypt” in PaCIFICextended UCODE [6] (see also Section 5.1).

```
behavior encrypt
proc crypt(plaintext, ciphertext)

; load key
posedge LOAD_KEY=1
        KEY=10027821 ; fixed key
posedge LOAD_KEY=0
; process single data item
transfer 1 INDATA
level      INDATA=plaintext
           ACK_IN=1
continue timeout: 16
         error: INIT=0 ACK_OUT=1
posedge  ciphertext=OUTDATA
level    ACK_IN=0
endtransfer INDATA

; wait for end of pipeline flush
exception
continue INIT=1
; execution terminates here
end behavior
```

Fig. 5. UCODE for behavior “encrypt"

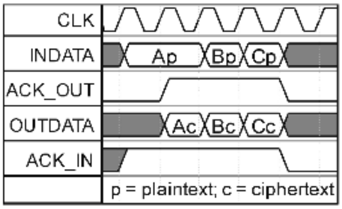


Fig. 6. Signal timing for the crypt IP

The function prototype in the `proc` statement corresponds to the C function, with variables being passed by reference. Fig. 6 displays the signal timing described by the `transfer` block above with `INIT := 1`.

The following sequence is defined in the PaCIFIC specification for the port `INDATA`:

```
; data
sequence
  bigendian 32 bit signed
endsequence
```

From this description, the `crypt` function in the example of Section 3 can be generated. The function terminates after encrypting one data word from the memory pointed to by `plaintext` and delivering it to `*ciphertext`:

```
crypt(int *plaintext, int *ciphertext)
```

7 Hardware Experimental Results

To evaluate the feasibility and efficiency of the PaCIFIC approach, the Xilinx High-Performance 16-Point Complex FFT/IFFT [14] of the Core Generator suite was coupled to an ANSI C program applying the PaCIFIC algorithms manually, since an automatic tool flow is not yet available. The FFT expects data to be continuously streamed to its input buses as well as from its outputs. For simplicity, the 16 bit real and imaginary buses are combined into 32 bit buses carrying complex numbers. The output data is available after an initial latency of 82 cycles. To efficiently source and sink data, two stream engines are employed with a FIFO capacity of 256x32 bit each. The test platform was an ACE-V ACS [16]. The relevant platform HW used here includes a 100MHz microSPARC IIep CPU with 64 MB of DRAM and a Virtex 1000 -4 FPGA. The CPU accesses the FPGA via PCI and a PLX PCI9080 local bus bridge. For comparison, the FFT was also exercised on a second test platform, an ADM-XRC card attached via PCI to a standard PC (AMD Duron 800 MHz, 256 MB SDRAM). The ADM-XRC is a subset of the ACE-V providing the same Virtex FPGA and PLX local bus bridge.

The C program executed by the processor reads the source data from a file into the DRAM, calls the FFT HW implemented on the FPGA and finally writes the result back to disk (Fig. 7).

```
int main(int argc, char* argv[]) {
  FILE* infile, * outfile;
  int* dram_in, * dram_out;
  infile = fopen("time.dat", "r");
  outfile = fopen("freqspec.dat", "w");
  dram_in = calloc(16384, sizeof(int));
  dram_out = calloc(16384, sizeof(int));
  fread (dram_in, sizeof(int), 16384, infile);
  vfft16(dram_in, dram_out); /*** HW function call ***/
  fwrite(dram_out, sizeof(int), 16384, outfile);
...
}
```

Fig. 7. C program calling FFT HW

Table 1. FFT mapping results (a) and performance results (b) with PaCIFIC

a)	Area Slices	Total V1000	BSR*	Total V1000
FFT	1386	11%	0	0%
S/I*	1385	11%	4	13%
Sum	2771	22%	4	13%

*S/I: Stream engines and interface control; BSR: BlockSelectRam

b)	Clk cycles ACE-V	Clk cycles ADM-XRC/PC
S/I* read startup latency	8	8
PCI read startup latency	39	29
FFT processing	4178	4178
Memory transfer overh.	4096	4096
PCI processing overh.	8036	6333
PCI write flush overh.	199	58
Sum	16556	14702

Data for the FFT logic is sourced and sunk by two stream engines co-located on the FPGA which access the DRAM in bus master mode. The naive approach without PaCIFIC would require a manual set-up of the stream engines and the control signals for the FFT. Instead, all of this is wrapped by PaCIFIC into a single function call. After application of the PaCIFIC algorithms, the SW part was compiled using gcc, while the resulting RTL description for the stream engines and interface control logic was synthesized with Synplify 7.3.3. It was subsequently mapped with ISE 6.2.01i, embedding the FFT core netlist. The achievable clock speed without optimized floorplanning for the mapping results in Table 1a is 27 MHz.

Table 1b shows the performance results for the FFT processing 4096 words on both ACE-V and ADM-XRC/PC at 27 MHz FPGA clock. The time spent in SW processing is not considered here since it depends mostly on the host's file I/O capabilities rather than PaCIFIC interface design assuming that a naive approach would also access memory in master mode.

8 Conclusions and Future Work

We presented PaCIFIC, a strategy for using complex IP cores from within ANSI C programs as seamlessly as pure C SW functions. The HW-specifics unfamiliar to a SW developer are encapsulated in the PaCIFIC framework. Instead, the IP provider supplies the details required for core integration as a machine-readable description. The HW/SW interfaces are then generated automatically, thus raising design productivity by closing the gap between the vertical HW and SW design flows. This approach applies not only to COMRADE or the specific domain of adaptive computing systems, but generally to all HW/SW co-design environments. The unified notation for IP configuration and interface protocol description enables (semi-) automatic design composition. Reusable interface descriptions allow the separation of interfaces and implementation details. Although many of the underlying concepts have already been explored separately, it is their combination that catalyzes a new and easy-to-use HW/SW co-design flow. Additionally, reconfigurable platforms such as ACSs profit from PaCIFIC's ability to generate lightweight native interfaces at the datapath-level between IP and the rest of the system. This avoids the overhead incurred by requiring on-chip bus-compatible wrappers between the generic HW blocks. Future work will provide the tool support for PaCIFIC within COMRADE.

References

1. P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau. EXPRESSION: An ADL for System Level Design Exploration. Technical Report, University of California, Irvine, USA, 1998
2. H. Tomiyama, P. Grun, A. Halambi, N. Dutt, A. Nicolau. Architecture Description Languages for Systems-on-Chip Design. 6th Asia Pacific Conference on Chip Design Language, Fukuoka, Japan, 1999
3. F. Doucet, M. Otsuka, S. Shukla, R. Gupta. An Environment for Dynamic Component Composition for Efficient Co-Design. Proc. Design Automation and Test in Europe, 2002
4. N. Kasprzyk, A. Koch, U. Golze, M. Rock. An Improved Intermediate Representation for Datapath Generation. International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, 2003

5. N. Kasprzyk, A. Koch. Advances in Compiler Construction for Adaptive Computers. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, 2001
6. A. Koch. FLAME: A Flexible API for Module-based Environments – User's Guide and Manual. TU Braunschweig (E.I.S.), Braunschweig, Germany, 2003
7. W. Thronicke. Konzept und Realisierung einer allgemeinen Parametrisierungsstrategie von Systemmodellen unter besonderer Berücksichtigung der Wiederverwendbarkeit. PhD thesis, University Paderborn, Germany, 2000
8. A. Zeller. Configuration Management with Version Sets. PhD thesis, TU Braunschweig, Germany, 1997 (http://www.infosun.fmi.uni-passau.de/st/papers/zeller-phd/)
9. H. Lange, M. Radetzki. IP Configuration Management with Abstract Parameterizations. Proc. International Workshop on IP Based SoC Design, Grenoble, France, 2002
10. CoWare Inc. N2C Scenario Library, 2001
11. Celoxica Ltd. Handel-C Language Reference Manual, 2001
12. H. Lange. PaCIFIC. Technical Report, TU Braunschweig (E.I.S.), Germany, 2003
13. T. Hoare. Communicating Sequential Processes. Prentice Hall International Series in Computer Science, 1985
14. Xilinx Inc. High-Performance 16-Point Complex FFT/IFFT V1.0 Product Specification, http://www.xilinx.com
15. A. Koch. A Comprehensive Prototyping Platform for Hardware-Software Codesign. Workshop on Rapid Systems Prototyping, Paris, France, 2000
16. L. Carloni, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. International Conference on Computer-Aided Design, San Jose, USA, 1999

Increasing Pipelined IP Core Utilization in Process Networks Using Exploration

Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere

Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science (LIACS),
Leiden University, The Netherlands
{claus,kienhuis,edd}@liacs.nl

Abstract. At Leiden Embedded Research Center, we are building a tool chain called *Compaan/Laura* that allows us to do fast mapping of applications written in Matlab onto reconfigurable platforms, such as FPGAs, using IP cores to implement the data-path of the applications. A particular characteristic of the derived networks is the existence of selfloops. These selfloops have a large impact on the utilization of IP cores in the final hardware implementation of a Process Network (PN), especially if the IP cores are deeply pipelined. In this paper, we present an exploration methodology that uses feedback provided by the Laura tool to increase the utilization of IP cores embedded in our PN. Using this exploration, we go from 60MFlops to 1,7GFlops for the QR algorithm using the same number of resources except for memory.

1 Introduction

To better exploit the reconfigurable hardware devices that are coming to market, a number of companies like AccelChip and Celoxica and research groups around the world [5, 1] are developing new design methodologies to make the use of these devices more ubiquitous by easing the way these devices are to be programmed. At Leiden Embedded Research Center, we are developing a design methodology that allows us to do fast mapping of applications (DSP, imaging, or multi-media) written in Matlab onto reconfigurable devices. This design methodology is implemented into a tool chain that we call *Compaan/Laura* [10]. The Compaan tool analyzes the Matlab application and derives automatically a parallel representation, expressed as a Process Network (PN). A PN consists of concurrent processes that are interconnected via asynchronous FIFOs. The control of the input Matlab program is distributed over the process and the memory is distributed over the FIFOs. Next, the Laura tool synthesizes a network of hardware processors from the PN. Laura derives automatically the VHDL communication structure of the processor network as well as the control logic (interfaces) of the processors needed to attach them to the communication structure. The computation that has to be performed in every processor is not synthesized by the Laura tool. Instead, the tool integrates commercial IP cores in every processor to realize the complete computation.

When pipelined IP cores are used in a PN processed by Laura, new data can be read without waiting for the finalization of the current computation. Therefore, a network is seen as a big pipelined system. To maximize the throughput of a network, we need to

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 690–699, 2004.

carefully consider how to couple the pipelines of the processors in a network. In this coupling, one particular characteristic plays an important role. This is the existence of selfloops. A *selfloop* is a communication channel that sends data produced by a processor to itself. Selfloops appear very frequently in PNs and they impact the utilization of IP cores thus have a great impact on the overall performance of PNs. A pipelined IP core works at maximum throughput if the network provides as many independent operations as the depth of the pipeline.

In this paper, we present how we can perform a design space exploration to increase the utilization of IP cores. Using exploration, we change in a systematic way the size of the selfloops as they are a measure of independent operations. This measure is obtained at compile-time by extending Laura with a *Profiler* option. This provides systematic hints for further exploration. Using this profiler, we performed an exploration that is presented at the end of this paper. In this exploration, we realizes a throughput improvement from 60MFlops to 1,7GFlops for a QR algorithm, using the same number of processors.

2 Related Work

There are numerous researchers that recognized the need for fast performance estimation to guide the compilation of a high-level application for deriving alternative designs. The PICO project [5] uses estimations based on the scheduling of the iterations of the nested loop to determine which loop transformation lead to shorter scheduling time. The MILAN project [1] provides a design space exploration and simulation environment for System-on-Chip architectures. MILAN uses simulation techniques to derive estimates used in the evaluation of a given application. In [3] the authors use an analytical performance model to determine the best mapping for their processor array. An analytical model is used also in [8] to derive performance and area for a given nested-loop program. Adding more independent streams to increase the efficiency of a design is not new for hardware designers [7]. All these projects are focused on synchronous systems, and, therefore, a global schedule of their system can be derived at compile time. However, we derive estimations regarding the throughput of deeply pipelined IP cores in the absence of a global network schedule. In [9], we have shown already that we can apply loop transformations to increase the performance of our networks.

3 The Compaan/Laura Tool Chain

In general, specifying an application as a PN is a difficult task. Therefore, we use our *Compaan* Compiler [6] that fully automates the transformation of a Matlab code into PNs. Subsequently, the *Laura* [12] tool takes as its input this PN specification and generates synthesizable VHDL code that targets a specific FPGA platform. The Compaan and Laura tools together, realize a fully automated design flow that maps sequential algorithms written in subset of Matlab onto reconfigurable platforms. This design flow is shown in Figure 1.

In the first part of the design flow, an application specification is given in Matlab to Compaan to be compiled to a PN representation. The applications Compaan can handle are parameterized static affine nested loop programs, which can be described using a subset of the Matlab language. In the second part of the design flow, Laura transforms a PN specification together with predefined IP cores into synthesizable VHDL code. The IP cores are needed as they implement the functionality of the functions calls used in the original Matlab program. In the third part of the design flow, the generated VHDL code is processed by commercial tools to obtain quantitative results. These results can be interpreted by designers, leading to new design decisions. This is done with the help of Mattransform tool [9] that performs a source-to-source transformation on the Matlab code. By rewriting the Matlab code, we can explore different mappings. When an obtained algorithm instance meets the requirements of the designer, the corresponding VHDL output is synthesized by a commercial tool and mapped onto an FPGA platform. Recently, we have extended Laura with a 'Profiler' option to provide hints at compile time that can be used by Mattransform to rewrite the application to increase the performance of the input algorithm.

To show the relation between a function call in Matlab program and an IP core, we need to explain the way Laura realizes the functionality of the function call. This functionality is wrapped and executed in a corresponding processor of the derived PN. To implement the PN in hardware, Laura uses the notion of Virtual Processors and hardware communications channels (i.e. FIFOs).

A Virtual Processor is composed of four units: a *Read* unit, a *Write* unit, an *Execute* unit, and a *Controller unit*, as shown in Figure 2. The Execute unit is the computational part of a virtual processor. The Read unit is responsible for assigning all the input arguments of the Execute unit with valid data. The Write unit is responsible for distributing the results of the Execute unit to the relevant processors in the network. The Controller of the Virtual Processor synchronizes all the units of the processor. The Read unit and the Write unit can block the next execution when a blocking-read or a blocking-write situation occurs, thereby stalling the complete processor. A blocking-read situation occurs when data is not available at a given input port. A blocking-write situation occurs when data cannot be written to a particular output port. In the Execute unit an IP core

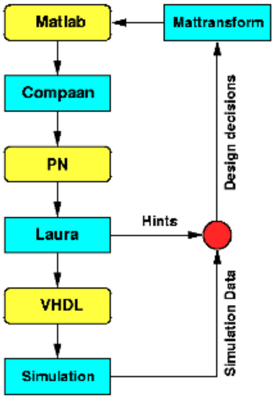


Fig. 1. The Compaan/Laura tool chain

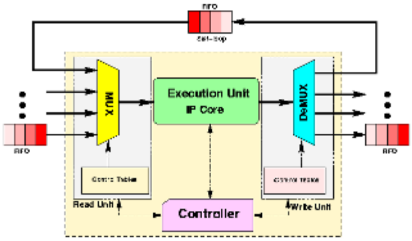


Fig. 2. The Virtual Processor model

is embedded. This IP core implements the functionality specified in the original Matlab code. The controller automatically fires the execution unit when data is available.

4 Problem Definition

The task to interpret the quantitative results obtained from VHDL traces can be very difficult in a complex PN network. Therefore, we want to guide the designer using the tool chain with indications that are obtained from the analysis of the network and prior knowledge on the IP cores.

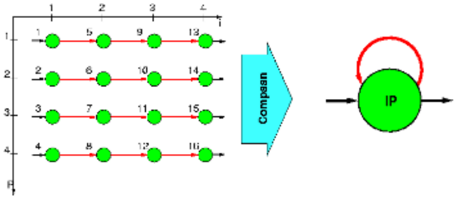


Fig. 3. Simple Example

Laura's Virtual Processor is a pipelined processor. Therefore, the filling and flushing of the pipeline reduces the throughput below the maximum level achievable. One of the problems that affect the efficiency of our designs is given by the data availability. Data availability is affected when selfloops are involved. When a processor has a selfloop, it can happen that the processor want to read data from the selfloop, but that data is still in the pipeline. Because the data is in the pipeline, the processor has to wait, reducing the efficiency of the pipeline of the IP core.

Let us consider a simple example by looking at the dependency graph (DG) in Figure 3. This DG is folded by Compaan into a single processor that has a single self-loop, which corresponds to data dependencies in the i direction. In Compaan, the nodes in the DG are scheduled. In Figure 3, the schedule is given by the p direction (e.g. 1,2,3,4,..,14,15,16). It is obvious that the operations 1,2,3,4 are independent, i.e. there is no arrow between them and, therefore, no data dependency. Therefore, the processor has to store the result of these operations into a temporary memory until they are consumed by the following 4 independent operations (e.g. 5,6,7,8). In our case, this memory is a selfloop, implemented as a FIFO channel. Hence, the necessary size for the selfloop is given by the number of independent operations, which is 4 in the case given in Figure 3. If we consider an alternative schedule given by the i direction (e.g. 1,5,9,13,..,12,16) the size of the selfloop FIFO is one. In this case, the data is immediately consumed by the processor after being produced.

Now, let us assume that the processor is pipelined and its pipeline depth is equal to 4. For the first schedule example, the pipeline achieves the maximum throughput because all the pipeline stages are filled with independent operations. The processor takes data either from a source (i.e. the first four iterations) or from itself (i.e. the remaining iterations).

In the second schedule example, the pipeline is underutilized because the data generated by the processor is not yet available at its input due to the pipelining. This case should be avoided as the pipeline of the IP core is used for only 25% of its capacity.

5 Solution Approach

The number of independent operations that can be mapped onto a processor is a key metric to obtain efficient implementations. Hence, we developed a procedure that reports the number of independent operations that are mapped onto a Virtual Processor that has selfloops. We call this the *Profiler* of Laura. The procedure computes the size of a selfloop, which represents the amount of independent operations for a processor. The number is reported back to the designer who may take the necessary decisions to improve the quality of the analyzed design. Design decisions such as retiming (i.e. skewing), loop swapping, adding more streams of the same problem and unfolding can be applied to control the amount of independent operations that are mapped onto a processor. Using the procedure to compute the size of the selfloop and the given design decisions, we can close the loop in Figure 1 needed to perform a design space exploration to improve the efficiency of an input algorithm mapped into a reconfigurable platform.

6 Detecting the Selfloop Size

Normally, we cannot determine the size of a FIFO in a PN as we only specify the partial order between the processors. To determine the size of each FIFO would require a global schedule of the network, which cannot be easily determined. The selfloop is, however, a special case as this kind of communication channel starts and ends on the same processor and the writing and reading to/from this channel respects the internal schedule of the processor. We exploit this special case to determine the size of the FIFO channel.

We use the polytopes model [2] to represent mathematically a statical nested for loop program that is taken as input by our tool chain. Each processor is characterized by a polytope that represents its iteration space. The original for-loops gives us a schedule on this iteration space. This schedule is implemented in the Read and Write units of the Virtual Processor and its responsibility is to communicate the right data with other processors in the network.

The order in which write and read operations are performed over the selfloop depends on the local schedule of the processor. The selfloop size is given by the number of write operations that are done before the first read operation is performed, which defines the *run in* period. Due to the schedule of a processor, it is guaranteed that after the first read is done, other read operations will follow at constant time intervals, which defines the *steady stage*. In this stage, the read operations are interleaved with the write operations. The steady stage is followed by the *run out* period, which is characterized by performing only read operations. Hence, to compute the selfloop size, it is necessary to determine how many write operations were made in the run in period. This is equivalent to counting the number of integral points in the polytope that defines the run in period. To compute the points, we made use of the Ehrhart theory [4].

A processor can have one or more selfloops and each selfloop can have a different size. The minimum size out of all the processor selfloops represents the minimum amount of independent operations that are mapped on the IP core. Therefore, we consider this minimum size for our efficiency analysis.

7 Increasing the Throughput of a Virtual Processor

The Virtual Processor is a pipelined model; reading, executing, and writing are done in a pipeline fashion, as shown in Figure 2. To process a token, the read unit requires τ_{read} cycles to fetch the token. The execute unit requires $\delta_{pipeline}$ cycles in the IP core to execute. The write unit requires τ_{write} cycles to write a token into an output FIFO. To achieve 100% utilization for the IP core in an ideal network, the required number of independent operations Σ that needs to be mapped on the IP core is given by Equation 1.

$$\Sigma = \delta_{pipeline} + \tau_{write} + \tau_{read} \tag{1}$$

7.1 Pipelined IP Cores and Independent Operation

To optimize the IP efficiency, we need to look at the number of pipeline stages and the size of the FIFO selfloop. If the size of the FIFO is smaller than the number of pipeline stages, then we can increase the number of independent operations. This can be done in two ways. The first way is to apply an unimodular transformation (e.g., skewing, loop swapping) on the local schedule of the Virtual Processor local schedule that contains the IP core. The second way is to add

$$P = \Sigma - Size_{FIFO} \tag{2}$$

independent streams of the same problem to be processed in a pipeline fashion by our network (i.e., data-stripping). This gives us the necessary independent operations in the processors for an optimal throughput. These P independent stream must be interleaved in a optimal way. We rely on Compaan to do this for us by rewriting the original Matlab code with one extra for-loop that has as upper bound the value P. This loop must be the inner most loop to achieve the interleaved execution of the streams. If the size of the selfloop is larger than the number of pipeline stages, then the core is overloaded with independent operations. We can take advantage of this surplus of independent operation by unrolling this processor to increase the number of parallel running resources. An unrolling operation balances the workload on two or more identical processors, and may improve the performance for the entire algorithm. The proposed procedures of independent streams and unrolling do not guarantee an efficiency of 100% of the IP core. This is because the core efficiency depends also on the data dependencies between different processors.

7.2 Single Cycle IP Cores and One Independent Operation

A special case exists when the IP core is not pipelined at all. If a selfloop of size 1 is mapped on the IP core, we have an instance of the classic case of *data hazard*. In this situation the selfloop is replaced with a simple wire between the read and write unit of the virtual processor. This technique is very beneficial, as the wire increases the throughput of the processor and requires less hardware resources.

8 QR: A Case Study

To explain the hints the profiler implemented in Laura presented and how that effects design decisions, we now consider the QR case, which is widely used in signal processing applications [11]. The Matlab code for the QR algorithm that can be processed by Compaan is presented in Figure 4. It shows two function calls (*bcell* and *icell*) surrounded by parameterized for-loops. Compaan will generate a PN based on the for-loops and the variables passed on to the function calls. What happens in the *bcell* and *icell* is irrelevant to Compaan, but not for Laura. The network we obtain is given in Figure 4. Observe that each function call becomes a process in Compaan and that both the *bcell* and *icell* processes have selfloops.

```
for k = 1:1:T,
  for j = 1:1:N,
    [r(j,j), rr(j,j), a, b, d(k) ] =
      bcell( r(j,j), rr(j,j), x(k,j), d(k) );
    for i = j+1:1:N,
     [r(j,i), x(k,i)] =
       icell( r(j,i), x(k,i), a, b );
    end
  end
end
```

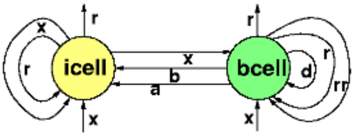


Fig. 4. The QR matlab code and the equivalent PN

The *bcell* and *icell* are realized by integrating a **bcell** and **icell IP** core on the execute unit in a virtual processor (See Figure 2) for the *icell* and *bcell* function call. The **bcell** and **icell** IP cores are pipelined and have 55 and 42 stages, respectively. To arrive to an optimal QR implementation each core must, therefore, have 57 and 44 independent operations, according to Equation 1. We look at QR with the parameters set to the typically values of **N = 7** and **T = 21**. For the case presented in Figure 4, our profiler reports for the smallest selfloop of **icell** a size five and for **bcell** a size 1. Given the deep pipelines of the **icell** and **bcell** IP cores, both of them are underutilized and the profiler recommends either to explore a space of additional independent QR streams from 1 to 57, or to perform a skewing operation.

First, we choose to explore the design decision of adding independent streams, and then we evaluate the design decision of skewing. For each experiment, we derive a VHDL representation of the algorithm using the Compaan/Laura tool chain that we can simulate to obtain quantitative data. The execution duration(No_{cycles}) for each experiment is measured in cycles. Each of the **icell**(No_{icell}) and **bcell**(No_{bcell}) operations contain 11 and 16 floating point operations, respectively. The average speed of our QR network mapped onto an FPGA [1] platform is 100Mhz. We use the next formula to compute how

[1] VIRTEX-II 6000, Synplify Pro7.2, Xilinx ISE5.2

many million floating point operations per second (MFLOPS) can be achieved in each experiment.

$$\Delta = \frac{No_{icell} * 11 + No_{bcell} * 16}{No_{cycles}} * 10^6 \quad (3)$$

8.1 QR: Adding More Streams

As we input the original QR algorithm in the chain, the profiler reports that we have to add 57 independent streams to the **bcell** IP core and 40 independent streams to the **icell** IP core by applying Equation 2. Hence, we choose to explore the space made by running 1, 10, 20, 30, 40 and 57 independent streams. In this case, the stream represent complete instances of the QR algorithm. The results of this exploration are given in Figure 5.

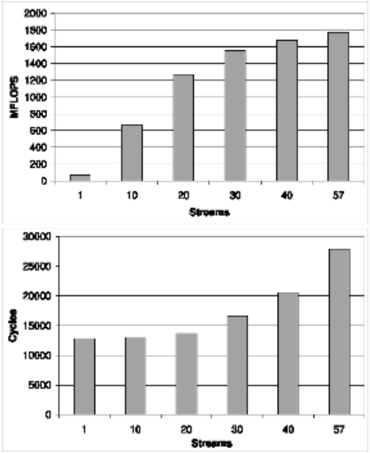


Fig. 5. QR experiments with additional streams

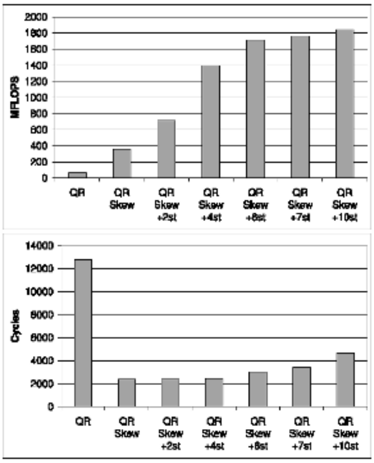


Fig. 6. QR experiments with skewing and additional streams

We observe that the saturation point is marked by the running 40 QR instances. At this point, the profiler reports 4.54 times more independent operations than the **icell** can handle. Therefore, we choose to unfold the **icell** twice and four times respectively. This leads us to 1764MFLOPS and 1767MFLOPS, respectively. The first transformation gives us an additional 81MFLOPS, while the second one only a difference of 3MFLOPS. It is obvious that the second operation is not as successful as the first one. We should also indicate that although we didn't use more IP cores, we obtained the higher throughput at the expense of more memory.

8.2 QR: Skewing

A second option to increase the number of independent operations in each processor is to skew the algorithm in Figure 4. After applying this transformation, the profiler indicates 7 independent operations in **bcell** and 21 in **icell**. However, these numbers are not sufficient to fill the entire pipeline.

To achieve a higher throughput for our experiment, we must either increase the dimension of the input problem or add more independent streams, i.e., QR instances. For the first case the required parameters will be **N = 57** and **T = 44**. For the second one, a space of 10 independent QRs is suggested by the profiler. We choose to explore 2, 4, 6, 7, and 10 independent streams. The results are shown in Figure 6. The first bar from the graph represents the throughput of the original QR, the second bar is the skewed version and the following bars are the values obtained for the skewed QR algorithm with increasing number of independent QR streams.

We choose to unroll the QR algorithm running 4 QR instances. In this case, the profiler reports a 181% utilization for the **icell**. Unrolling the **icell** core twice gives us only 6MFLOPS more throughput compared with the non-unfolded algorithm. This shows us that the decision was not so successful. The reason for this is that the unfolded **icell** cores work in a mutual exclusive fashion, nulling the effect of the unrolling operation.

8.3 Discussion

Based on the values given in Figure 6, skewing the algorithm gives us from the beginning an important amount of independent operations that allows to obtain a higher computational throughput without adding independent streams. The skew transformation fills the pipeline with parallel data that belongs to the same input problem. However, it may be that the skewing transformation doesn't achieve the maximum throughput of the cores and, therefore, solutions such as increasing the dimensions of the input problem (i.e. changing the values of **T** and **N**) or adding more independent streams may be required.

The existence of parallel operations that belongs to the original algorithm helps the designer in achieving the maximum throughput with minimum amount of independent streams added. These operations are exposed to the architecture through the skewing transformation. In the non-skewed version the saturation of our architecture is reached around 40 streams, while in the case of the skewed version the saturation point is given by 6 streams. In real life, it is more likely that only a small number of independent instances of an algorithm need to be computed at the same time. Therefore, the skewed version is more appealing to start with in any design space exploration. The overloading of an IP core with many streams can be solved either by increasing the clock speed for that particular processor (i.e., using various clock domains) or by unrolling it. In our experiment, we showed that applying high-level transformations (e.g., skewing, unfolding, data stripping) as offered by the Laura profiler as hints leads to an increased throughput. Nevertheless, quantitative simulation data is needed to assess its final usefulness.

9 Conclusions and Future Work

A particular characteristic of the derived networks we obtain from running the Compaan/Laura tools, is the existence of selfloops. These loops have a large impact on the utilization of the IP cores and in the final hardware implementation of a PN. This is especially the case when the IP cores are deeply pipelined. To improve the efficiency, the designer has to make design decisions like skewing, unrolling, loop swapping and data stripping. To help the designer to in making these decisions, we have implemented the profiler in Laura. The profiler uses manipulation of polytopes to compute at compile time the size of selfloops. This size is indicative for the number of independent operations available in an algorithm. The computed hints provides by the profiler, help to steer design decisions. Doing this in a iterative manner, a designer can explore options to improve the throughput of a process network. In this paper, we have shown for the QR algorithm that we could improve the performance from 60MFlops to 1.7GFlops by using the hints from the profiler. Using the hints, we could improve the utilization of mapping the QR algorithm on a FPGA with deeply pipelined IP cores by a factor of 30, using the same IP cores albeit at the expense of more memory.

To improve the efficiency, the designer has to make design decisions. These operations can be expressed at the Matlab level using the Mattransform tool. Currently, the hints provided by the profiler needs to be manually expressed. Future work is to make a connection between the Mattransform tool and the hints from the profiler.

References

1. A. Bakshi, V. K. Prasanna, and A. Ledeczi. Milan: A model based integrated simulation framework for design of embedded systems. In *ACM SIGPLAN workshop* , 2001.
2. U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, 1993.
3. S. Derrien and S. Rajoupadyhe. Loop tiling for reconfigurable accelerators. In *FPL 2001, UK*, 2001.
4. E. Ehrhart. *Polynômes arithmétiques et Méthode des Polyédres en Combinatoire*. Birkhäuser Verlag, Basel, international series of numerical mathematics vol. 35 edition, 1977.
5. V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. Pico: Automatically designing custom computers. *Computer*, 35(9):39–47, 2002.
6. B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *CODES*, San Diego, USA 2000.
7. C. Leiserson and J. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1, 1983.
8. K. S. Shayee, J. Park, and P. Diniz. Performance and area modeling of complete fpga designs in the presence of loop transformations. In *FPL 2003, Portugal*, 2003.
9. T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *CODES'02*, USA 2002.
10. T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using kahn process networks: The compaan/laura approach. In *DATE2004*, Paris, France, 2004.
11. R. Walke and R. Smith. 20 gflops qr processor on a xilinx virtex-e fpga. In *Advanced Signal Processing Algorithms, Architectures, and Implementations X*, volume 4116, 2000.
12. C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *FPL'03*, Portugal, 2003.

Distribution of Bitstream-Level IP Cores for Functional Evaluation Using FPGAs

Rawat Siripokarpirom

Department of Distributed Systems
Technical University Hamburg-Harburg
Schwarzenbergstr. 95, 21073 Hamburg, Germany
siripokarpirom@tuhh.de

Abstract. Due to their flexibility, increased logic density and low design costs, Field-Programmable Gate Arrays (FPGAs) have become a viable option for implementing many kinds of applications such as custom computing machines, rapid system prototyping, hardware emulation, IP verification and evaluation. This paper proposes an alternative approach that allows IP providers to deliver their IP to customers for functional evaluation before purchase, by mapping IP cores into SRAM-based FPGA logic and distributing them as a bitstream file for a particular device so that customers can use their FPGA boards to try-out the IP as a black-box, pre-verified design component. This paper also presents a simple hardware/software infrastructure and its prototype implementation that allows for seamless integration of hardware IP into an existing simulation environment. In addition, a case study is given to demonstrate the proposed approach and some security issues concerning bitstream-level IP distribution are also discussed.

1 Introduction

With the growing design productivity gap and the need for shorter time to market, reuse of intellectual property (IP) and rapid system prototyping methodology using programmable logic devices have become crucial for System-on-a-Chip (SoC) and embedded system designs. The design time can be significantly reduced if existing IP cores are (re-)used and embedded earlier into the design for evaluation purposes. IP cores are offered by many vendors and delivered in different forms for evaluation and deployment. Traditionally, customers have needed to rely on data sheets and pre-compiled, encrypted simulation models delivered by IP providers to evaluate IP before purchase. The idea presented in this work originates from the question whether it is possible to map IP blocks into FPGA logic and distribute them as a device programming file with associated software packages so that customers can test the IP blocks using an FPGA board at their local sites. Using hardware IP blocks may help speed-up a simulation process over traditional simulation approaches.

In order to enable customers to evaluate the hardware-mapped IP, they must be able to incorporate the IP block into their simulation environments. This requires a hardware/software infrastructure that facilitates IP evaluation on the

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 700–709, 2004.

customer side with minimal hardware requirements. In addition, due to a large variety of FPGA prototyping boards among customers, it is difficult to find a common HW/SW infrastructure that meets such requirements. These problems hamper the use of bitstream-level IP evaluation approach in practice. To address such problems, this paper proposes a cost-effective HW/SW infrastructure with minimal hardware requirements. Customers can evaluate IP blocks either standalone or together with other design components by (i) applying input vectors to the target IP block, (ii) retrieving output vectors from the device and (iii) finally sending them back to the host computer for visualization and analysis. In order to enable a software simulator to control and communicate with one or more hardware-implemented IP cores, this work makes use of a JTAG (Joint Test Action Group, the IEEE 1149.1 Standard [1]) link, which is widely used for system-level testing, debugging and device configuration. To be more specific, Xilinx's Parallel Cable III and Altera's ByteBlasterMV cable are used for Xilinx and Altera FPGA devices, respectively. Despite its bit-serial nature and moderate data transfer rate, using a JTAG link has many advantages such as low cost, low-pin-number requirements, a simple communication protocol, etc. Another advantage is that the same programming cable can also be used as a communication channel between the software and the internal FPGA logic, provided that a built-in JTAG hard-macro is available in the device. Recently, some FPGA vendors such as Xilinx and Altera have already embedded a built-in JTAG hard-macro into their devices. All these advantages make the proposed approach simply applicable to many FPGA boards available today.

The rest of the paper is organized as follows. Section 2 overviews related work. The hardware and software part of the implemented framework are described in Section 3 and 4, respectively. An application scenario of the proposed approach is presented in Section 5. Security issues associated with the use of a bitstream-level IP distribution approach is discussed in Section 6. Finally, Section 7 provides the conclusions for this work.

2 Related Work

In order to enable customers to evaluate an IP core in their designs prior to purchase, some IP providers offer their IP as pre-compiled and/or encrypted simulation models for functional simulation. An IP core may be developed as a behavioral, non-synthesizable model or a cycle-accurate, synthesizable model, written in a Hardware Description Language (HDL) or C/C++, and then compiled into a secure simulation model using a model compiler. However, these models are usually tool-specific or vendor-specific (e.g., Synopsys's SWIFT Interface and SmartModel Library [2] and Summit Design's Visual IP [3]). Some IP providers offer encrypted HDL models for FPGA-based designs under a product evaluation agreement. Such models can be synthesized and mapped into FPGA logic using vendor-specific design tools (e.g., Altera's OpenCore evaluation programs [4] and Mentor Graphics' Inventra IPX [5] used in Altera encrypted IP flow) so that customers can obtain more accurate information about the area and performance of the IP within their designs. Customers cannot generate the final bitstream or any output netlist file before purchasing the IP. Altera's OpenCore Plus evaluation programs allow customers to evaluate IP and test their designs in hardware before purchase by using a time-limited programming file. Time-limited IP cores operate for only a pre-determined number of clock cycles, after which they are disabled.

Another approach is to use a remote, distributed client-server simulation environment, for instance, as reported in [7][8][9]. A connection between the client's simulator and the server simulator running on a server remotely hosted by an IP provider must be established for IP evaluation. Customers can select an IP block, which will be instantiated and simulated by the server simulator. Through an associated wrapper simulation model, which is responsible for the secure communication and synchronization with the server simulator, customers can evaluate the IP block as if it were instantiated and simulated locally on their computer. IP blocks are protected because they are not delivered to customers. The main drawback of such a remote IP evaluation approach is that it causes a large amount of communication and synchronization overhead and the Internet bandwidth (or traffic) can have a large impact on the overall simulation performance. In addition, sending data through the Internet in a secure manner requires considerable efforts. In order to serve multiple users concurrently at the same time, multiple licenses for third-party HDL simulators and additional computing resources may be required at the IP provider site. Instead of using an HDL simulator, a hardware emulator or a simulation accelerator can be deployed on the server side. For example, Aptix's IP Test Drive [6] offers the capability to evaluate hardware-mapped IP over the Internet. The user can browse the IP repository and select an IP core from a catalog for a test run. If the selected IP core is already mapped into an Aptix FPGA-based emulation system located remotely on an emulation-services site, then user can verify the IP core using a testbench. In [10] the authors use Java applets to deliver IP cores for functional simulation within a Web browser running on the user computer. However, the IP cores must be described in JHDL, a Java API for circuit modeling, and compiled as Java classes. JHDL supports only Xilinx FPGA families.

3 JTAG-Based Hardware Infrastructure for IP Evaluation

As mentioned previously, one needs a hardware infrastructure that facilitates the control of hardware execution of one or more IP blocks in the FPGA. To this end, a simple JTAG-based internal scan structure as illustrated in Figure 1 has been proposed. In this structure, a so-called *Virtual I/O Socket*, which mainly consists of a capture/shift register and an input update register, is needed for each IP block. The output ports of the input update register are connected to the input ports of the IP block. The output ports of the IP block are connected to the capture ports of the capture/shift register. The bit width of the capture/shift register is equal to the maximum value of the total bit width of all inputs and

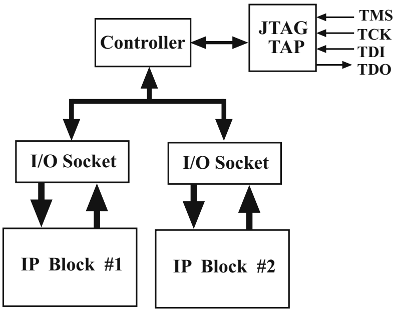


Fig. 1. Scan-based hardware structure supporting multiple IP blocks

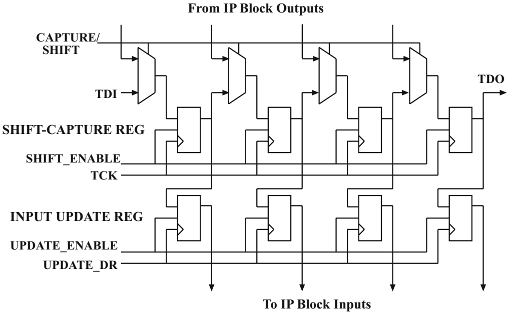


Fig. 2. Virtual I/O socket of bit length 4

the total bit width of all outputs of the IP block because this register is shared by the input and the output vector. The bit width of the input update register is equal to the number of the input ports of the IP block.

Figure 2 shows an example of a virtual I/O socket, where both the capture-shift register and the input update register have the same length of 4 bits. As illustrated in Figure 2, the capture-shift register is built with a chain of flip-flops and 2:1 multiplexors. The input update register is composed solely of flip-flops. All virtual I/O sockets are controlled by a controller. This controller uses signals that are provided by the built-in JTAG TAP unit (e.g. TCK, TDI, SHIFT-DR, UPDATE-DR, etc.) to generate clock and control signals. Control signals are used, for example, to select an I/O socket and to control the capture-shift-update operations.

This hardware structure is very similar to a typical boundary-scan chain. The key difference is that one or more internal scan paths are built using FPGA logic resources instead of using boundary-scan cells. Therefore, the length and the number of scan paths are not limited by the number of available pins of the device. This allows the user to set or control the value of the input port of arbitrary bit length and capture the value from the output port of arbitrary bit length. The values for all input signals are generated by the software and applied to the IP block through the internal scan structure.

To apply an input test vector, two capture-shift-update cycles are needed. In the first cycle, a new input vector is shifted into the capture/shift register of an I/O socket and then the input update register is updated with the new value stored in the capture/shift register. Note that the output value that was captured from the output of the IP block at the beginning of the first cycle and shifted out during the shift operation is unused and can be ignored. The second cycle begins with the capture operation, followed by the shift and update operation. During this cycle the shifted-in value in the capture/shift register is ignored by the input update register (i.e., the input to the IP block is unchanged and stable). The time period between these two consecutive cycles must be longer than the largest propagation delays and clock-to-output delays inside the IP block and the output is captured when all signals have become stable after applying an input vector to the circuit. In case the circuit has a clock input to the flip-flops, it must be ensured that the setup-time condition of the flip-flops is always hold. This can be achieved by properly delaying the clock input signal to the IP block. During the capture operation all input signals except the clock signal are applied directly to the input ports of the IP block, whereas the clock signal is delayed by a half period of the TCK signal. This value is sufficiently large enough for the most IP cores if assumed that the TCK frequency is not large than 1 MHz. The TCK signal is generated by the control software on the host PC. Note that the proposed hardware structure provides both controllability and observability at the I/O interface of the IP block. This is sufficient for black-box IP models whose contents are treated as “invisible”.

4 Integrating IP into Simulation Environment

In order to use the IP cores for simulation, the user needs a bitstream file to program the target device. A parameterizable VHDL model of the scan structure described in Section 3 has been developed. This model must be implemented together with the selected IP cores. Currently, all these components are instantiated manually and connected together in a VHDL file that serves as the top-level entity for the final FPGA implementation. All these files are passed to the standard FPGA design flow to generate the final bitstream. To facilitate device configuration and communication through a JTAG link, a Java API was implemented. It supports both Xilinx and Altera devices. The Java Native Interface (JNI) is used to enable the Java API to call low-level C routines to access the Parallel port attached to the download cable, while trying to keep the platform-specific part low to reduce platform dependency. The Java-based API supports both Windows and Linux PC platforms. To demonstrate how to interface an FPGA-mapped IP core with a simulation environment, a Java-based digital circuit simulation framework (with an event-driven simulation kernel) has been developed [11], which is similar to the JHDL framework. It can perform both EDIF netlist simulation and FPGA-based hardware emulation. For simplicity of

```
public class Counter extends ExternalComponent {
  public Counter( Cell parent, int outputWidth ) {
    clkinput("CLK"); // the clock input
    input("RESET"); // the reset input (active-low)
    input("ENABLE"); // the count enable
    input("UPDOWN"); // the up/down control (up=0,down=1)
    output("Q", outputWidth ); // the counter output
  }
  public void evaluate() {
    emulate(); // call the emulate() method of its superclass
  }
}
```

Fig. 3. A Java wrapper class for the binary counter

explanation, a simple 16-bit binary counter is chosen as an IP block under evaluation. This component has four inputs (*clk*, *reset*, *enable*, and *updown*) and one output (Q). The counter value is set to 0 if the asynchronous reset is low. Otherwise, it is either incremented or decremented by one (controlled by the updown input) at the rising clock edge and when enable is high. To simulate the counter, a Java wrapper class must be created, as shown in Figure 3. During simulation, each time when any input of the counter module is changed, the `evaluate()` method is invoked, which in turn calls the `emulate()` method of its superclass. This method is responsible for applying new input values and retrieving output values of the IP block in the hardware over the JTAG link.

```
// create Wire objects associated with the input ports of the cell
Wire rst = c.getWire("RESET"), clk = c.getWire("CLK"),
     enable = c.getWire("ENABLE"), updown = c.getWire("UPDOWN");
for(int i=0; i<100; i++){ // generate input vectors for 100 cycles
  rst.assign( (i==0) ? ’0’ : ’1’);
  enable.assign( (i > 3 && i < 95) ? ’1’ : ’0’);
  updown.assign( (i < 50) ? ’0’ : ’1’ );
  clk.assign(’1’); sim.runFor(10.0); // clk ’1’ for 10ns
  clk.assign(’0’); sim.runFor(10.0); // clk ’0’ for 10ns
}
```

Fig. 4. A portion of Java code for input stimuli generation

The Java code for input stimuli generation is shown in Figure 4. Input vectors are generated by assigning values to the `Wire` object associated with each input port. The Java testbench code is shown in Figure 5. In this example, the target hardware platform is an FPGA board with only one Xilinx Virtex device (with the instruction length of 5 bits) in the JTAG chain.

In addition, an AES core from [12] was used as a more complex IP core. The Verilog code of the AES core was synthesized into a gate-level EDIF netlist and

```
public void main( String args[] ) {
  Cell c = new Counter(null, 16); // create a 16-bit counter cell
  SimTableAdapter sta = new SimTableAdapter(); // create a signal tracer
  sta.add( c.getPortList() ); // keep trace of all I/O signals
  try {
    // define the JTAG chain of the board
    JtagChainInfo jci = new JtagChainInfo(new int[]{5});
    jci.selectDeviceID(0); // select the 1st device in the chain
    // create the board manager for Xilinx Virtex FPGA
    bm = new JtagBoardManager(new VirtexJtagProgrammer("LPT1",jci));
    bm.configure("counter16.bit"); // program the device
    bm.register(0x01, c); // register the counter IP (ID=0x01)
  } catch (JtagException jex) { // error report
    System.err.println(jex.getMessage()); System.exit(-1);
  }
  Simulator sim = new Simulator(); // create the simulator
  sim.load( c ); // load the cell into the simulator
  sim.addTraceEventListener( sta ); // add a TraceEvent listener
  sim.start( ); // start the simulator
  /** CODE FOR INPUT STIMULI GENERATION GOES HERE. **/
  sim.stop(); // stop the simulator
  sta.trace2vcd("wave.vcd"); // generate a VCD output file
}
```

Fig. 5. A Java testbench for the binary counter

a structural VHDL netlist, targeting the Xilinx Virtex-E family. The AES core (together with the scan structure) was mapped into a Virtex XCV300E chip using the Xilinx ISE 5.1 tools. The synthesis results are shown in Table 1. The AES IP block in the last row includes the original AES core, its I/O socket and the controller. The difference in logic utilization mainly comes from the fact that extra resources are needed to implement the internal scan structure. In this case study, the AES core has a very high-pin count, and therefore, requires a very long capture-shift and input update register.

Table 1. Logic utilization (targeting an XCV300E-PQ240 device)

	#Slices	#DFFs+Latches
AES Core (original)	1862 (60.61%)	611 (8.59%)
AES IP Block	1995 (64.94%)	1155 (16.23%)

For performance comparison, three simulation approaches were performed using the same testbench (either in Java or in VHDL, but functionally equivalent): (i) gate-level simulation using a Java-based hardware simulator, (ii) gate-level

Table 2. Simulation performance comparison

	Runtime (500 cycles)	Speed (cycles/sec)
Gate-level Java simulation	475.2 sec	1.05
Gate-level VHDL simulation	182.5 sec	2.74
FPGA-based emulation	3.0 sec	166.67

VHDL simulation using ModelSim and (iii) hardware emulation using an FPGA board with Java testbench. All experiments were done on a Notebook with an 800MHz processor, running Windows XP. The simulation results are shown in Table 2. For each experiment, the simulation was run for 500 clock cycles. An example of the Java testbench can be found in [11]. Although only a Java-based simulation framework was chosen to demonstrate the functionality of the proposed approach, the described method can be adapted to support other simulation environments. For instance, one may implement the JTAG communication layer in C/C++ and use it directly with the SystemC or other HDL simulators (e.g., via the Verilog Programming Language Interface). The proposed approach can be used with any FPGA board, provided that the FPGA device has enough logic capacity and a built-in JTAG unit, and can be programmed through a download cable with JTAG probes. Indeed, many commercially available FPGA boards fulfill these hardware requirements.

5 Bitstream-Level IP Distribution Scenario

One may envision an application scenario in which an IP provider offers online IP evaluation services. A user can visit the IP provider's Web site and request for one or more IP cores for evaluation that will be distributed in a bitstream format. Before downloading any bitstream file, the user must first complete the user registration process and accept some IP evaluation agreements, and after that he or she has to submit some necessary information like the name and parameters (if needed) of the IP core and the target FPGA device. Such information will be used by the server to generate the bitstream data for the requested IP core. The selected IP core together with its control and interface logic, which is required for the communication with the software running on the host PC, is passed through a standard FPGA design flow, including synthesis, mapping, placement and routing and bitstream generation, respectively. The IP provider may provide pre-generated bitstream files for some selected FPGA devices so that the implementation steps can be skipped, resulting in shorter processing time on the server. In the next step, the user can download the bitstream file, together with the software packages that are necessary for interfacing the implemented IP core to the user's simulation environment. The user can purchase a license when he or she is completely satisfied with a core's functionality after evaluation. After purchase the user will receive the original IP cores, which may be available in different forms (e.g., RT-level HDL models or technology-specific gate-level netlists). In addition, it is possible to generate time-limited versions of hardware-implemented IP blocks, which operate only for a pre-defined number of simulation clock cycles. A simple hardware counter associated with each IP block is implemented in FPGA logic and incremented each time when an input vector is applied to the IP block during simulation. When it reaches a pre-defined value, the communication with the corresponding IP block is disabled.

IP providers can offer such a service as an additional means of distributing evaluation IP to their customers. Another advantage is that a design team can use this kind of hardware-verified IP from other sources in earlier stages of the design cycle. This reduces the development time, cost and risks significantly.

6 Security Concerns

IP core protection against illegal use or IP theft is always a major concern of IP providers when distributing their IP cores to customers. Since IP cores are wrapped with interface logic and distributed in a vendor-specific, proprietary bitstream format, they cannot be directly copied and integrated into any design for final implementations without extracting the circuit netlist and other necessary information from the bitstream data. Although reverse engineering of a bitstream is possible and can be used to extract the implemented IP cores, it is an extremely difficult process and requires large efforts (e.g., time and money). The bitstream by itself does not preserve any design hierarchy or names of instances and nets, so it is hard to decipher anything out of it. While it may be cost effective to reverse-engineer a small design targeted at a simple FPGA architecture, it is a daunting task to extract individual IP blocks from a flat netlist of a one-million gate design implemented in an FPGA with a more complex architecture. Even if someone were able to write a program for reverse-engineering of bitstreams that extracts the circuit information from a bitstream file, some technical questions still arise:

- Is the extracted netlist functionally equivalent to the original IP core?
- Does the extracted netlist have any circuit part that does not belong to the original IP core (e.g. the scan structure in our case) that can be removed or optimized away while preserving the original functionality of the IP core? How to locate them in the netlist?
- Can the extracted netlist be converted to other technologies targeting ASIC or SoC designs and meet given constraints (e.g., area and performance)?

Furthermore, the IP provider may generate a *bitstream-evaluation* version of an IP core which has not been optimized against any constraints (e.g., area and timing) during synthesis. This evaluation version can be considered as a lower-quality, but functionally-equivalent version of the original IP. In such a case, if someone could extract the circuit netlist from the bitstream, he or she would get only an evaluation version.

7 Conclusion

In this paper a methodology has been proposed that allows IP providers to map their IP cores into SRAM-based FPGA logic and distribute them to customers for functional evaluation. Customers can evaluate the IP core stand-alone or together with other components in their design. A HW/SW framework and its prototype implementation that demonstrates the underlying concept were also presented. The implemented framework supports both Xilinx and Altera devices and, therefore, can be used with various FPGA boards. Security issues associated with the bitstream-level IP evaluation approach were also discussed. Despite some limitations such as no support for timing simulation, limited visibility (due to the use of black-box IP models) and moderate security for IP protection, the proposed approach offers a new opportunity for both FPGA vendors and IP providers to deliver their IP as hardware executables to customers for functional evaluation.

References

1. IEEE Standard Test Access Port and Boundary-Scan Architecture. IEEE Std. 1149.1-1990, May 1990.
2. Synopsys Inc., Guide to SmartModel Documentation, November 2003, http://www.synopsys.com/
3. Summit Design, Inc., Visual IP, http://www.summit-design.com/
4. Altera Corp. IP Megastore web site, http://www.altera.com/products/ip/design/ipm-design.html
5. Inventra IPX Version 1.1 for LeonardoSpectrum Altera Release, http://www.mentor.com/inventra/ipx/
6. Aptix Corp., IP Test Drive, http://www.eSoCverify.com
7. M. Dalpasso, A. Bogliolo and L. Benini, *Specification and Validation of distributed IP-based designs with JavaCAD.* In Proc. of Design, Automation and Test in Europe Conference & Exhibition, pages 684-688, January 1999, Munich, Germany.
8. A. Fin, and F. Fummi, *A Web-CAD methodology for IP-core analysis and simulation.* In Proc. of the 37th Design Automation Conference, pages 597-600, June 05-09, 2000, Los Angeles, California, USA.
9. H.-P. Wen, C.-Y. Lin and Y.-L. Lin, *Concurrent Simulation-based Remote IP Evaluation over the Internet for System-on-a-Chip Design.* In Proc. of International Symposium on Systems Synthesis, pages 233-238, Sept. 30 - Oct. 3, 2001, Montreal, Quebec, Canada.
10. M.J. Wirthlin and B. McMurtrey, *IP delivery for FPGAs using Applets and JHDL.* In Proc. of the 39th ACM/IEEE Design Automation Conference, pages 2-7, June 10-14, 2002, New Orleans, Louisiana, USA.
11. R. Siripokarpirom and F. Mayer-Lindenberg, *Hardware-Assisted Simulation and Evaluation of IP Cores Using FPGA-based Rapid Prototyping Boards.* To appear in Proc. of the 15th IEEE International Workshop on Rapid System Prototyping (RSP 2004), June 28-30, 2004, Geneva, Switzerland.
12. OPENCORES.ORG, AES Project, http://www.opencores.org/

SOC and RTOS: Managing IPs and Tasks Communications

Arthur Segard and François Verdier

ETIS Laboratory, UMR CNRS 8051
ENSEA - Université de Cergy-Pontoise,
6 av. du Ponceau – 95014 Cergy-Pontoise – FRANCE
{segard,verdier}@ensea.fr

Abstract. We present in this paper a development method for SOC platforms associating a processor running a RTOS and hardware IP's. This method is based on encapsulating IP's for unifying communications and resource sharing between software and hardware tasks. A hardware wrapper that abstracts the IP behaviours through a standard interface and a software encapsulation (the IP *Alter Ego*) thereby giving access to the Operating System functions are presented. Details about implementing this model on two different FPGA platforms are given.

1 Introduction

Today's demand for highly complex digital systems has lead to the integration of several complementary often heterogeneous processing modules into a single chip. Such System-On-Chips (SOC's) are particularly good candidates for a wide range of new processing cores in multimedia applications. SOC systems are generally built around an Instruction Set Processor (ISP) connected to modules optimized for specific computations. Examples of such modules are data compression (MPEG-2, JPEG2000) or channel coding/decoding cores in telecommunication applications (VITERBI, Turbo-Codes, LDPC).

Recent advances in programmable technologies have allowed the integration of some reconfigurable areas (FPGA nodes for example) in which Intelectual Properties (IP) can be allocated. These new architectures are, in order of magnitude, faster than processor-only solutions and sustain flexibility. However, exploiting this flexibility leads to a very complex design flow. Different models and abstraction levels have to be handled for designing both processor code, ASIC or FPGA designs and communications between all modules have to be carefully and efficiently designed (busses, networks, protocols). In addition, an efficient memory management is also necessary for designing good applications.

Many different methods have been used to speed up the desing flow. One such method which have received a lot of attention is the hardware and software co-design methodology. A unified language describes the whole architecture. Compilation, performance analysis, hardware/software partitionning, RTOS targetting are steps necessary to generate both sequential code for ISP and bit-streams for

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 710–718, 2004.

programmable nodes. The Pilot [1] method or OCAPI-XL [2] are good examples of such methods. The latter uses robust communication methods [3] and targets dynamic and partial reconfiguration of programmable logic fabrics. However, the communication and interconnection resources are strongly constrained (for data, control and configuration streams), and hence, do not seem adaptable to every SOC or FPGA platforms.

SOC's can also be considered as heterogeneous multi-processors systems. Automatic application generation tools [4] can here be used by targetting the specific architecture. Once again, these tools depends on robust communications concepts [5] based on high levels of abstraction. However dynamic reconfiguration of FPGA nodes cannot be taken into account without great efforts.

The operating model presented in this paper covers the development of complex SOC's with one ISP running a simple multi-task real-time operating system and several dedicated reconfigurable (or not) hardware modules. The aim of this work is to target the wider possible range of architectures and to propose a generic model for handling such systems. This model will allow any hardware or software module to ask for a system call, to execute any software function, including communication management functions.

This article is organized as follows: Section 2 presents our model for unifying communications between heterogeneous IP's (the *Alter-Ego* model). Section 3 gives details about the implementation of this model and illustrates its portability through two different implementations. Section 4 presents the dynamic and partial reconfiguration perspectives of this work and Section 5, our conclusions.

2 Software *Alter-Ego* Model Description

In a real-time operating system a task is a portion of code to which (among other informations) a unique identifier and a priority is given. The goal of the scheduler task is to share the processing time (and resources) between all tasks depending on their priorities. A task can ask for the creation of another task that will be also executed by the processor. The *Alter Ego* (AE) model is based on a simple and portable real-time OS and allows any module in the SOC to ask for a system call such as task creation, semaphore query, etc. This is done by encapsulating each IP module within a software task - the *Alter Ego* - managed by the OS. An IP can communicate with its corresponding task, this task being in charge of executing the requested calls.

The objective of this work is to provide a unified level that abstracts hardware IP's through software-only tasks. Developing an application under such a model makes managing hardware tasks simple.

IP encapsulation is done on three levels. The first one is the IP wrapper - a wrapper between the hardware IP and the SOC communication channel. A specific Interrupt Service Routine (ISR) constitutes the second level. The third level being the AE itself. These three levels are presented below.

2.1 IP Wrapper

Each hardware (or software) IP is encapsulated in a wrapper whose role is to establish a link between the IP and the main ISP through the SOC communication channels. The wrapper is composed of three information registers: 1) a control register, 2) a request register and 3) a data return register. A dedicated signal is also generated by the wrapper in order to interrupt the ISP. Figure 1 illustrates the wrapper interface.

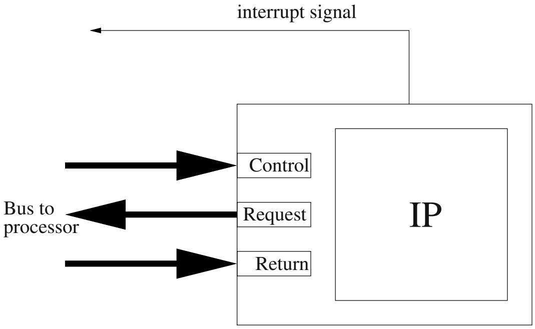


Fig. 1. Interface to the IP

1. The control register is used by the AE to initialize the IP (configuring or transmitting some parameters) and to control the IP (execution start, stop, suspend). This register gives to the processor the total or partial control over the IP behaviour. By using the processor interrupt controller, the IP uses its interrupt signal to request attention to the processor. On activation of this signal the Interrupt Service Routine wakes up the AE.
2. The request register contains the “instruction” the IP is requesting. This instruction can be any function identifier along with all of its parameters. The AE is in charge of interpreting the “instruction”.
3. The data return register is used by the processor to send back to the IP the results of the requested function (if any).

Depending on the IP the three registers can be implemented in a form more suitable to the IP behaviour. FIFO memory or shared-memory spaces can also be used instead of registers for transferring important volumes of data.

2.2 The *Alter Ego*

The behavior of the AE is as follows (see Figure 2): The first step consists in transmitting control data to the IP through the control register. This step initializes the IP. Where reconfigurable nodes are concerned, this step is also responsible in the IP configuration phase i.e. transmitting the configuration bistream

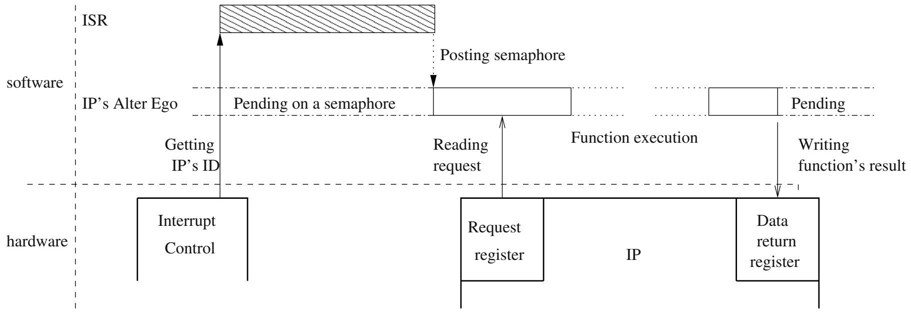


Fig. 2. Hardware/software scheduling

to the FPGA node for example (see Section 4). The AE is then placed in a sleeping state by waiting for the IP semaphore to be released. At this point, the consequence is that apart the configuration (if any) and the initialization phase, the IP AE does not represent any execution charge to the processor. Figures 3 and 4 give examples of coding AE and ISR.

```
Sem_ptr *sem_array[];

Void alter_ego (void) {
    int request;
    int result;
    func_ptr func_array[];
    init_func(func_array);
    Init_IP() ;
    sem_array[IP_id]=OSSemCreate();
    While(1) {
      OSSemPend(sem_array[IP_id]);            /* Wait for semaphore */
      request=Get_Request_Reg();              /* Get request number */
      result=Exec_func(func_array[request]);  /* Execute function */
      Write_Result_Reg(result);               /* Send back result */
    }
}
```

Fig. 3. *Alter Ego* code example

By activating its interrupt request, the IP initiates the execution of the ISR whose role is to release the semaphore. The AE is then woken up and enters the communication phase with the IP. This phase comprises three steps:

1. Reading of the requested function ID,
2. Execution of the software function (system call, software library function, etc),

```
Void int_control(void) {
  int IP_id ;           /* IP identifier */

  IP_id=get_IRQ_number( );  /* identify IRQ request */

  OSSemPost(sem_array[IP_id]);  /* release IP semaphore */
}
```

Fig. 4. ISR source code

3. Where applicable, transmitting the function results back to the IP through the data return register.

This model can be used also in a multi-processor operating system context. However, this model applies only if the IP behaviour can be considered as a finite state machine.

3 AE Implementations

The AE model was successfully implemented on two different SOPC platforms. The first one was a Xilinx development board equipped with a VirtexII-pro FPGA that integrates a PowerPC processor (PowerPC 405) and a 1-million gates FPGA. The PowerPC processor runs a very simple real-time operating system: μCOS-II [6]. The OS kernel is very compact and is based on a preemptive scheduler. Classical inter-task communication primitives are included in this OS: semaphores, mailboxes and signals.

For validation purpose, two software IP's were linked to the processor in the form of two 32-bit microBlaze processors running specific code. The two microBlaze processors were encapsulated as described in the previous Section. Figure 5 gives an overview of the SOC architecture used in this platform.

The second platform is described Section 3.4.

3.1 Critical Resources Sharing Example

On this platform, our *Alter Ego* model was employed to share critical resources between the two IP's and the central processor. In a multi-task context, sharing resources can be done by using semaphores. Our model constitutes a way to share a critical resource (a UART IP on the OPB bus in our case) among multiple processors without the need of a complex multi-processor operating system. A mutual exculsion semaphore (MUTEX) was attributed to the UART.

The AE associated to each microBlaze processors works as follows:

1. MUTEX request (OS primitive)
2. use of the UART (IP native code)
3. MUTEX restitution (OS primitive)

MUTEX request and restitution by the microBlaze are done thanks to the AE running on the PowerPC.

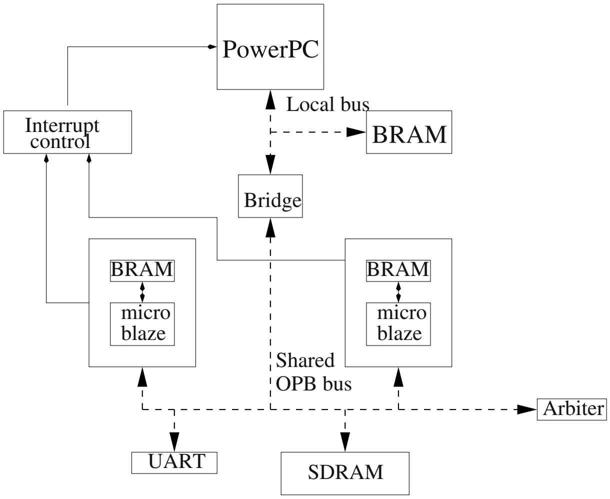


Fig. 5. Interface to the IP

3.2 Memory Allocation Example

A more complex communication scheme was also available by using the same AE behaviours when dynamic allocations in central memory are needed by multiple IPs. In this case, executions of the library function *malloc()* were requested by the IP's. Request registers in the IP wrappers contained here both the predefined *malloc()* function ID and the size of the memory bloc needed. Data return registers were filled by AE's with a pointer on the allocated memory bloc. A MUTEX was then be used for important data exchanges between the two IPs and the main processor.

3.3 Implementation Costs

Table 1 gives implementation results of AE and ISR codes. These results were extracted from a PowerPC405 platform running μCOS-II. Code memory occupation and number of bus transfers are given in 32 bits words. The AE costs are valid for execution of functions with no parameters and integer return type (similar to code on Figure 3). More complex functions calls would cost a few more bus transfers (for parameters and results) and CPU cycles.

As we note in Table 1 both memory and CPU occupation are very restricted. ISR execution time is strictly constant and predictable thus matches real time constraints.

3.4 Model Portability

Portability of this model was validated with another programmable platform: an ALTERA FPGA development board with an Apex20K chip. The central processor here, was a 32-bit NIOS processor running the same RTOS. Two dedicated

Table 1. Memory occupation, CPU cycles and bus occupation of AE and ISR

	Code memory (32b words)	CPU cycles	bus transfers
AE	176	358	2
ISR	52	92	1

IP's were added: an integer *log* and a *sqrt* hardware cores computing respectively the natural logarithm and the square root. Two MUTEXes were used again for sharing the IP's between multiple tasks. Despite its simplistic nature, this example showed the portability of the AE model.

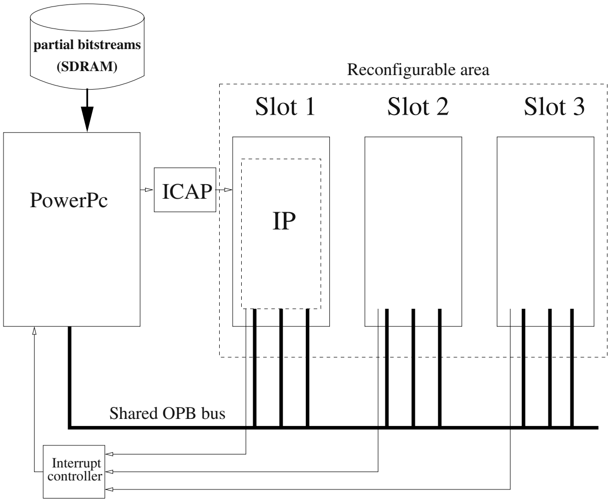


Fig. 6. Implementation of partial and dynamic reconfiguration on a VirtexII-pro platform

4 Partial and Dynamic Reconfiguration Perspectives

Practical partial reconfiguration of IP's is not yet implemented in our platforms. However, both the necessary hardware structure and specific OS services were defined. Figure 6 shows a VirtexII-pro architecture allowing the implementation of partial reconfiguration. This hardware organization is similar to the one suggested in [7]. Several vertical IP slots (due to the vertical reconfiguration frames) are created on the FPGA fabric. These slots can be filled with partial configuration bit-streams through the processor ICAP interface. Control, request and data return registers of IP's are accessible through a shared bus crossing all slots and implemented with *bus macros*. The IP's interrupt signals are connected to a single interrupt controller.

A dedicated OS *reconfiguration service* will be necessary for a software management of partial reconfigurations. This service will be invoked by AE in the initialization phase for i) finding an empty FPGA slot and ii) transferring the IP bit-stream to the ICAP interface. The development of this specific OS service and the associated hardware implementation is still under progress.

5 Conclusion

We presented in this paper a development method for SOC platforms. This method consists in encapsulating IP's and unifying communications and resource-sharing between software and hardware tasks. This encapsulation is twofold: i) a hardware wrapper that abstracts the IP behaviours through a standard interface – control, request and data return channels – and ii) a software encapsulation (the IP AE) giving access to the Operating System functions.

Advantages of this method reside in the fact that all hardware IPs are handled as software tasks. Creating, deleting and communicating with IPs is made possible through classical OS primitives. By using their AE, hardware IP's can interface with software functions and ask for system calls. This model was developed to target real-time embedded systems and thus optimized to use only a very restricted bus bandwidth, low memory occupation and restricted additional gate count. This model is portable and may be adapted to any kind of embedded operating systems.

Dynamic reconfiguration of IPs was taken into account in this model and constitutes the future development. Based on the *Alter Ego* model we can argue that hard-soft migration is also possible by including any software code emulating the IP's in the AE tasks. Calls to a dedicated *hardware scheduler* could manage reconfigurable resources and distinguish between different hard-soft partitions depending on quality of service goals.

References

1. Chen, Z., Cong, J., Fan, Y., Yang, X., Zhang, Z.: Pilot- A Platform-Based HW/SW Synthesis for FPSoC. In: Workshop on Software support for Reconfigurable systems. (2003)
2. Vanmeerbeeck, G., Schaumont, P., Vernalde, S., Engels, M., Bolsens, I.: Hardware/Software partitioning of embedded system in OCAPI-xl . In: Ninth International Symposium on Hardware/Software Codesign. (2000)
3. Marescaux, T., Mignolet, J.Y., Bartic, A., Moffat, W., Verkest, D., Vernalde, S., Lauwereins, R.: Networks on Chip as Hardware Component of an OS for Reconfigurable Systems. In: FPL. (2003)
4. Lyonnard, D., Yoo, S., Baghdadi, A., Jerraya, A.A.: Automatic Generation of Application-Specific Architecture for Heterogeneous Multiprocessor System-on-Chip. In: DAC. (2001)

5. Svarstad, K., Nicolescu, G., Jerraya, A.A.: A Model for Describing Communication between Aggregate Objects in the Specification and Design of Embedded Systems. In: DATE. (2001)
6. Labrosse, J.J.: MicroC/OS-II, The Real-Time Kernel. CMP Books, Lawrence, Kansas 66046, USA (2002)
7. Blodget, B., McMillan, S., Lysaght, P.: A Lighweight Approach for Embedded Reconfiguration of FPGAs. In: Proceedings of DATE, (2003) 399–400

The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays

Steven J.E. Wilton[1], Su-Shin Ang[2], and Wayne Luk[2]

[1]Dept. of Electrical and Computer Eng.
University of British Columbia
Vancouver, B.C., Canada
[2]Department of Computing
Imperial College
London, England

Abstract. This paper investigates experimentally the quantitative impact of pipelining on energy per operation for two representative FPGA devices: a 0.13µm CMOS high density/high speed FPGA (Altera Stratix EP1S40), and a 0.18µm CMOS low-cost FPGA (Xilinx XC2S200). The results are obtained by both measurements and execution of vendor-supplied tools for power estimation. It is found that pipelining can reduce the amount of energy per operation by between 40% and 90%. Further reduction in energy consumption can be achieved by power-aware clustering, although the effect becomes less pronounced for circuits with a large number of pipeline stages.

1 Introduction

Energy consumption has become a critical concern for Field-Programmable Gate Arrays (FPGAs). The programmability of FPGAs is afforded through the use of long routing tracks and programmable switches laden with parasitic capacitance. During high-speed operation, the switching of these tracks causes significant power dissipation. Hence an FPGA can consume up to two orders of magnitude more power than an Application-specific Integrated Circuit (ASIC) in the same technology [22].

FPGA power consumption can be reduced by optimizing the architecture of the programmable fabric or the Computer-Aided Design (CAD) algorithms used to map circuits onto the FPGA. Previous work has proposed CAD algorithms that optimize circuit implementations in an attempt to minimize energy, and achieve energy reductions of approximately 23% [10]. This improvement is unlikely to be sufficient to enable FPGA-based hand-held devices, or to significantly reduce the cost of expensive packaging. Significant gains are only possible at the algorithm or system design level. During algorithm or system design, the designer has considerable freedom regarding resource allocation and scheduling; correct decisions here are likely to result in a much larger impact on energy and power consumption than optimizations performed by logic optimization or physical design CAD tools.

One of the simplest but effective ways of reducing the energy per operation of a circuit is pipelining. A highly pipelined circuit suffers fewer glitches than an unpipelined circuit, since it typically has fewer logic levels between registers. Fewer

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 719–728, 2004.

glitches means that less dynamic power is dissipated during each cycle, which reduces the energy per operation.

The ability of pipelining to reduce glitches in ASICs is well-known [15]. In an FPGA, pipelining may be even more effective. Unlike an ASIC, in which signals can be routed using any available silicon, FPGAs implement interconnects using fixed metal tracks and programmable switches. The relative scarcity of programmable switches often forces signals to take longer routes than would be seen in an ASIC or custom integrated circuit. As a result, the potential for unequal delays among signals, and hence the creation of glitches, is likely more than that in an ASIC. Thus, we would expect pipelining to be an effective energy reduction technique for FPGAs.

Another reason that pipelining should work well in FPGAs is that commercial FPGAs contain one or more flip-flops in every logic block. These flip-flops often go unused. Thus, the additional flip-flops required for pipelining are usually "free" in an FPGA. On the other hand, FPGA clock trees are large and consume significant power; pipelining places further demands on the clock tree.

In this paper, we investigate the effectiveness of pipelining on the energy of FPGA circuit implementations, and compare it to the energy improvements that can be obtained by lower-level, power-aware synthesis algorithms. Specifically,

1. we present quantitative measurements of the impact of pipelining on the energy per operation consumed by datapath circuits in both a 0.13µm CMOS high density/speed FPGA and a 0.18µm CMOS low-cost FPGA;
2. we approximate the best possible gains that can be obtained by pipelining/retiming by considering circuits with registers after every logic element;
3. we investigate the interaction between pipelining (a system-level design optimization) and clustering (a lower-level design optimization), and determine how the degree of pipelining affects the effectiveness of the lower-level CAD algorithms in reducing energy.

The results of this study are important for four reasons. First, they provide guidance to a system designer designing FPGA circuits for a low-power application. As we will show, pipelining can reduce the energy per operation by 40% to 90%; this is the kind of reduction needed if FPGAs are to be used in handheld applications. Second, the results provide guidance to FPGA CAD designers by quantifying the effectiveness of high-level and low-level energy optimizations. As we will show in the next section, there has been significant research into power-aware retiming and other related algorithms; our results give an indication of how useful these optimizations can be at reducing energy. Third, pipelining also reduces the time for design tools to estimate power consumption; current vendor tools can take an excessive amount of time for power estimation when dealing with designs with a low degree of pipelining. Finally, there has been work on placing pipeline registers within the interconnect fabric of an FPGA [17]. Our results suggest that, in addition to increasing the clock rate, these registers will also be effective at reducing energy.

2 Preliminaries

Power consumption for circuits in CMOS technology has a static component and a dynamic component. The static power component is mainly due to leakage current. The dynamic power component is mainly due to switching activities for charging and discharging load capacitance. Although static power is becoming increasingly significant, dynamic power still dominates, even in a 0.13µm technology.

It is well-known that undesirable switching activities are usually caused by glitches: spurious pulses at the output of a combinational component due to input signals arriving at different times because of unequal input propagation delays. Techniques have been proposed to reduce such glitches, for instance by restructuring multiplexer networks and inserting selective delays [15], by logic decomposition based on glitch counting and location [4], and by selective gate freezing [1].

Several researchers have applied retiming for power and energy optimization. For example, different pipelined designs can be obtained by adding flip-flops to circuit inputs; such flip-flops can then be relocated by the retiming algorithm to reduce combinational path length and the associated switching activities [13]. Retiming can be combined with supply voltage scaling [3] and with register disabling [6].

Other techniques for reducing switching activities, including loop folding [8] and finite state machine decomposition [14], have also been studied. In addition, the effect of high-level compiler optimizations on system power has been investigated [7].

The power and energy optimizations described above are not developed specifically for FPGAs. FPGA-specific optimization schemes have also been reported. These include boolean optimization of multiple lookup tables [9], perturbation-based word-length optimization [5], implementation of critical loops in configurable logic [19], empirical energy modeling based on surface-curve fitting [16], and combination of power-aware CAD algorithms such as technology mapping and clustering [10].

Many of these optimization techniques rely on the reduction of switching activity by the insertion of registers. Yet the effectiveness of doing this in a modern FPGA has not been quantified. Our work, therefore, is orthogonal to these previous studies, since most of the previous studies rely on a significant improvement in power as a result of glitch reduction. In our work, we quantify what sort of reduction is possible.

The effect of pipeline granularity on reducing power consumption has been examined for Xilinx XC3000 [2], XC4000 and Virtex devices [20]; it is shown that glitch power reduction compensates the synchronization overheads due to pipelining in these small circuits. Our work is different from these previous studies in several ways. First, we consider FPGAs fabricated in modern including a 0.13µm CMOS technology; the power characteristics of integrated circuits implemented using these technologies are very different from those of integrated circuits implemented using older technologies. Second, we consider the pipeline-aware reductions in the context of an entire CAD flow, and examine the interactions between pipelining (a system-level optimization) and clustering, a lower-level power-optimization stage that has been shown to be effective at reducing power.

3 Impact of Pipelining

In this section, we quantify the impact of pipelining on FPGA implementations, first for a high speed and high density 0.13µm CMOS FPGA, and then for a low-cost 0.18µm CMOS FPGA.

3.1 Experimental Methodology

We employ an experimental methodology to investigate pipelining in FPGAs. We use four benchmark circuits; for each circuit, we create several versions, each with a different degree of pipelining. For some circuits, we add pipeline stages by modifying the original hardware description code by hand; for other circuits, we use an automatic synthesis tool [11] that generates circuits with differing degrees of pipelining. In all cases, the function of all versions of each circuit is the same, except for the additional latency imposed by pipeline stages. The first three columns of Table 1 list respectively our benchmark circuits, the number of registers, and the logic depth in each version of the circuit.

For each design, we also create a version with a pipeline stage after every logic element. Unlike all other versions of the circuit, this version is likely to have a different behaviour from the original circuit, since paths containing different numbers of logic elements will have different numbers of registers. The results from this version of each circuit will give an estimate of the best possible optimization achievable using pipelining. To generate these circuits, we use facilities provided through the Quartus University Interface Program (QUIP) which allow us to insert registers after logic synthesis but before place and route. The rows labeled “Max” in Table 1 show the statistics for these circuits. For all but the multiplier circuit, the depth of these “maximally pipelined” circuits is larger than one LE (Logic Element); this is because we do not insert pipeline registers along the carry and cascade chains.

The circuits are implemented on an Altera Nios Development Kit (Stratix Professional Edition) which contains a 0.13µm CMOS Stratix EP1S40F780C5 device. The input pins are not driven externally, since this could consume significant power, possibly dwarfing the on-chip power we want to measure. Instead, we generate our test vectors on-chip using a linear-feedback shift register. For the same reason, we do not drive a pin with each circuit output. Instead, we combine all outputs using a multi-input exclusive-or gate and feed the result to an output pin. We use an exclusive-or gate rather than leaving the outputs floating, to ensure that Quartus does not “optimize away” parts of the circuit not used to drive output pins. The outputs are registered before the exclusive-or gate, preventing glitches appearing at the inputs to the exclusive-or gate. This ensures that the power consumed by the exclusive-or gate is constant across all versions of a circuit. Finally, we register the output of the exclusive-or gate to ensure that glitches on the output pin do not overwhelm the glitch power internal to the FPGA. The clock of each circuit is driven by a scaled version of the 50Mhz on-board oscillator.

3.2 Effect of Pipelining on a 0.13µm FPGA

The final two columns of Table 1 show our measured power results. These are obtained by measuring the current entering the board from the power supply, and multiplying by the power supply voltage. The first of these columns shows the overall board (system) power. This system power includes the power of the board's transformer. In this paper, we are interested in the dynamic power of the FPGA itself. Therefore, we have subtracted the quiescent power when the board is idle, and recorded the results in the final column of Table 1 – we have found that the power dissipated by an idle FPGA and board is independent of the configuration of the FPGA on that board. This difference between the quiescent power and the total power represents the dynamic power of both the FPGA and the board.

In our experimental set-up, we have no way of isolating the dynamic power of the FPGA itself. To estimate this quantity, we have simulated our circuits using the Quartus simulator and power estimator. These simulation results are shown in the sixth column of Table 1. We also record the component of the FPGA power that is due to dynamic switching inside the logic block array; the fifth column of Table 1 shows these measurements.

In gathering the results in Table 1, we use the same clock speed for all versions of a given circuit. By holding the clock rate constant while varying the amount of pipelining, we obtain measurements proportional to the total energy per operation for each circuit. Normally, pipelining is used to increase the clock frequency, thereby increasing the number of operations per second. However, pipelining can also be used to reduce power. As we will show, by pipelining a circuit without changing the clock frequency, power reductions can be achieved without a reduction in operations per second, provided that the additional latency can be tolerated at the system level.

The results from Table 1 are startling. For the 64-bit unsigned multiplier, the difference in dynamic system energy between our most pipelined variant and our least pipelined variant is 81%. For the other benchmark circuits, this difference ranges from 40% to 82%. When we focus on just the dynamic logic block energy, the difference is as high as 98%. In contrast, lower-level physical design optimizations can typically reduce energy by up to 23% [10]. The results in Table 1 show that, indeed, system-level optimizations such as pipelining can have a far more significant impact on the overall energy dissipation.

Table 1 also shows the results for the "maximally pipelined" variant of each benchmark circuit, in which a register is used at the output of every logic block. As the table shows, for all of the benchmark circuits, the energy dissipated by the maximally pipelined variant is smaller than the energy dissipated by all the other versions. However, in three of the circuits, the difference in energy between the maximally pipelined variant and the next-most pipelined variant is small. This implies that there is little opportunity of reducing glitch energy further by increasing the number of pipeline stages in these circuits.

Table 1. Pipelining results for 0.13µm FPGA.

Bench mark Circuit	Number of Pipeline Stages	Number of Registers	Max. Stage Depth (LE's)	FPGA Power Estimate by Quartus		Measured System Power	
				Logic Block Power (mW)	Total FPGA Power (mW)	Total Power (mW)	Dynamic Power (mW)
64-bit integer array mult.	2	361	105	2 289	2 748	9 657	7 659
	4	555	73	1 977	2 436	7 263	5 265
	8	943	57	173	631	5 427	3 429
	16	1 719	49	59.3	512	4 563	2 565
	32	3 271	45	38.5	498	4 041	2 043
	64	6 374	43	38.7	497	3 654	1 656
	Max	15 730	1	86.4	545	3 402	1 404
Triple-DES encrypt. circuit	6	3 823	33	1 225	1 684	4 204	2 206
	12	4 542	17	2 008	2 467	4 041	2 043
	24	5 983	9	365	824	3 015	1 017
	48	9 023	5	144	603	2 718	720
	Max	20 579	2	109	568	2 664	666
8-tap FP FIR filter	1	100	132	Could not estimate		12 645	10 647
	2	353	100	3 951	4 420	7 866	5 868
	4	545	43	1 987	2 468	5 580	3 582
	Max	6 738	31	219	776	3 834	1 836
Cordic circuit	2	2 147	160	Could not estimate		6 507	4 509
	4	3 811	80	512	971	5 139	3 141
	8	6 835	40	152	611	4 437	2 439
	16	13 171	20	105	565	4 716	2 718
	Max	25 191	20	107	567	4 140	2 142

3.3 Effect of Pipelining on a 0.18µm FPGA

The results in Table 1 are obtained using a 0.13µm FPGA. Intuitively, in an FPGA implemented using a less aggressive technology, we would expect a larger component of the overall energy to be dynamic energy, and hence glitch energy. Thus, pipelining may even be more effective. On the other hand, an FPGA implemented in a 0.18µm technology will likely contain fewer logic elements than one in a 0.13µm technology, hence the user circuits will likely be smaller, and thus the potential for glitches may be reduced.

Table 2 shows measured energy results obtained using a 0.18µm FPGA, a Xilinx Spartan XC2S200 device on a Celoxica RC-100 board. Since the chip is smaller than that employed in the previous section, we have scaled down our benchmark circuits by reducing the bit-width in the multiplier, the number of parallel Cordic modules in the Cordic circuit, and the number of taps in the FIR filter. We do not attempt to reduce the size of the triple-DES circuit. As the results show, the impact of pipelining on power – and hence energy per operation – is similar to that for the 0.13µm chip, although less pronounced.

Table 2. Pipelining Results for 0.18μm FPGA.

Benchmark Circuit	Number of Pipeline Stages	Measured System Power	
		Total Power (mW)	Dynamic Power (mW)
16-bit unsigned integer array multiplier	1	5 124	4 116
	2	3 924	2 916
	4	3 312	1 304
	8	3 192	2 184
	16	3 168	2 160
4-tap Floating Point FIR filter	1	10 476	9 468
	2	9 048	8 040
	4	7 800	6 792
Cordic circuit to compute sine and cosine of angle	2	5 777	4 408
	4	4 094	2 727
	8	3 364	1 995
	16	2 888	1 519

4 Interaction Between Pipelining and Clustering

Energy optimization can be performed during high-level system design by techniques such as pipelining, in addition to optimization during low-level synthesis/physical design. Related work has shown that reductions of up to 23% are achievable during synthesis and physical design [10]. In the previous section, we have shown that larger reductions are achievable during system design by pipelining the circuit.

A true power-aware CAD flow would likely attempt to optimize power at both the system design level and the synthesis/physical design level. However, it is conceivable that fewer power reduction opportunities will exist for the lower-level tools, if the higher-level tools are power-aware. In our case, pipelining will tend to reduce the number of nodes with very high switching activities due to glitches, so that there are fewer of these nodes for the physical design tools to optimize. At the extreme, in a very heavily-pipelined circuit, there are no glitches, meaning all nets will have roughly the same activity. This means that the physical design tools will not be able to effectively optimize for power consumption.

In this section, we investigate whether the effectiveness of lower-level tools is affected by pipelining at the system level. We focus on clustering, since it has been shown that clustering is more effective at reducing power than other low-level CAD stages [10]. Commercial FPGAs contain logic elements arranged in clusters; these are known as Logic Array Blocks in Altera parts and Configurable Logic Blocks in Xilinx parts. Each of these clusters contains between two and ten logic elements implemented as lookup-tables. Clustering attempts to pack tightly-connected logic elements together into clusters.

The power-aware cluster algorithm described in [10] attempts to minimize energy by encapsulating high-activity nets within a cluster, so that they can be implemented using low-capacitance intra-cluster connections. It avoids separating the pins of a high-activity net among several clusters such that the net must be implemented on high-capacitance inter-cluster connections. Intuitively, this will be less effective if most nets have similar activities; in this section, we investigate whether this intuition holds.

4.1 Experimental Methodology

We illustrate our methodology by considering the Cordic circuit and the 64-bit integer multiplier, since these two circuits have the largest number of variants. Each circuit is first optimized and mapped to lookup-tables by Quartus. Then, using facilities provided in the Quartus University Interface Program (QUIP), we feed each technology-mapped netlist into both the power-aware cluster algorithm described in [10] and the non-power-aware cluster algorithm described in [12]. Next, each of these clustered circuits is read back into Quartus, placed and routed, and implemented on the FPGA. The power consumption, which is proportional to energy per operation, is then measured as in Section 3.

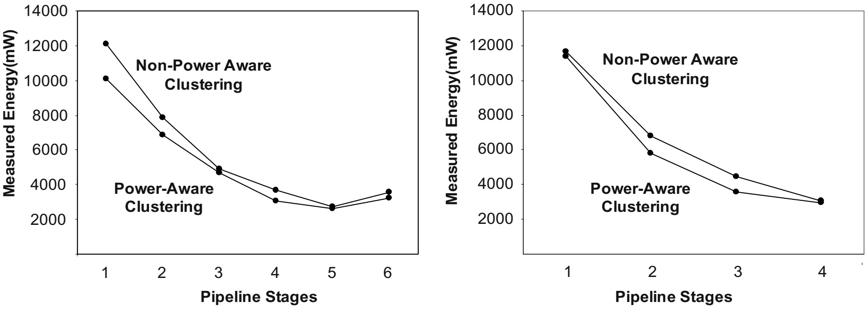
Note that neither of the clustering algorithms in [10] and [12] uses carry chains or cascade chains. As a result, it is not meaningful to compare these results with the results in Table 1 which are obtained using the Quartus clusterer, since those results do employ carry and cascade chains where appropriate. Nonetheless, the trends observed in this section will likely hold in a carry/cascade chain-capable clusterer.

4.2 Results

Figure 1 shows the results for the array multiplier and the Cordic circuits. The horizontal axis on each graph is the number of pipeline stages, and the vertical axis is the measured system (board) dynamic power, which is proportional to the energy per operation of the circuit. The top line in each graph represents the energy obtained when using the non-power-aware cluster algorithm, while the lower line represents the energy obtained when using the power-aware cluster algorithm. As the graphs show, the improvements obtained by pipelining are much more significant than those obtained by making the cluster algorithm power-aware.

The graph also illustrates the interaction between the two optimization schemes: the reduction achieved by the cluster algorithm varies as the degree of pipelining changes. In general, for the array multiplier, the reduction achieved by the cluster algorithm decreases as the degree of pipelining increases. For both circuits, the power-aware cluster algorithm is ineffective at reducing power for the most heavily-pipelined variants, since it has fewer high activity nets to work with.

These results are significant. They indicate that for most circuits, it does make sense to optimize for power both at the system level as well as during low-level synthesis and physical design. The results also show, however, that it is not reasonable for a designer to rely on pipeline-aware synthesis and physical design. Incorrect system-level design decisions, such as a bad choice for pipelining depth, can not be “made up for” by low-level CAD tools.



a) 64-bit integer multiplier

b) Cordic circuit

Fig. 1. Power-aware and non-power aware clustering results.

5 Conclusions

In this paper, we have shown that pipelining has a significant impact on the energy dissipation in an FPGA. Pipelining reduces the number of spurious glitches which, in turn, reduces dynamic power. The primary contribution of this paper is the quantification of just how effective this technique can be in modern FPGAs. Among our four benchmark circuits, we have found that pipelining can reduce the amount of energy per operation required by an algorithm by between 40% and 90%.

These results are important for a number of reasons. For system designers, it illustrates the need for careful planning of a datapath and pipelining during system design. No matter how good the subsequent power-aware CAD tools are, it is unlikely that they will be able to “make up for” a bad decision during system design. Although designers have been aware of this before, our results suggest that correct pipelining is especially critical in modern FPGAs. For the CAD research community, these results suggest that more attention needs to be paid to high-level system-level optimizations, such as pipelining. Traditional power-aware optimization studies focus on lower-level technology mapping, clustering, or physical design. The gains achievable at these low level are important, but they will not be enough to make handheld FPGA devices a reality. On the other hand, the significant energy improvements shown in this paper can make FPGAs appropriate for a much larger class of low-power applications than ever before.

References

1. L. Benini et al, Glitch power minimization by selective gate freezing, IEEE Trans. VLSI Systems, 8(3): 287-298, 2000.
2. E. I. Boemo et al, Some notes on power management on FPGA based systems, Field Programmable Logic and Applciations, LNCS 975, Springer, 1995, pp. 149-157.
3. N. Chabini et al, Unification of basic retiming and supply voltage scaling to minimize dynamic power consumption for synchronous digital designs, Proc. ACM Great Lakes Symposium on VLSI, 2003.

4. K. S. Chung, T. Kim and C. L. Liu, A complete model for glitch analysis in logic circuits. Journal of Circuits, Systems, and Computers, 11(2): 137-154, 2002.
5. G. A. Constantinides, Perturbation analysis for word-length optimization, Proc. Int. Symp. field-Programmable Custom Computing Machines, 2003, pp. 81-90.
6. Y. L. Hsu and S. J. Wang, Retiming-based logic synthesis for low power, Proc. Int. Symp. Low Power Electronics and Design, ACM Press, 2002, pp. 275-278.
7. M. Kandemir et al, Influence of compiler optimizations on system power, IEEE Trans. VLSI, 9(6):801-804, 2001.
8. D. Kim and K. Choi, Power conscious high level synthesis using loop folding, Proc. 34th Design Automation Conference, 1997.
9. B. Kumthekar et al, Power optimization of FPGA-based designs without rewiring, IEE Proc., 147(3): 167-174, 2002.
10. J. Lamoureux and S. Wilton, On the interaction between power-aware FPGA CAD algorithms, Proc. ICCAD, 2003.
11. W. Luk et al, Parameterized hardware libraries for configurable system-on-chip technology, Canadian Journal of Elect. and Computer Engineering, 26(3/4):125-129, 2001.
12. A. Marquardt, V. Betz and J. Rose, Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density, ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays, Feb. 1999, pp. 37 - 46.
13. J. C. Monteiro, S. Devadas and A. Ghosh, Retiming sequential circuits for low power, Proc. ICCAD, pp. 398-402, 1993.
14. J. C. Monteiro and A. L. Oliveira, Finite state machine decomposition for low power, Proc. 35th Design Automation Conference, 1998.
15. A. Raghunathan, S. Dey and N. K. Jia, Register transfer level power optimization with emphasis on glitch analysis and reduction, IEEE Trans. CAD, 18(8):114-1131, 1999.
16. L. M. Reyneri et al, A hardware/software co-design flow and IP library based on Simulink, Proc. 38th Design Automation Conference, 2001.
17. A. Singh et al, Interconnect pipelining in a throughput-intensive FPGA architecture, ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays, 2001, pp 153-160.
18. D. Singh and S. Brown, The case for registered routing switches in Field Programmable Gate Arrays, ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays, 2001, pp. 161-169.
19. G. Stitt et al, Using on-chip configurable logic to reduce embedded system software energy, Proc. Int. Symp. Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2002, pp. 143-151.
20. G. Sutter et al, Logic depth, power, and pipeline granularity: updated results on XC4K and Virtex FPGAs, Computacion Reconfigurable & FPGAs, Publicaciones Digitales S.A., 2003, pp. 201-207.
21. W. Tsu, et al, HSRA: High-speed, hierarchical synchronous reconfigurable array, ACM Seventh International Symposium on Field-Programmable Gate Arrays, Feb. 1999.
22. P. Zuchowski et al, A hybrid ASIC and FPGA architecture, Proc. ICCAD, 2002, pp. 187-194.

A Methodology for Energy Efficient FPGA Designs Using Malleable Algorithms*

Jingzhao Ou and Viktor K. Prasanna

Department of Electrical Engineering, University of Southern California
Los Angeles, California, 90089-2560 USA
{ouj,prasanna}@usc.edu

Abstract. A recent trend towards integrating FPGAs with many heterogeneous components, such as memory systems, dedicated multipliers, etc., has made them an attractive option for implementing many embedded systems. Paradoxically, the integration that makes modern FPGAs an attractive computing substrate also makes the development of energy efficient FPGA designs very challenging in practice. This is due to the many alternatives available for implementing a desired functionality and a lack of high-level models of FPGA architectures that can accurately capture the energy dissipation behavior of alternatives. To address these issues, we propose a methodology for energy efficient FPGA designs using malleable algorithms. Malleable algorithms are used to expose the architecture-platform aware specifications of alternate implementations of the desired functionalities. Our methodology consists of three major design steps: domain-specific energy performance modeling, development of malleable algorithms, and system-level optimization. Energy efficient designs are realized through close interaction among these three design steps. To illustrate the proposed design methodology and demonstrate the benefits of designing using malleable algorithms, we present the development of a beamforming application through a high-level MATLAB/Simulink based FPGA design tool developed by us. By tuning the design knobs exposed by malleable algorithms, the design of the beamforming application identified through system-level optimization achieves up to 30% energy reduction compared with other designs considered in our experiments.

1 Introduction

Increasing density and integration of various hardware components, such as embedded multipliers, memory blocks, and RISC processors, etc., have made FPGAs an attractive option for implementing many embedded systems. Besides, with the proliferation of portable and mobile devices, energy efficiency has become increasingly important in the design of these embedded systems .

Paradoxically, the integration of heterogeneous components that makes modern FPGAs an attractive computing substrate makes energy efficient FPGA designs very difficult in practice. The main reason is that the progress in the design automation at VLSI level has outpaced the corresponding evolution of high-level abstractions that allow a

* This work is supported by the United States National Science Foundation (NSF) under award No. CCR-0311823.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 729–739, 2004.

designer to develop applications on such heterogeneous devices. Also, rapid and accurate energy estimation is challenging for FPGA designs. On the one hand, energy estimation using RTL (Register Transfer Level) simulation (which can be accurate) is too time consuming and can be overwhelming considering the fact that there are usually many possible implementations of an application on FPGAs. On the other hand, the basic building blocks of FPGAs are look-up tables (LUTs), which are too low-level an entity to be considered for high-level modeling and rapid energy estimation. No single high-level model can capture the energy dissipation behavior of all possible implementations on FPGAs. Lacking such a rapid energy estimation technique would further prevent optimizing the energy performance of the complete applications.

In this paper, we address the following design problem. The target application is assumed to be decomposed into a set of kernels with precedence relations between them. Various hardware implementations of these kernels on the target FPGA device are also given. Our goals are: (1) to find appropriate high-level abstractions of the given implementations which enable rapid and accurate energy performance estimation of a design instance; (2) to traverse the design space populated through the high-level abstractions and identify energy efficient design(s).

There are two main contributions of this paper. We propose the concept of *malleable algorithms*, which are architecture-platform aware specifications of alternate implementations of a given functionality. Details of malleable algorithms are further discussed in Section 3. The other contribution is a design methodology for energy efficient FPGA designs based on malleable algorithms. There are three main thrusts in our design methodology. First, *domain-specific modeling* is used to derive analytical models that capture the high-level energy dissipation behavior of the alternate implementations of the kernels. In [3], we have shown that the energy performance of these implementations can be obtained rapidly and fairly accurately using these analytical models. Second, *malleable algorithms* are developed to encapsulate the various implementations of the kernel as well as the analytical models obtained using domain-specific modeling. Finally, *system-level optimization* is performed by tuning the design knobs exposed by malleable algorithms to identify energy efficient designs of the target application.

For the sake of illustration, we provide an implementation of our design methodology based on a start-of-the-art MATLAB/Simulink based high-level design tool. We design a beamforming application using this tool. By tuning the design knobs exposed by malleable algorithms, the design of the beamforming application identified through system-level optimization achieves up to 30% energy reduction compared with other designs considered in our experiments.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces the concept of *malleable algorithms*. Section 4 presents our design methodology based on malleable algorithms. Section 5 discusses an implementation of the design methodology using a high-level MATLAB/Simulink based FPGA design tool developed by us. In Section 6, we demonstrate the development of an adaptive beamforming application using malleable algorithms. We conclude in Section 7.

2 Related Work

The Carte compiler [11] contains a library for FPGA implementations of various functionalities, which is used in the hardware and software compilation processes for their computers. These implementations are represented as RTL netlist files. Different implementations of the same functionality are not captured by their library. Also, energy performance of the various FPGA implementations is not captured.

High-level design tools are becoming popular for designing using FPGAs. One important kind of tools are those based on MATLAB/Simulink, such as *DSP Builder* [1] from Altera and *System Generator* [12] from Xilinx. These tools provide a high-level (arithmetic level) abstraction of the underlying hardware resources and allow the application designers to describe the data flow and its hardware realization directly through this high-level abstraction. While resource utilization can be obtained rapidly using these MATLAB/Simulink based tool, no energy performance information is associated with the high-level abstraction provided by these tools. Besides, systematic traversal of the design space and identification of energy efficient designs are also not supported by the current versions of the tools.

Other high-level design tools are such as DK2 [4] from Celoxica and Forge [12] from Xilinx which use high-level languages such as C and Java for designing using FPGAs. When using these tools, the application designers describe their applications using C or Java and rely on the compiler to infer the appropriate architecture for implementing the application and to perform optimizations such as loop unrolling, pipelining, etc. The output of these tools is either HDL code or EDIF netlist. While these system level design tools have proved to be capable of simplifying the design complexity and reducing the design time, the compilers used by these tools do not optimize the energy performance of the generated FPGA designs.

3 Malleable Algorithms

First of all, we assume that there are two roles, an algorithm designer and an end user, involved in the design process. An *algorithm designer* is concerned with providing the end user with efficient implementations of the set of kernel functionalities that constitute the complete application. For the adaptive beamforming application discussed in Section 6, Levinson Durbin recursion and FFT are examples of such kernels. An *end user* is concerned with designing and implementing an “efficient” hardware solution for an application, such as the beamforming application discussed in Section 6. Energy dissipation, latency, throughput, etc., are some of the considerations from the end user’s perspective. Also, we define a *platform* as a device provided by a vendor which consists of a set of hardware components for processing, storage, and interconnection. In this paper, we focus on platforms with FPGA as the major component on the device. The Xilinx Virtex-II Pro [12] is such an example platform we are targeting.

Based on the above assumptions and definitions, a *malleable algorithm* is defined as an architecture-platform aware specification of alternate implementations of a given functionality. The availability of such alternatives is due to the large amount of programmable resources and heterogeneous embedded components found on many modern FPGA devices (e.g. the platforms of interest), which provide a very high flexibility in the hardware implementation. Most importantly, these alternatives would result in different amounts of energy dissipation for the execution of the same functionality. For example, there are three possible hardware bindings for implementing storage elements on Virtex-II Pro, which are registers, slice based RAMs, and embedded Block RAMs (BRAMs). Our work in [2] shows that registers and slice based RAMs have better energy efficiency for implementing small amount of storage while BRAMs have better energy efficiency for implementing large amount of storage. Another example is that matrix multiplication can be implemented using many architectures including a linear array or a 2-D array. A 2-D array implementation uses more interconnects and can result in more energy dissipation compared with a linear array implementation.

A malleable algorithm captures two types of knobs, one describing the high-level functionalities while the other describing the alternative implementations of the high-level functionalities on a specific platform. These knobs are identified by the algorithm designer and are made available to the end user. For example, when developing an FFT kernel, we identify two kinds of knobs as shown in Table 1 (see Section 6.2). Then, in the system-level optimization discussed in Section 6.3, we enumerate these knobs to evaluate the possible implementations of an beamforming application and identify an energy efficient design for this application. Therefore, malleable algorithms play a key role in the interaction between the algorithm designer and the end user. They encapsulate the algorithm designers' understanding of performance modeling and *potential optimization knobs* that may or may not be exploited by the end user. By handling the knobs exposed by malleable algorithms at high-level, the end user can use an available optimization technique to realize the final implementation.

Besides, malleable algorithms make it possible to model FPGA platforms at high-level and thus enable rapid and accurate energy estimation for various implementations on a specific FPGA platform. This is because malleable algorithms relate the performance of the various implementations on a specific platform to a set of knobs identified by the algorithm designer. By restricting the modeling to a well-defined algorithm and architecture, techniques such as domain-specific modeling (see Section 4) can be applied to provide high-level abstractions of the low-level RTL implementations. Our experiments have shown that such careful abstractions can lead to rapid and fairly accurate energy estimation. In this scenario, malleable algorithms can be used to encapsulate the domain-based knowledge of the kernels identified by the algorithm designer and expose them to the end user for optimizing the energy performance of the complete application.

Malleable algorithms are loosely analogous to parameterized soft-IP (Intellectual Property) cores provided by EDA vendors that are instantiated by the end user at logic synthesis time. However, one of the significant differences between a malleable algorithm and a soft-IP core is that the former is defined at a higher level of abstraction than the latter, which is typically designed and optimized at the register transfer level. Such high-level abstractions would bring two major advantages to the designer. One is that it would not be possible to encapsulate all of these optimization possibilities into a single IP core described at the register transfer level. However, by using architecture-level abstractions, a single malleable algorithm is able to express these optimization possibilities. This ability makes malleable algorithms much more flexible than state-of-the-art parameterized soft-IP cores. Another advantage is that the high-level abstractions developed using our techniques enable fast high-level simulation. For example, the arithmetic level simulation of the IP cores within MATLAB/Simulink using the high-level representations provided by *System Generator* are usually much faster than the behavioral and architectural simulations in traditional FPGA design flows [12]. Most importantly, we have shown in [5] that some important low-level parameters such as switching activities, etc. can be predicted through such high-level simulation. These predicted parameters are crucial for obtaining rapid and accurate energy estimation.

Platform based designs are proposed by Keutzer *et al.* [8] for application development on SoCs (System-on-Chips). In their approach, a *platform* is defined as a layer of abstraction with two views. The upper view allows an application to be developed without referring to the lower levels of abstraction. The lower view is a set of rules that classify a set of components belonging to the platform. Platform based designs distill the complex world of system-level design into a few core concepts that have the potential of making the design process more manageable and efficient. However, to the best of our knowledge, no concrete methodology has been proposed to facilitate *algorithm design*. Here, algorithm design is defined as the use of an "exposed" model of the architecture platform to specify alternate implementations of a specific functionality. In our design methodology, a malleable algorithm is used to represent such an "exposed" model. The alternatives of the given functionality are captured by the high-level and low-level design parameters associated with the malleable algorithm while their performance can be calculated rapidly using the performance models specified by the malleable algorithm. Given a target application consisting of a set of kernels, system-level optimization is performed based on the malleable algorithms for the kernels.

4 Design Methodology

The application is decomposed into a number of kernels. We consider an application graph as input to our design methodology, which describes the communication and precedence relationships between the kernels. Based on the application graph, the proposed design methodology for energy efficient FPGA designs is shown in Figure 2(a).

In our design methodology, various alternative implementations of the kernels are first developed, which provide design trade-offs regarding energy, area and time. Using domain-specific modeling, analytical energy models are derived, which can be used to quickly estimate the performance of these implementations. By combining the implementations with corresponding performance models, malleable algorithms are developed for the kernels. Then, the end user describes the target application using the developed malleable algorithms. Finally, system-level optimization is performed based on this high-level description and identifies energy efficient design(s). The three major design steps (the shaded boxes in Figure 2(a)) are further explained in the following.

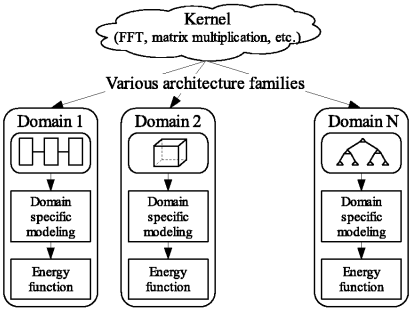


Fig. 1. Domain-specific modeling

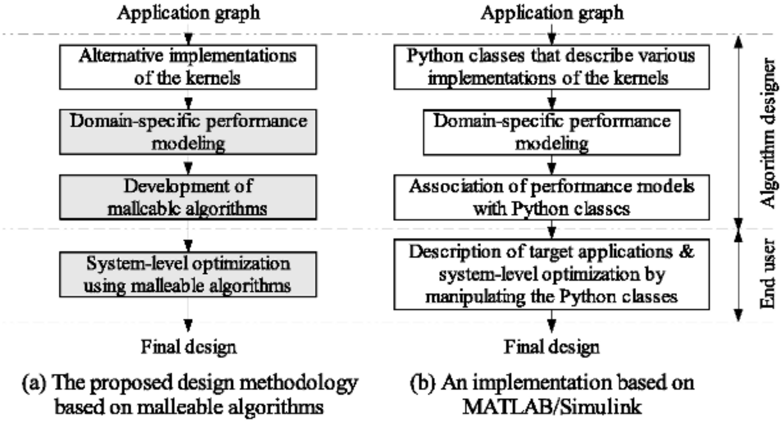

Application graph
Alternative implementations of the kernels
Domain-specific performance modeling
Development of malleable algorithms
System-level optimization using malleable algorithms
Final design
(a) The proposed design methodology based on malleable algorithms
Application graph
Python classes that describe various implementations of the kernels
Domain-specific performance modeling
Association of performance models with Python classes
Description of target applications & system-level optimization by manipulating the Python classes
Final design
(b) An implementation based on MATLAB/Simulink
Algorithm designer
End user

Fig. 2. Design flows

• **Domain-specific energy performance modeling:** Domain-specific modeling is a hybrid approach (top-down followed by bottom up) toward performance modeling for kernels. While more details about this technique can be found in [3], we summarize it here for the sake of completeness.

As shown in Figure 1, we group implementations of a kernel based on their architectures and algorithm families. By doing so, we impose a high-level architecture onto the FPGA implementations within each domain. For a given kernel within each domain, performance models are first defined analytically as energy functions of the design parameters of the kernel. Low level simulations are then performed to estimate the constants in the analytical models for a specific hardware platform. The performance of these implementations, such as energy dissipation, latency, etc., can be calculated rapidly using these (analytical) performance models. We have developed several kernels such as matrix multiplication, matrix factorization, etc. and have shown that domain-specific modeling can lead to an average estimation error around 10% for these kernels [5].

• **Development of malleable algorithms:** The first step in developing malleable algorithms is to associate the various implementations of the kernels with the analytical models developed using domain-specific modeling. In addition, malleable algorithms expose the knobs that describe the high-level functionalities of the kernels as well as those that control the low level realization. These knobs are identified through domain-specific modeling and based on the possible application requirements. They are accessible to the end user during system-level optimization. One way of developing malleable algorithms is to use the object-oriented mechanism to perform such encapsulation and exposition as illustrated in Section 5.

• **System-level optimization using malleable algorithms:** Given an end-to-end application graph and a set of malleable kernel algorithms for the kernels, the application is described using the malleable algorithms. Basically, this includes describing the interface between the kernels and the design constraints. Then, system-level optimization consists of exploring the design space exposed by the malleable algorithms and identifying the settings of the algorithms which lead to energy efficient designs. Various combinatorial algorithms can be applied in the optimization process. For example, considering the beamforming algorithm discussed in Section 6.3, the application graph can be described as a linear pipeline. A dynamic programming based algorithm proposed in [6] can be used to find a design with minimum energy dissipation for executing one instance of the application.

The proposed design methodology is accomplished through the interactions between algorithm designers and end users. According to their roles discussed in Section 3, for the design flow shown in Figure 2(a), the algorithm designer is responsible for developing the various implementations of the kernels, using domain-specific modeling to analyze their energy performance, developing malleable algorithms by combining these information while the end user is responsible for system-level optimization using the malleable algorithms developed by the algorithm designer.

5 An Implementation of the Design Methodology Based on MATLAB/Simulink

To illustrate the development of malleable algorithms and our design methodology, we enhance *System Generator* [12], a MATLAB/Simulink based high-level FPGA design tool. This results in an add-on tool called *PyGen*, which provides additional functionalities. By creating an interface between Python and the MATLAB/Simulink based system level design tools, our tool allows the use of Python language [10] for describing FPGA designs within the high-level modeling environment provided by MATLAB/Simulink.

As shown in Figure 3, each block in the *System Generator* block set is mapped to the corresponding basic Python classes within the *PyGen* class library. By manipulating the basic Python classes, the designer can derive their own extended classes (the shaded blocks in Figure 3) and develop parameterized designs for the kernels.

We use *PyGen* to support the development of malleable algorithms. This is realized using the object-oriented mechanism to encapsulate the various implementations of the kernels as well as the energy performance models obtained through domain-specific modeling. As shown in Figure 4, the Python classes encapsulate (1) high-level functionalities of the kernels which can be for high-level simulation in MATLAB/Simulink; (2) exposed hardware design knobs which control the generated low-level implementations on the target device; (3) performance models for rapid energy estimation. See [5] for more details on the software architecture of *PyGen*.

The design flow of the proposed design methodology realized using *PyGen* is shown in Figure 2(b). Various implementations of the kernels are described as Python classes. After organizing the class hierarchy and classifying the classes into different domains, domain-specific modeling is performed within each domain to derive high-level performance models. Malleable algorithms are then developed by associating these performance models with the Python classes. The identified design knobs are exposed as data attributes of the Python classes. Finally, description of the target applications and system-level optimization are performed by manipulating these Python classes.

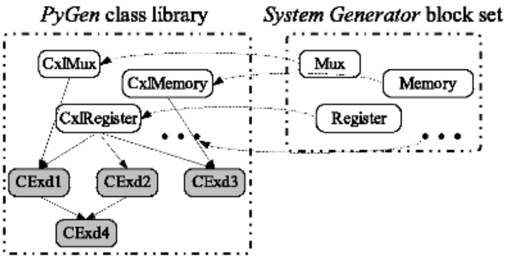


Fig. 3. Python class library within *PyGen*

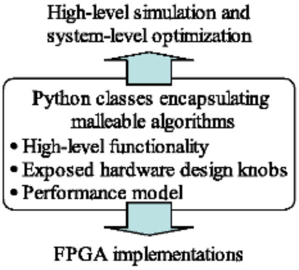

High-level simulation and system-level optimization
Python classes encapsulating malleable algorithms
• High-level functionality
• Exposed hardware design knobs
• Performance model
FPGA implementations

Fig. 4. Development of malleable algorithms in *PyGen*

6 An Illustrative Example

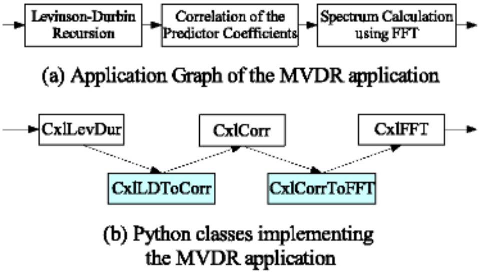
To illustrate the proposed design methodology and demonstrate its effectiveness in developing energy efficient FPGA designs, we show the process of developing a beamforming application. Adaptive beamforming is used by many telecommunication systems such as software defined radio systems for better utilization of the limited radio spectrum. Energy efficiency is an important metric when implementing this application as these systems are usually battery operated.

6.1 MVDR Adaptive Beamforming

We consider a fast MVDR (Minimum Variance Distortionless Response) spectrum calculation application, which is described in [7]. This application uses Levinson Durbin recursion and saves much computation that is otherwise required by the direct calculation of the spectrum. It is part of the MVDR adaptive beamforming process. The application graph of the beamforming application is shown in Figure 5(a), which consists of three kernels: Levinson Durbin recursion, correlation of the predictor coefficients, and spectrum calculation using FFT.

6.2 Development of Malleable Algorithms for the Kernels

First of all, the algorithm designer develops malleable algorithms for the kernels that compose the beamforming application. A malleable algorithm of the spectrum calculation kernel using FFT is developed as a Python class *CxlFFT*. It contains two types of parameters. (1) *High-level functional parameters* describe the functionalities of the Python class. They are identified by considering the application requirements (see Section 6.3). (2) *Low-level architecture parameters* describe the hardware design knobs identified through domain-specific modeling on Virtex-II Pro, our target FPGA platform. All the parameters are associated with the corresponding data attributes of the Python class, which are shown in Table 1. When this Python class is instantiated with the specific design parameters, *PyGen* will generate the corresponding hardware implementation. A performance model is obtained using the domain-specific modeling technique as described in Section 4. This model is associated with the corresponding data attributes of the Python class. Rapid power estimation can be performed using this performance model and the switching activity information estimated through high-level simulation within MATLAB/Simulink. An average estimation error of 6% is observed by comparing with the results from low-level RTL simulation. See [5] for more details on the various implementations of the FFT kernel, the derivation of the performance models and the experimental results.


Levinson-Durbin Recursion
Correlation of the Predictor Coefficients
Spectrum Calculation using FFT
CxlLevDur
CxlCorr
CxlFFT
CxlLDToCorr
CxlCorrToFFT

(a) Application Graph of the MVDR application

(b) Python classes implementing the MVDR application

Fig. 5. MVDR beamforming

Using a similar approach, we develop malleable algorithms for the other two kernels, which are represented by Python classes *CxlLevDur* and *CxlCorr*. The design parameters captured by these two classes are: *Frq* (operating frequency), *M* (number of antenna elements in the system), *degPar* (degree of parallelism), and *Precision* (precision of the data). Details of the development process of these malleable algorithms are not shown due to space limitation.

Table 1. Data attributes of Python class *CxlFFT* for an FFT kernel

High-level functional parameters	Low-level architecture parameters
Frq (operating frequency)	*Arch* (architecture: unfolded or folded)
nPnt (number of frequency points)	*Sto* (hardware binding of storage elements: registers, slice-based RAM or Block RAM)
Precision (data precision, number of bits)	*degPar* (degree of parallelism)

6.3 System-Level Optimization Using Malleable Algorithms

Using the malleable algorithms developed in Section 6.2, the end user describes the target application and performs system-level optimization. As shown in Figure 5(b), description of the beamforming application uses the Python classes that implement the three kernels. In addition, two interfacing Python classes, *CxlLDToCurr* and *CxlCorrToFFT*, are developed to describe the data communication between the kernels, such as the input/output data patterns and the buffering requirements, etc. Description of design constraints are performed by writing Python code to manipulate these Python classes. Then, the correctness of the designs described in Python classes can be verified by instantiating them with appropriate design parameters, generating corresponding Simulink models, and performing arithmetic level simulation in MATLAB/Simulink.

Since the beamforming application can be described as a linear pipeline of kernels, the dynamic programing algorithm proposed in [6] can be applied to find a design for this application so that the energy dissipation for executing one data sample is minimized. First, we create a trellis. The nodes on the trellis represent the various implementations of the kernels. They are obtained by enumerating the design knobs exposed by the corresponding Python classes implementing these kernels while satisfying the application requirements. Taking the FFT kernel as an example, while *Frg*, *nPnt* and *Precision* are set according to application requirements, we enumerate the various possible combinations of *Arch*, *Sto* and *degPar* and obtain five nodes on the trellis representing different possible implementations of this kernel. The weights of these nodes are the costs of executing the kernels using the corresponding implementations represented by them. The weights can be obtained by querying the performance models encapsulated by the Python classes. Besides, there are edges between the nodes which describe the communication between the kernels. Similarly, the weights of the edges can be obtained from the Python classes that implement the communication channels between the kernels. Thus, the identification of the energy efficient design is formulated as an optimization problem, which is to find a traversing path on the trellis with minimum energy cost. The dynamic programming algorithm identifies such a path in an iterative manner. See [6] for more details on the problem formulation and the optimization algorithms.

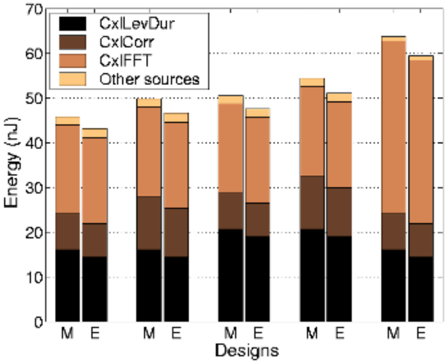


Fig. 6. Energy performance of various designs of the MVDR application (*M* denotes measured data; *E* denotes estimated data obtained by querying the performance models associated with the Python classes)

To show the effectiveness of the proposed design methodology, we perform exhaustive search and identify five designs with lowest energy dissipation according to their energy performance obtained through low-level RTL simulation. We also traverse the MATLAB/Simulink design space using our *PyGen* tool and identify five designs with lowest energy dissipation using the performance models associated with the Python classes. It turns out that these five designs are the same as the five designs identified through low-level simulation. Figure 6 shows the measured (through low-level simulation) and estimated energy performance of these designs. For these five designs, the identified design (e.g. the left most design shown in Figure 6) achieves an energy reduction of up to 30% compared with the other designs considered. For this identified design, the FFT kernel is implemented with *Arch* = *unfolded*. The FFT implementation using an unfolded architecture occupies more slices than the one using a folded architecture. The implementation using an unfolded architecture has less control and storage overheads and thus improves the energy efficiency of the complete application.

7 Conclusion

The concept of malleable algorithms was proposed. We presented a design methodology based on this concept for developing energy efficient applications on FPGAs that contain heterogeneous components. We are currently exploring the potential of malleable algorithms for application synthesis using FPGAs that integrate embedded processors.

References

1. Altera, Inc., `http://www.altera.com`.
2. S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang, "Energy Efficient Signal Processing Using FPGAs," *ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2003.
3. S. Choi, J.-W. Jang, S. Mohanty, V. K. Prasanna, "Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures," *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2002.
4. DK2, Celoxica, Inc., `http://www.celoxica.com/products/tools/dk.asp`.
5. J. Ou and V. K. Prasanna, "PyGen: A MATLAB/Simulink based Tool for Synthesizing Parameterized and Energy Efficient Designs Using FPGAs," *Field-Programmable Custom Computing Machines (FCCM)*, 2004.
6. J. Ou, S. Choi, and V. K. Prasanna, "Energy-Efficient Hardware/Software Co-Synthesis for a Class of Applications on Reconfigurable SoCs," *Int. Journal of Embedded Systems*, 2004.
7. S. Haykin, "Adaptive Filter Theory," 3rd Edition, *Prentice Hall*, 1991.
8. K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. on CAD*, 19(12), Dec. 2000.
9. Mentor Graphics, Inc., `www.mentor.com`.
10. Python, `http://www.python.org`.
11. "Carte Programming Environment," online available at `http://www.srccomp.com/`.
12. Xilinx, Inc., `http://www.xilinx.com`.

Power-Driven Design Partitioning

Rajarshi Mukherjee and Seda Ogrenci Memik

Electrical and Computer Engineering, Northwestern University
{rajarshi, seda}@ece.northwestern.edu

Abstract. In order to enable efficient integration of FPGAs into cost effective and reliable high-performance systems as well potentially into low power mobile systems, their power efficiency needs to be improved. In this paper, we propose a power management scheme for FPGAs centered on a power-driven partitioning technique. Our power-driven partitioner creates clusters within a design such that within individual clusters, power consumption can be improved via voltage scaling. We tested the effectiveness of our approach on a set of LUT-level benchmark netlists. Further we did constrained placement of the clusters into predefined V_{dd}^{high} and V_{dd}^{low} regions for a single FPGA. Average savings in power consumption with our approach is 48% whereas penalty in channel width and wire length due to constrained placement is 23% and 26% respectively.

1 Introduction

Despite exponential improvements in logic density and performance, energy efficiency of the FPGA technology did not keep up. As heat dissipation becomes an increasingly important concern for wired systems, power consumption of FPGAs needs to match more stringent standards in order to ensure the performance goals and reliability. Similarly, inefficiency in power consumption poses a major obstacle to inclusion of FPGAs in many emerging low power mobile systems.

In this paper, we present a high-level power management methodology for FPGAs. We propose a power-driven partitioning technique that identifies partitions in a design, which can be implemented using lower supply voltage levels. Through voltage scaling we allow longer delay for parts of a design while the overall latency of the design is unchanged. In every design, based on the overall timing constraint, there is a set of *critical* nodes/operations/gates, while the remainder of the design is *non-critical*. Hence, it is possible to identify the inherent time slack possessed by individual building blocks in a design. We analyze this design metric and systematically exploit potential relaxation in timing constraints in order to create opportunities for voltage scaling. Our techniques can be utilized for various FPGA-based systems. In multi-FPGA systems applications are partitioned among several devices. Using our partitioning technique voltage scaling can be applied at chip level, using different supply voltages for different devices. While mapping a design onto a single FPGA chip, the design is partitioned such that portions identified by our partitioning technique can be placed within voltage islands at different V_{dd} levels on the same chip.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 740–750, 2004.

Finally, for dynamically reconfigurable systems, the voltage supply level can be dynamically adjusted and our partitioning technique can be used to create configuration contexts such that lowering of the supply voltage is feasible for some of the partitions.

Our specific contributions in this paper are as follows:

- We propose a partitioning algorithm that effectively exploits *relaxable* timing constraints within a design to trade-off delay against lower supply voltage levels,
- We present experimental results using LUT-level netlists to demonstrate the improvement in power using our proposed technique, and
- We present an experimental evaluation of the impact of creating voltage islands on the physical design stages.

The rest of the paper is organized as follows. In Section 2 we present an overview of related work. Section 3 presents the problem formulation and our algorithm. Our experimental setup and results will be presented in Section 4. We conclude with a summary in Section 0.

2 Related Work

In the past, various proposed synthesis techniques addressed the problem of power optimization by improving metrics such as switching activity of a schedule or binding, total used logic resources, interconnect, etc. to generate the most efficient circuit in terms of power consumption [1], [2], [3], [4], [5]. In our approach we are addressing a different power optimization paradigm, namely voltage scaling. Our work is complementary to the above-mentioned techniques.

Partitioning has been studied for multi-FPGA systems and for dynamically reconfigurable systems [6], [7], [8], [9]. The main objectives for optimization have been traditionally cut cost, i.e., the number of connections between partitions, and the number of partitions. In this work, we propose a partitioning scheme that addresses a new objective, namely availability of time slack in a partition. We investigate the potential impact of partitioning on power. Depending on the particular application of our technique, we take other partitioning objectives into account as well.

Voltage scaling is a well-known tool for improving energy efficiency of electronic systems. It has found applications for a wide variety of circuit technologies and design styles including microprocessors, ASICs and real time embedded systems [10], [11], [12], [13]. In this work, we make use of this general optimization technique and investigate its application on FPGA-based systems. Investigation of circuit level issues to implement voltage scaling is beyond the scope of this paper. However, the feasibility of implementing the necessary hardware for voltage scaling is evident considering the successful implementations in other technologies. In addition, Chen et al. reported recent results on the feasibility of dual supply voltage FPGA fabrics [14], [15].

3 Power Management Using Voltage Scaling

In this section, we will formulate our power management problem and discuss two applications of our proposed technique. Next, we will describe our power-driven partitioning algorithm in detail.

3.1 Problem Formulation

We assume that a LUT-level netlist is represented with a *Directed Acyclic Graph (DAG)*. The longest path from any of the primary inputs to any of the primary outputs defines the longest combinational path, i.e., the *critical path* in the design. Logic blocks that reside on the critical path are called *critical nodes*. The rest is referred to as *non-critical* nodes. Taking the length of the critical path as our timing constraint and assuming that all input signals arrive at the same time, we can assign *arrival* and *required* times for each node. The difference between the required time and the arrival time of a node is called *time slack*. This entity will be equal to zero for critical nodes, while it takes a positive value for non-critical nodes. Note that we are estimating the time slack of each LUT at a high level. Naturally, the timing behavior of a design will highly depend on the net delays, hence, on placement and routing. An accurate estimation of interconnect delay cannot be made before physical synthesis. Being aware of this fact, we will evaluate the impact of placement and routing. Details of this study will be presented in Section 4.3. We would like to stress once again that time slack as defined above is the best estimation we can have at this early stage of the design flow to identify the non-critical portions of a design.

Our power management technique relies on the observation that the slack possessed by individual nodes can be used as a guide to create partitions within which all nodes would maintain a certain level of timing freedom, which in turn can be exploited through scaling the voltage supply fed into that partition. It is well known that the dynamic power reduces by the square of the supply voltage ($P \sim C_{load} V_{dd}^2 f_{switch}$), and the delay increases linearly ($D \sim C_{load} / V_{dd}$) as the supply voltage is decreased. Hence, it is possible to perform a tradeoff between power consumption and performance by changing the supply voltage.

Our aim is to identify partitions in a design, such that the total power consumption is minimized while resource constraints associated with the partitioning problem are satisfied. For a given partition, the length of the longest path and the amount of time slack available along that path will be used to compute the voltage scaling within a partition. The voltage-scaling factor (Sc_Fac) is then defined as

$$Sc\_Fac = \frac{longest\_path\_length\_in\_partition}{longest\_path\_length\_in\_partition + slack}$$

Our partitioning scheme can find applications at various levels. Next we differentiate between two cases and are shown in 0.

3.1.1 Chip-level Voltage Scaling for Multi-FPGA Systems

Many FPGA-based hardware acceleration systems employ multiple FPGAs. Partitioning is frequently used to map a large design onto several FPGA devices. Such systems are generally wired; hence, they do not operate under a tight power budget. However, heat dissipation is becoming a growing concern. Overhead due to cooling systems can be reduced through effective power management. More importantly, reliability issues arising due to excessive heat dissipation require closer attention as the FPGA manufacturing technology reached submicron levels. For multi-FPGA systems we apply our partitioning technique to create partitions such that some of those partitions can be assigned to a device operating at a lower supply voltage level.

3.1.2 Localized Voltage Scaling for Single FPGA Systems

A design flow targeting a single FPGA device can also benefit from our power-driven partitioning technique. In this case, by embedding voltage islands on a chip we can enable different parts of a design to operate at different voltage levels. Generating these voltage islands on a single FPGA will incur certain hardware costs. Considering the overhead of creating voltage islands –additional circuitry for voltage scaling, level conversion, etc. the number and layout of different voltage islands can be constrained. We believe that the benefits of voltage scaling will justify the additional hardware cost in the next generation FPGA architectures. We will elaborate more on this issue as we present our results in Section 4.

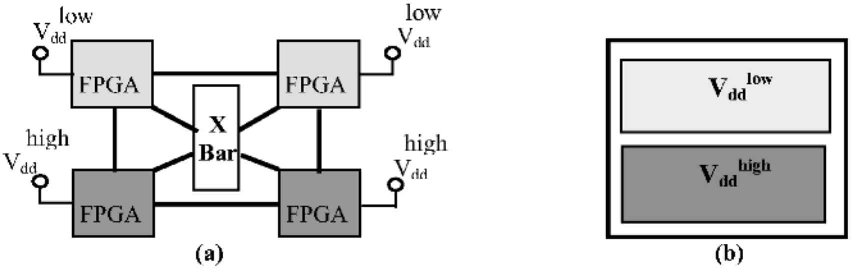


Fig. 1. Application of voltage scaling in two different scenarios. (a) Multi-FPGA system. (b) Single FPGA containing voltage islands.

3.2 Power-Driven Partitioning Algorithm

Our algorithm takes in a LUT – level netlist. The input buffer connected to the input port (*IBUF*) is considered a primary input and output buffer connected to the output port (*OBUF*) is considered a primary output. We assume that each LUT has a delay of 1 unit when operating at full supply voltage level.

Our algorithm tries to identify clusters of nodes along a path, which can share the time slack available along that path. The reasons why we consider only nodes along a path are twofold. First, since nodes along a path have data dependencies, they are likely to be placed in the same partition (or be placed close physically in the case of single FPGA systems) even without voltage scaling considerations. Second, the slack values on the nodes along a path are usually close to each other. Clearly, a good power-driven partitioning algorithm clusters nodes with similar slack values together.

Our algorithm first finds the slack values on the nodes, forms an initial set of clusters assuming the availability of arbitrary scaling factors. Then, in a refinement phase the voltage scaling factors of the clusters are adjusted according to the available supply voltage levels in the target hardware.

The slack at each node is calculated by computing the difference of required and arrival times of the signals by sorting the nodes in topological order. The algorithm takes as input the feasible scale factor Sc_Fac – the minimum amount by which voltage can be scaled. Then, it selects the nodes in topological order thereby selecting nodes from the input level and going towards the output. If the node has zero slack it is added to the non-scaled partition. After choosing a non-zero slack node *v,* which is not already added to any partition, it is checked whether

$$Sc\_Fac > \frac{longest\_path\_in\_partition}{longest\_path\_in\_partition + slack}$$

If this condition is satisfied, then a new cluster is created and the algorithm tries to grow the cluster. (As we add the first node to a partition, the delay of the longest path in the partition is equal to the delay of the node itself.) Iteratively, minimum slack fanout nodes are selected (not already in another cluster) and added to the cluster if it is feasible to add a new node and the slack of the path is updated as the minimum of the slacks of the nodes in the path. Let us call this entity $\text{slack}^{\text{Path}}$. Similarly, the length of the longest path in the partition is updated.

Once we stop adding any more nodes to a cluster c_i, we will have the following information about this cluster: $\boldsymbol{M_i}$: number of nodes along the longest path in c_i, $\boldsymbol{L_i}$: length of the longest path in c_i, $\boldsymbol{slack^{Path}}$: available slack along the longest path in c_i. (Without violating any overall timing constraints of the circuit, we can slow down the longest path in this cluster by $\text{slack}^{\text{Path}}$ time units.), ***node_delay***: amount of time by which each individual node in c_i can be slowed down. We can formally express *node_delay* as follows:

$$node\_delay = \frac{slack^{Path}}{M_i}$$

Since the same voltage-scaling factor will be applied to all nodes within a partition, the slowdown of each node will be same. The voltage supply for this cluster can be scaled down by the factor of

$$scale^{cluster_i} = \frac{L_i}{L_i + slack^{Path}}$$

Iteratively, clusters are created until all nodes are assigned to a cluster.

3.2.1 Post-processing of Clusters

After creating an initial set of clusters our next goal is to assign these clusters into voltage scaled FPGAs or individual voltage islands on a single chip. The circuitry employed for scaling will have a pre-defined sensitivity. In other words, the incremental steps by which we can adjust the voltage level are quantized, e.g. voltage scaling can be within a range of 0.64 Volts in 8 steps of 0.08 Volts. First, we perform a pass over all clusters and round up their voltage levels to the nearest feasible level.

If the number of clusters in the initial partition is less than or equal to the number of FPGA devices in the system then the assignment is straightforward. For the single FPGA case, we will identify two distinct voltage levels: V_{dd}^{high} and V_{dd}^{low}. Hence, each cluster will be merged to either one of these two levels. For the multi-FPGA case, we can allow more voltage levels since one voltage regulator will serve one individual chip. We developed two heuristics to achieve the finalized partitioning of the netlist into *N* voltage scaled partitions (*N*=2 for single FPGA scenario). We refer to these schemes as Delay Based Merging and Connectivity Based Merging. Delay Based Merging tries to satisfy the resource constraint while searching for the best possible merging in terms of power gain. Essentially, this method tries to group nodes with the most similar slack values together in the same merged cluster. Connectivity Based Merging puts higher emphasis on reducing the cut cost of the final partitioning result. It achieves this by merging clusters with highest number of mutual connections together. We omitted the details of these post-processing heuristics and present only the results of Connectivity Based Merging due to space considerations. After completing the partitioning phase, the total power consumption now becomes P_{scale}. If the total power consumption without any voltage scaling was P, the ratio is given by

$$\frac{P_{scale}}{P} = \frac{(V_{dd}^{scale})^2 \times k + (n-k)}{n}$$

where, *n* is the total number of nodes in the circuit and *k* is the number of nodes in the voltage scaled partition. For an *r-way* partition into *r* FPGA devices, we take the same approach. Assume that the partition sizes are $k_1, k_2, k_3, \ldots k_r$. The ratio of the scaled power consumption to the non-scaled power consumption is given by

$$\frac{P_{scale}}{P} = \frac{(V_{dd}^{scale_1})^2 \times k_1 + (V_{dd}^{scale_2})^2 \times k_2 + \ldots + (V_{dd}^{scale_r})^2 \times k_r}{n}$$

4 Experiments

We present our experimental results in this section. First, we summarize our experimental setup and the parameters we used. Then, we report achieved reduction in total power consumption on a collection of benchmarks.

4.1 Experimental Setup and Parameters

The experiments are formed on a set of MCNC combinational benchmarks. Further we also tested with synthetic benchmarks of 1000, 5000, 10000 and 15000 nodes. Table 1 summarizes relevant characteristics: number of nodes, edges and *critical nodes* - the number of nodes with zero slack in the DAG representation of each MCNC benchmark. In Table 2 we show the average values of the same for synthetic benchmarks. It is to be noted that though the results are presented for combinational circuits, the algorithm is equally applicable to sequential circuits where its combinational block can be partitioned into voltage clusters.

The LUT level netlist for MCNC benchmarks were in .net format. The power-driven partitioner reads in an .edif or .net file and produces a partition for *N* FPGAs in a multi-FPGA system or *N=2* voltage islands on a single FPGA. The initial delay of each LUT is assumed to be 1unit. We further assumed that voltage scaling is done in increments of 0.06 Volts and scale factor is 0.62. Finally, we assume that for the multi-FPGA scenario, all FPGA devices in the system are identical, i.e., all have the same capacity. Our techniques are independent of the actual values of these parameters; hence, they can operate under different assumptions.

4.2 Results: Power Driven Partitioning

After creating clusters we perform an initial refining, where we round off voltage scaling values according to the smallest scaling step. Next we perform our post processing heuristics of Delay Based Merging and Connectivity Based Merging. In both the merging we generate N-voltage clusters each of roughly the same size. We observe that while we obtain better power improvement using Delay Based Merging the cut cost is consistently inferior to the Connectivity Based Merging. Symmetrically, Connectivity Based Merging always yields better-cut cost at the expense of reduced power improvement. Hence, the possible trade-off between the cut cost objective and power improvement is evident. Table 1 and Table 2 shows the results of only connectivity based merging. We report the cut cost, i.e., the number of inter partition connections as well as the percentage power improvement obtained. For MCNC benchmarks, the average power improves 13.4% for 2-way to 14.7% for 8 way whereas the average cut cost increases from 2389.5 for 2-way to 2530.9 for 8-way. For Synthetic benchmarks the average power improvement is 27.45% for 2-way and 48.2% for 8-way.

Table 1. Partitioning Results for MCNC benchmarks

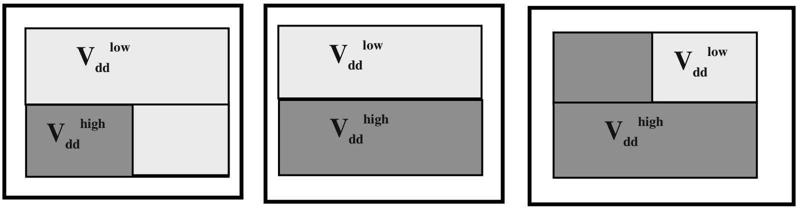
				2-way		4 way		8-way	
MCNC	Nodes	Edges	Crit Nodes	Cost	%Imp	Cost	%Imp	Cost	%Imp
Alu4	1522	5401	1180	1467	13	1483	13	1556	17
apex2	1878	6690	1453	2158	13	2190	14	2335	14
apex4	1262	4461	1169	440	6	440	6	440	6
des	1591	5863	1326	1851	18	1888	18	1890	19
ex1010	4598	16069	3557	5660	13	5748	13	5868	13
ex5p	1064	3940	941	663	9	663	9	663	9
misex3	1397	4955	992	1405	19	1501	22	1576	22
pdc	4575	17154	3179	4901	13	5141	14	5315	14
seq	1750	6159	936	1685	21	1804	21	1832	21
spla	3960	13763	2919	3665	9	3803	9	3834	12
Maximum				5660	21	5748	22	5868	22
Average				2389.5	13.4	2466.1	13.9	2530.9	14.7

Table 2. Partitioning Results for synthetic benchmarks (average)

Synthetic			2-way		4 way		8-way	
Nodes	Edges	Crit Nodes	Cost	%Imp	Cost	%Imp	Cost	%Imp
1000	1751	129	360	27.2	794	39.4	941	47.6
5000	8323	209	588	29.6	1392	41.8	1662	50.2
10000	16270	276	775	27.8	1962	41.6	2357	48.4
15000	24220	359	1023	25.2	2578	37.6	3087	46.6
Maximum			1023	29.6	2578.2	41.8	3087	50.2
Average			686.45	27.45	1681.45	40.1	2011.8	48.2

4.3 Results: Using Power Driven Partitioning in Physical Design

We now present the results of the constrained placement of the clusters identified by our algorithm onto for voltage islands supplied by different V_{dd}s on a single FPGA. We propose to use four quadrants as four voltage islands, each of which can be potentially supplied by either V_{dd}^{high} (say 1.3V) or V_{dd}^{low} (say 0.8V) rather than having each individual logic block being V_{dd} programmable. The location voltage islands is an architectural parameter. Figure 2 shows the 3 possible configurations. We assume that it is possible to have level converters along the borders of the quadrants. FPGA having dimensions D_X, D_Y has total ($D_X \times D_Y$) logic blocks. Our placement uses f – the fraction of critical nodes to total logic blocks. Configuration 2(a) is used if is f ¼, 2(b) is used if ¼ f ½ and 2(c) is used if ½ f ¾. Obviously for ¾ f 1, all the quadrants must be supplied with V_{dd}^{high} and there is no power improvement possible by V_{dd} scaling.



(a) **(b)** **(c)**

Fig. 2. Proposed Voltage Island configurations in a FPGA

Our placement works as follows: The cluster of *critical nodes* identified by power driven partitioning algorithm presented in Section 3.2 must be placed in a V_{dd}^{high} region – (determined by f) otherwise it would mean slowing down the critical path. The power improvement is higher if all *non-critical nodes* get placed in the V_{dd}^{low} region although they have a freedom to move to V_{dd}^{high} quadrant if it improves the congestion cost.

The placement is based on the simulated annealing implemented in VPR [16]. The linear congestion cost function used in VPR is given by

$C_{linear\_conj} = \sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)^{\beta}} + \frac{bb_y(i)}{C_{av,y}(i)^{\beta}} \right]$ where for each net $bb_i(x)$ and $bb_i(y)$ denote the horizontal and vertical spans of its bounding box, q(i) is the compensating factor and $C_{av,x}(i)$ and $C_{av,y}(i)$ are average channel capacitances in the x and y directions respectively. Our cost function *C* is similar to that presented in [16] and is given as:

$$\Delta C = \Delta C_{linear\_conj} + \alpha \Delta matched(j) + \gamma(1 - matched(j))$$

where matched(j) returns 1 if the jth logic block is placed in its matching voltage quadrant and 0 if not. Δmatched is the difference between matched(j) in the previous placement and matched(j) in the present placement and penalizes a move that brings a block from matched quadrant to unmatched quadrant; (1-matched(j)) penalizes a move that moves a block in unmatched quadrant to another location in the unmatched quadrant; α, γ are appropriate constants and γ> α.

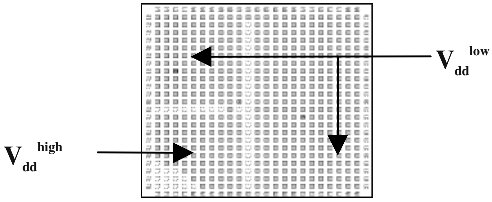


Fig. 3. Constrained Placement of LUTs clusters of apex2

Table 3. Placement and Routing results for MCNC benchmarks

							Unconstrained		Constrained –Power Driven			
Circuits	Logic Blocks	Crit Block	Dx=Dy	%*f*	PI	PO	WL	CW	WL	CW	Blocks in V_{low}	%Imp
alu4	389	51	20	13	14	8	20262	33	22151	37	289	46
apex2	485	66	23	12	38	3	31738	40	37348	47	385	49
apex4	334	77	19	21	9	19	20317	40	23383	48	247	46
des	415	47	32	5	256	245	25199	25	44587	41	357	53
ex1010	1201	250	35	20	10	10	72916	43	104744	69	908	47
ex5p	280	22	17	8	8	63	17572	41	19625	47	213	47
misex3	362	57	20	14	14	14	21650	36	24606	39	281	48
pdc	1190	141	35	12	16	40	98312	61	116923	71	911	48
seq	448	94	22	19	41	35	27904	39	35767	48	342	47
spla	955	133	31	14	16	46	71330	57	86536	66	717	47
Average							40720	42	51567	51	465	48
Avg. Penalty for Const-Power Driven					Wire length		26.64%	Channel width			23.61%	

For FPGAs it is common practice to pack the LUTs into logic blocks. We used T-Vpack [16] to pack LUTs with clusters of size 4 and 10 inputs per cluster option. After packing, the power driven partitioner partitions the logic blocks for V_{dd}^{high} and V_{dd}^{low} regions. For apex2 benchmark the characteristics are: the smallest FPGA size is

23×23 logic array, out of 485 clusters there are 66 critical logic blocks and f is 12%. The placement is of type shown in Figure 2(a) and the placed circuit is shown in Figure 3.

Ten MCNC benchmarks were clustered and Table 3 shows their total number of logic blocks, primary inputs (PI) and outputs (PO), % f and the FPGA size $D_X \times D_Y$. Wire length (WL) and channel width (CW) are shown for unconstrained placement vs. power driven partitioning and region-constrained placement. We also report the number of non-critical blocks in the V_{dd}^{low} region and the power improvement over unconstrained partitioning. The results show that it is possible to get an average power improvement of 48% with 23% penalty in channel width and 26% penalty in wire length.

5 Conclusions

In this paper, we presented a power optimization technique for FPGA-based systems. Our proposed approach aims to create opportunities for voltage scaling by grouping nodes in a LUT-level netlist into clusters. The partitioning approach targets to exploit the maximum flexibility in timing constraints and convert it into voltage scaling. We developed a partitioning algorithm to perform this task. Within the general framework of our algorithm resource constraints can be resolved. Our experimental results reveal that it is indeed effective. Based on the high-level power improvement estimation, we did a constrained placement of the clusters onto voltage islands in a single FPGA and showed power improvement vs. penalty in wire length and channel width increase. Hence possible tradeoffs between cost function and power improvement is evident. An immediate extension to our work is to evaluate and actually compare delay after physical design due to constrained placement and area-power tradeoffs due to the presence of level converters.

References

1. Boemo, E.I., et al., eds. Some Notes on Power Management on FPGA-based Systems. Lecture Notes in Computer Science. Vol. 975. 1995, Springer-Verlag: Berlin. 149–157.
2. Chen, C., T. Hwang, and C.L. Liu. Low Power FPGA design – A Re-engineering Approach. in Design Automation Conference. 1997.
3. Sutter, G., et al. FSM Decomposition for Low Power in FPGA. in Design Automation Conference. 1998.
4. Chen, D., J. Cong, and Y. Fan. Low Power High-Level Synthesis for FPGA Architectures. in International Symposium on Low Power Electronic Design. 2003.
5. Lamoureux, J. and S.J.E. Wilton. On the Interaction Between Power-Aware FPGA CAD Algorithms. in International Conference on Computer Aided Design. 2003.
6. Brasen, D.R. and G. Saucier, Using Cone Structures for Circuit Partitioning into FPGA Packages. IEEE Transactions on CAD of Integrated Circuits and Systems, 1998. 17(7): p. 592–600.

7. Chan, P.K., M.D.F. Schlag, and J.Y. Zien. Spectral-Based Multi-Way FPGA Partitioning. in International Symposium on Field Programmable Gate Arrays. 1995.
8. Chang, D. and M. Marek-Sadowska, Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs. IEEE Transactions on Computers, 1999. 48(6): p. 565–578.
9. Liu, H. and D.F. Wong. Circuit Partitioning for Dynamically Reconfigurable FPGAs. in International Symposium on Field Programmable Gate Arrays. 1999.
10. Govil, K., E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed Setting of a Low Power CPU. in MOBICON. 1995.
11. Iyer, A. and D. Marculescu. Power Efficiency of Voltage Scaling in Multiple Clock, Multiple Voltage Cores. in International Conference on Computer Aided Design. 2002.
12. Yeh, C., et al. Gate-Level design Exploiting Dual Supply Voltages for Power-Driven Applications. in Design Automation Conference. 1999.
13. Simunic, T., et al. Dynamic Voltage Scaling and Power Management for Portable Systems. in Design Automation Conference. 2001.
14. Chen, D., et al. Low-Power Technology Mapping for FPGA Architectures with Dual Supply Voltage. in International Symposium on Field-Programmable Gate Arrays. 2004.
15. Li, F., et al. Low-Power FPGA Using Pre-Defined Dual-Vdd/Dual-Vt Fabrics. in International Symposium on Field-Programmable Gate Arrays. 2004.
16. Betz, V., J. Rose, and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, Feb 1999.

Power Consumption Reduction Through Dynamic Reconfiguration*

Michael G. Lorenz, Luis Mengibar, Mario G. Valderas, and Luis Entrena

Electronic Technology Department, Microelectronics Group
Carlos III University of Madrid
Avenida. de la Universidad, 30. E-28911-Leganés (Madrid), Spain
{lorenz, mengibar, mgvalder, entrena}@ing.uc3m.es

Abstract. Dynamic reconfiguration optimizes the use of hardware resources, and therefore may produce important reductions in power consumption. However, in a reconfigurable system the power consumption produced by the reconfiguration process itself must be taken into account. In this work the reconfiguration power consumption is characterized for a SRAM FPGA. In particular, we show that reconfiguration must be made at the highest frequency available in order to reduce power consumption. The results obtained allow to quantify the tradeoff between the energy saved by the use of dynamic reconfiguration and the energy wasted by the reconfiguration process. In this way, the power consumption reduction that can be obtained with the use of dynamic reconfiguration can be estimated.

1 Introduction

Power consumption is becoming a concern in programmable logic design as the size and performance of modern FPGAs increase. The use of the dynamic reconfiguration is one of the ways that can be used to reduce the power consumption [1],[2]. Since in a reconfigurable system the hardware resources are multiplexed in the time, reducing the size of the device, it can be expected that power consumption is also reduced.

Traditionally, power consumption is divided in static and dynamic power. Dynamic power consumption depends on the switched capacitance and frequency. For a particular processing task, it can be reasonably expected that dynamic power will be similar in a reconfigurable system and in a non reconfigurable system built with the same technology. Nevertheless, this can only be obtained if the activity of idle modules in the non reconfigurable system is fully suppressed, which in practice cannot be fully achieved.

Anyway, the advantage in static power consumption of a reconfigurable system is undeniable. It must be noted that current FPGAs have very important static power dissipation [3]. If K is the number of reconfigurations, then it is possible to use a K times smaller device, and the static consumption will be reduced in the same

* This work has been partially funded by Spanish Government, through the research project TIC-2003-01793 SIDECAR, (Identification and Communications Systems through Reconfigurable Hardware)

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 751–760, 2004.

proportion. Actually, the reduction is going to be in an intermediate value between K and 1, depending on each application and the efficiency of the used reconfiguration. On the other hand, in a reconfigurable system a new power component appears, that it is the consumption due to the reconfiguration.

Currently, there are tools that allow to estimate with acceptable accuracy both static and dynamic power consumption [4]. However, in a reconfigurable system it is necessary to take into account also the energy wasted by the configuration process itself. When configuration is performed only once at power-up, the configuration power consumption can be usually neglected. However, in a reconfigurable system it may be a primary source of power dissipation when reconfiguration is performed often. Unfortunately, tools or data for the estimation of the reconfiguration power consumption are not available.

In order to determine the power savings that can be obtained by using dynamic reconfiguration, the reconfiguration power must be first estimated. To the best of our knowledge, this new component of the power consumption has not been evaluated so far either by manufacturers or researchers, apart from some preliminary works [5]. The main objective of our work is the evaluation of the reconfiguration power consumption and to study the viability of the reduction of the consumption that can be obtained by means of dynamic reconfiguration, compensating the increment that takes place in the power consumption due to the reconfiguration with the reduction that is obtained in the other power components. To this purpose, the reconfiguration power has been measured directly for a set of benchmarks that cover the full range of possible FPGA occupation. Moreover, the variation of instantaneous power consumption during reconfiguration is analyzed in detail. Finally, a comparison of the power consumption in reconfigurable and non reconfigurable systems is performed to determine when a dynamically reconfigurable system is power effective.

The remaining of the paper is as follows. Section 2 introduces reconfiguration power consumption. Section 3 presents the experimental results of reconfiguration power measurement and analyze in detail the evolution of reconfiguration power and energy with time. Section 4 presents a comparison of the power consumption in reconfigurable and non reconfigurable systems. Finally, section 5 presents the conclusions of this work.

2 Reconfiguration Power Consumption

Fig.1 shows the supply current to the FPGA during the dynamic reconfiguration. Note that the current scale (y-axis) is set in a negative range.

The actual mechanism of FPGA reconfiguration is as follows. The process begins with the configuration of the look up tables, multiplexers and the flip-flops inside each individual cell. In a second stage all the interconnections paths are configured. From Fig.1 we can see that most of the energy is wasted in this second stage.

3 Reconfiguration Power Consumption Measurement

In this section we present the results of a set of experiments devoted to measuring the reconfiguration power consumption. For these experiments we have chosen a partially reconfigurable device AT40K20 FPGA from ATMEL. This device can be dynamically reconfigured for any number of cells. A set of designs has been chosen to include all the range of possible sizes. This way we tried to obtain estimation for the power consumption during reconfiguration for designs that go from a low percentage of the FPGA, to a percentage that practically uses the complete FPGA.

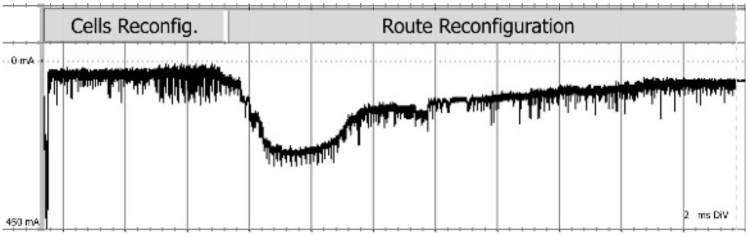


Fig. 1. Power consumption during dynamic partial reconfiguration (Atmel AT40K20 FPGA)

In order to estimate the reconfiguration power consumption, measurements of instantaneous power consumption have been made. Each of the power consumption measurements contains a series of data with up to a maximum of 32K measurements. Average power can be computed using the following equation:

$$P_m = \frac{1}{t}\int_0^t I \cdot V dt \cong \frac{1}{n}\sum_{i=1}^{n} I_i \cdot V_i \qquad (1)$$

This equation (1) if just the quantization of the power consumption in a given time interval. The number of samples and the sampling frequency is tuned in each series in order to get the greatest possible accuracy.

Power consumption measurements have been performed on a complete set of parameterized benchmark examples that cover a full range of FPGA occupation, from 2,5% up to 93%. The benchmark examples are library multipliers ranging from 4 bits up to 40 bits wide, with the inputs and outputs registered.

The example designs are placed and routed automatically by the software tool: "*Integrated Development System*" IDS from Atmel. All the examples have been made in four different versions, for the four reconfiguration frequencies allowed: 1 MHz, 4 MHz 8 MHz and 16 MHz. The energy consumption measurements are shown in Table 1.

As expected, reconfiguration energy increases with the size of the design at any reconfiguration frequency. Actually, reconfiguration energy increases linearly with the length of the bitstream. However, the energy consumption for the design with the largest number of cells, occupation 93,85%, has a value of 35 mJ at 1 MHz and 1,5 mJ at 16 MHz. This means that the dissipation of the circuit at 1 MHz is 5,2 times higher than at 4 MHz, 14,6 times higher than at 8 MHz and 23,9 times higher than at

16 MHz, (1,47 mJ). In order to explain this result, we have analyzed the evolution in time of the reconfiguration power.

Fig. 2 shows four curves that represent the measure of power consumed by the FPGA during the reconfiguration process. Each curve corresponds to a different reconfiguration clock frequency. In order to be able to compare them, the values of the X-axis have been scaled, so that the different events match in the graph as if they were made at the same clock frequency. Thus, for instance, the reconfiguration time at 16 MHz would be 16 times shorter than the reconfiguration time at 1 MHz. For this reason, we multiplied the curve at 16 MHz by a time scale of 16, the curve at 8 MHz by a time scale of 8 and so on. This way, the X-axis represents time as a fraction of the total reconfiguration time.

Table 1. Reconfiguration Energy Consumption

Bits of the Multiplier	Cells	Occupation Level %	Energy mJ At 1MHz	Energy mJ At 4MHz	Energy mJ At 8MHz	Energy mJ At 16MHz
4	25	2,44	0,2529	0,1043	0,0792	0,0714
6	48	4,69	0,4027	0,1653	0,1308	0,1120
8	81	7,91	0,8384	0,2676	0,2100	0,1778
10	122	11,91	1,6205	0,3612	0,2670	0,2314
12	173	16,89	2,3679	0,4634	0,3324	0,2816
14	231	22,56	3,6585	0,6712	0,4593	0,3709
16	292	28,52	5,8468	0,8855	0.5670	0,4354
18	364	33,55	9,3900	1,5035	0,7858	0,6089
20	445	43,46	15,020	1,9368	1,0113	0,7511
22	533	52,05	18,296	2,4365	1,1430	0,8210
24	632	61,72	26,942	3,4151	1,4805	1,0174
26	735	71,78	25,679	3,7226	1,5960	1,1003
28	846	82,52	30,725	4,6865	2,0167	1,2937
30	961	93,85	35,109	6,2058	2,3909	1,4705

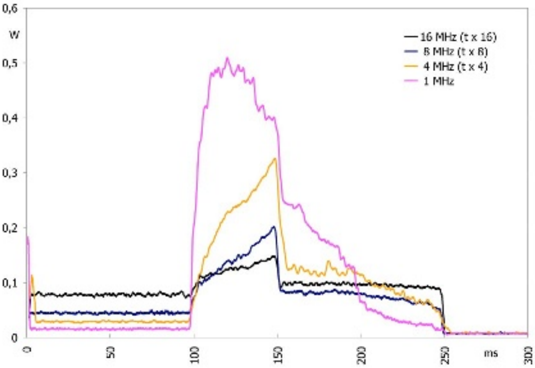


Fig. 2. Configuration Power Consumption

As it has been described previously, reconfiguration is divided in two phases. The first phase is the reconfiguration of the cells and can be observed in the left side of Fig.2. The second phase is the reconfiguration of the interconnection and can be observed in the right side of Fig.2. In order to analyze the contribution of each of these phases to the total power consumption, they are presented separately in Fig.3 and Fig.4.

Looking at Fig.3, it can be clearly seen that the power consumption during this phase of the reconfiguration at 16MHz is approximately double than at 8MHz, and at 8MHz has double power consumption than at 4MHz as well. If the instantaneous consumption is reshaped to 1 MHz its power consumption is approximately half than at 4MHz. In a first approach we can see that the instantaneous consumption is directly proportional to the frequency.

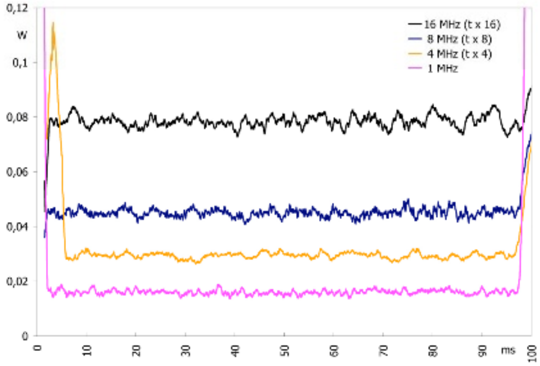


Fig. 3. Cells Configuration Power Consumption

The well-known equation of consumption in CMOS circuits is:

$$P = \sum_{j} I_{leak} V_{DD} + \sum_{i} K_i V_{DD} I_{SC} + \sum_{i} C_i V_{DD}^2 F_i \tag{2}$$

It is possible to see that during the reconfiguration period, the FPGA follows this equation very well. At high frequencies, the last term corresponding to the dynamic consumption prevails, so that power is proportional to the frequency. At low frequencies, i.e. at 1MHz, static consumption power, which corresponds with first term of the equation 2, becomes more important.

Fig. 4 shows the power consumption for the interconnection reconfiguration phase. Most of the total reconfiguration power consumption takes place in this phase, which shows a large peak of power. In particular, it can be observed that reconfiguration at 1 MHz takes more time and has a much larger peak than at the other frequencies, thus producing larger power consumption. We will try to explain the reason for this phenomenon.

First of all, we shall remark that we are measuring power for a partial reconfiguration process. The reconfiguration process of the interconnections between the used cells is performed without resetting the existing connections that were configured previously. Therefore, weak short circuits between different elements from the FPGA may take place during a short period of time, while different cells elements are being connected and others become disconnected. This type of consumption is like a short circuit power consumption that will increase with the time this current flows through the short-circuit. For this reason, when the FPGA is reconfigured at 1

MHz this short-circuit current will flow during a period of time up to 16 times greater than if the reconfiguration was made at 16MHz. This would justify a power consumption of up to 16 times more at 1MHz than at 16MHz, which is the same order of magnitude than the measured power increase.

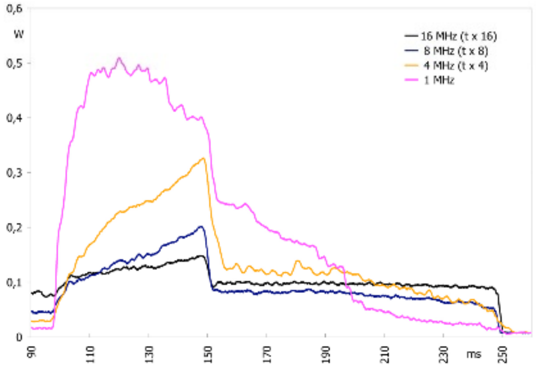


Fig. 4. Configuration Power Consumption of Interconnections

After the reconfiguration process has finished, the power consumption of the FPGA is identical for all cases and correspond to the static power consumption. For the FPGA used in the experiments, the static power consumption is approximately 8 mW.

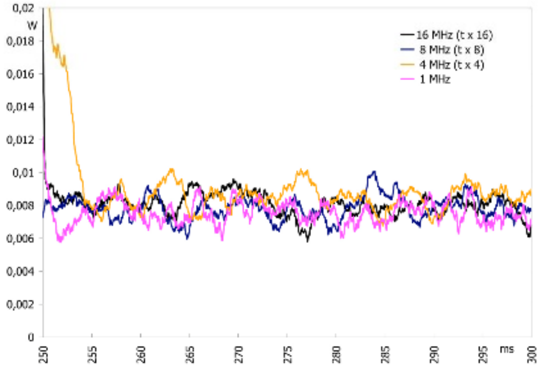


Fig. 5. Configuration Power Consumption of Interconnections

Perhaps it is more significant to study the reconfiguration consumption in energy terms. This is shown in Fig.6. If the reconfiguration takes place at different clock frequencies, it is not possible to show in the same x-axis an exact value of the energy for all the curves, so arbitrary units are used. In this case, the energy consumption for the reconfiguration at 1 MHz is observed to be much higher than at 16 MHz.

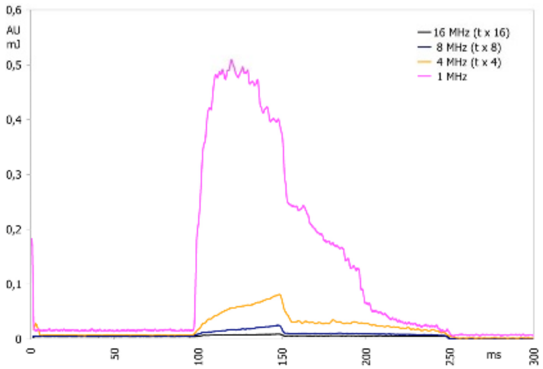


Fig. 6. Cells configuration Energy Consumption

The energy consumption during the initial phase of the FPGA reconfiguration can be observed in detail in Fig 7. In this case, the energy consumption at 16MHz is the smaller of all of them. For the rest of the frequencies, energy consumption increases in inverse proportion to the frequency. In summary, dynamic energy consumption is identical for all cases, and the differences are due to the static energy consumption, which is proportional to the duration of the reconfiguration process.

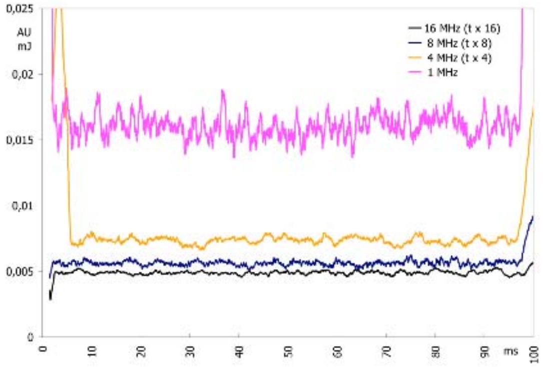


Fig. 7. Cells configuration Energy Consumption

The second phase of the energy consumption curve during the reconfiguration can be seen in detail in Fig.8. In this figure it can be observed that the reconfiguration energy consumption is substantially greater if the reconfiguration takes place at a frequency of 1MHz than to anyone of the other frequencies. The energy consumption in this phase is at least ten times smaller if the reconfiguration takes place at 4MHz than at 1MHz. In the other cases, at higher frequencies, the energy consumption is even smaller.

In conclusion, experimental results demonstrates that the energy wasted during the reconfiguration process is mainly dominated by short-circuit and static power consumption. Since these power components increase with time, the reconfiguration must be made at the higher frequency available in order to reduce the power consumption.

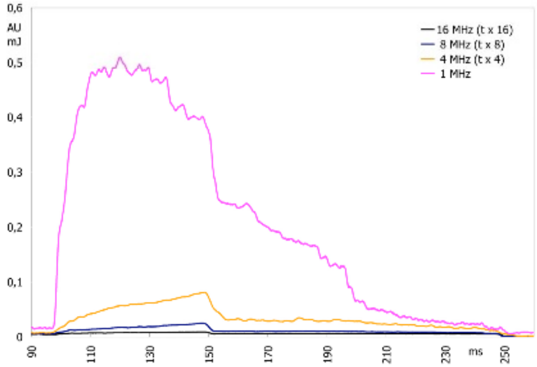


Fig. 8. Interconnections Configuration Energy Consumption

4 Reduction of Consumption with Reconfiguration

After characterizing the reconfiguration power consumption, the power savings produced by the use of dynamic reconfiguration can be estimated. To this purpose we will compare the energy required by a reconfigurable system with respect to a conventional implementation.

The energy required by a reconfigurable system to perform a particular processing is the sum of static, dynamic and reconfiguration power. If the processing requires reconfiguring the system K times, then the total energy can be expressed as follows:

$$E_{Total\_Reconfiguración} = E_{Static} + E_{Dynamic} + K \cdot E_{Reconfiguration} \tag{3}$$

where E $_{\text{Reconfiguration}}$ is the average energy wasted for a single reconfiguration.

In a non-reconfigurable system that performs the same processing, all processing units will be present at any time. Therefore, the system will be N times larger and the energy required will be:

$$E_{Total} = N \cdot E_{Static} + E_{Dynamic} \tag{4}$$

The static power consumption of the FPGA roughly depends on the size of the device. Hence, it can be estimated that static power consumption will be N times greater than in the case of a reconfigurable system. For the dynamic consumption, we will consider the most unfavourable case for the reconfigurable system, this is, identical power consumption in both systems. In practice, this requires eliminating all switching for idle modules and can only be achieved by applying some low power consumption techniques such as clock gating [6],[7]. In any case, these techniques are not fully effective in practice. Therefore, considering dynamic power consumption identical in both cases is a conservative assumption.

In order that the reconfigurable system is more power efficient, the following equation must be satisfied

$$E_{Total\_Reconfiguration} < E_{Total} \tag{5}$$

Operating in the previous equation and replacing values, we get

$$E_{Static} + E_{Dynamic} + k \cdot E_{Reconfiguration} - N \cdot E_{Static} - E_{Dynamic} < 0 \tag{6}$$

As the dynamic energy can be considered identical for both systems, this term can be eliminated to get:

$$\frac{K}{N-1} E_{Reconfiguration} < E_{Static} \tag{7}$$

For a typical occupation of the FPGA of 70%, the ratio between K and N results approximately:

$$\frac{K}{N-1} \approx \frac{K}{N} \approx 1.4 \tag{8}$$

Equation 8 can be rewritten in terms of average power and elapsed times:

$$1.4 \cdot P_{Reconfiguration} \cdot \ t_{Reconfiguration} < P_{Static} \cdot \ t_{Process} \tag{9}$$

This equation shows that the energy savings that can be obtained with a reconfigurable system depend on the ratio between reconfiguration power and static power as well as the ration between reconfiguration time and processing time. With the data presented in the previous section, measured for the Atmel's FPGAs, we get:

$$\frac{t_{Process}}{t_{Reconfiguration}} > \frac{1.4 \cdot P_{Reconfiguration}}{P_{Static}} \approx 16 \tag{10}$$

For this example, reconfiguration time is 15 ms for reconfiguration at 16 MHz. Therefore, the reconfigurable system will be power profitable if the processing time for each process is at least 240 ms. In practice, processing time is usually much larger than reconfiguration time. For instance, in a typical dynamically reconfiguration system for image processing such as [8], processing time is about 23 times larger than reconfiguration time.

5 Conclusions

The use of dynamic reconfiguration may produce important reductions in power consumption. However, in a reconfigurable system the power consumption produced by the reconfiguration process itself must be taken into account. In this work, the reconfiguration power has been characterized in order to determine precisely the power reduction that can be obtained by using dynamic reconfiguration in a design.

This study reveals that power savings will be obtained if a sufficient ratio of processing time to reconfiguration time is achieved.

Although it can be expected that reconfiguration power would be basically proportional to the length of the reconfiguration bitstream, we have shown that there are other factors that can produce a dramatic increase in the reconfiguration power. In particular, we have shown that reconfiguration power is mainly dominated by transient effects that occur during reconfiguration of the interconnection. In order to minimize these effects, reconfiguration must be made using the highest frequency available. Future research should be oriented to develop techniques and technologies to reduce as much as possible the interconnection reconfiguration power consumption.

References

1. Hartenstein R. “Trends in Reconfigurable Logic and Reconfigurable Computing” 9th International Conference on Electronics, Circuits and Systems, 2002. 2 , 15-18 Sept. 2002 Pages:801 - 808 vol.2
2. Rabaey, J.M.;” Reconfigurable processing: the solution to low-power programmable DSP” IEEE International Conference on Acoustics, Speech, and Signal Processing, 1997. ICASSP-97, Vol. 1 , 21-24 April 1997 pp.275-278 vol.1
3. Spartan-3 2.5V FPGA Family: Complete Data Sheet. DS099 Advance Product Specification. Xilinx Inc. March. 2004
4. Xpower, “Xpower Tutorial FPGA design”, Xpower (V1.0) May 11, 2001. Xilinx.
5. Becker J., Huebner M., Ullmann M., “Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations” Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI’03), 8-11 Sept. 2003 pp. 283 – 288
6. L. Benini, G. de Micheli, E. Macii, “Designing Low-Power Circuits: Practical Recipes”, IEEE Circuits and Systems Magazine, Vol 1(1) First Quarter 2001, 6-25.
7. Luis Mengibar, “Contribution to low power design in FPGAs” PhD Thesis Carlos III University of Madrid, 2003.
8. Michael G. Lorenz, Luis Mengibar, Luis Entrena, Raúl Sánchez-Reillo. “Data Processing System With Self-reconfigurable Architecture, for Low Cost, Low Power Applications”. Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, pp. 220-229

The XPP Architecture and Its Co-simulation Within the Simulink Environment

M. Petrov[1], T. Murgan[1], F. May[2], M. Vorbach[2], P. Zipf[1], and M. Glesner[1]

[1] Institute of Microelectronic Systems at Darmstadt University of Technology
Karlstrasse 15, 64283 Darmstadt, Germany
Mihail.Petrov@mes.tu-darmstadt.de
[2] PACT XPP Technologies
Muthmannstrasse 1, 80939 Munich, Germany
Frank.May@pactxpp.com

Abstract. This paper offers an overview of the XPP, a coarse-grained reconfigurable architecture, and presents a solution for its integration into a Simulink design flow for rapid prototyping. This includes a system-level co-simulation followed by the automated code generation for an embedded target platform. In order to realize this functionality, a custom Simulink module has been developed. During the co-simulation phase, it acts as a wrapper for an external simulator, whereas when code is generated, it is responsible for generating the appropriate function calls for communicating with the XPP device. Of these two aspects, only the co-simulation is considered here.

1 Introduction

The e**X**treme **P**rocessing **P**latform (XPP) [1] is a runtime reconfigurable architecture optimized for parallel data stream processing in applications where flexibility and fast reconfigurability are demanded. The XPP architecture combines the performance of FPGAs with the flexibility of DSPs and is designed to support parallel processing through pipelining, instruction-level, data flow, and task parallelism.

As with every new architecture, software tools have to be developed, which include design entry, simulation, and technology mapping. The present work is part of an ongoing effort to extend the software support for the XPP [2] and provides a solution for integrating this architecture within Simulink. This helps to reduce the development time in a rapid prototyping flow for embedded systems, which includes the simulation and the automated code generation for a specific target platform. In this paper we only deal with the co-simulation aspect of the Simulink integration.

We start with an overview of the XPP architecture, with emphasis on its internal structure and its interfacing with the outside world, followed by a presentation of the XPP development environment. In the next section, we introduce the XPP simulator working in client/server mode and we analyze the possibilities for extending the functionality of Simulink. A solution is then presented,

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 761–770, 2004.

which consists in developing a special module that acts as a wrapper for the XPP simulator, followed by a short concluding paragraph.

2 XPP Architecture Background

The XPP architecture is structured as a hierarchical array of reconfigurable *Processing Array Elements* (PAEs) connected through a packet-oriented communication network [1][3]. The regular array organization, in conjunction with runtime reconfiguration capabilities, make this architecture particularly suited for highly parallel stream-based data processing tasks commonly encountered in multimedia, telecommunications, and graphics applications.

In this respect, XPP has similarities with other coarse-grained reconfigurable architectures, such as the *KressArray* [4], *RaPiD* [5], or *Raw Machines* [6], which are also designed and optimized for stream-based applications. However, the essential difference is the automatic packet-handling mechanism and the runtime reconfigurability of the XPP.

In a previous paper [7], it has been shown that the XPP architecture can successfully handle the base-band processing requirements of modern wireless communication standards. Moreover, thanks to its fast runtime reconfigurability (coarse-grained architecture), it is possible to switch between different standards on-the-fly.

An XPP array, shown in Fig. 1, consists of a relatively small number of different PAEs and has a very simple and homogenous structure, which facilitates algorithm mapping and routing. The building blocks of the array are enumerated below, together with a short functional description.

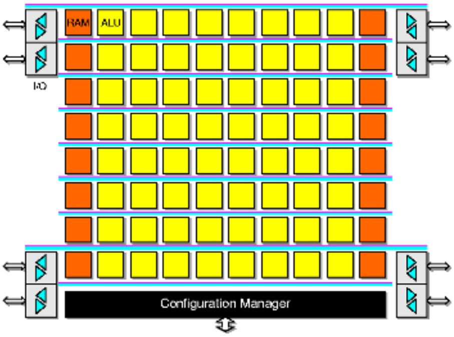


Fig. 1. Organization of an XPP array.

1. **ALU PAEs**, shown in Fig. 2, which integrate three types of objects with different functionalities: ALU, Back Register (BREG), and Forward Register (FREG). The ALU performs typical DSP functions, such as multiplication,

adition, comparison, sort, shift. The BREG provides vertical routing paths from bottom to top and an ALU that can be used for addition, barrel shifting, and normalization. The FREG provides routing paths from bottom to top and a specialized ALU that performs data stream control like multiplexing and swapping.

2. **RAM PAEs**, which provides intermediate RAM data storage. The RAM blocks are dual-ported, allowing simultaneous read and write, and have a typical size between 512 and 2K words. More RAM PAEs can be combined to a larger RAM with a contiguous address space. Moreover, they can also be configured as FIFOs.
3. **Configuration manager**, which handles all the configuration tasks of the array. Initially, it reads a configuration from an external RAM into the internal cache, then it configures the PAEs (opcodes, routing chanels configurations, constants). As soon as a PAE is configured, it can start its operation if data is available.
4. **I/O elements**, which act either as I/O ports or as interfaces to external RAMs. They will be detailed later in this section.
5. **Routing channels**, which run horizontally between PAE rows and route the data and events between PAEs. Individual PAEs are connected to a routing channel through configurable interconnects. The exact wiring of the routing network is depicted in Fig. 2, where four adjacent ALU PAEs are shown.

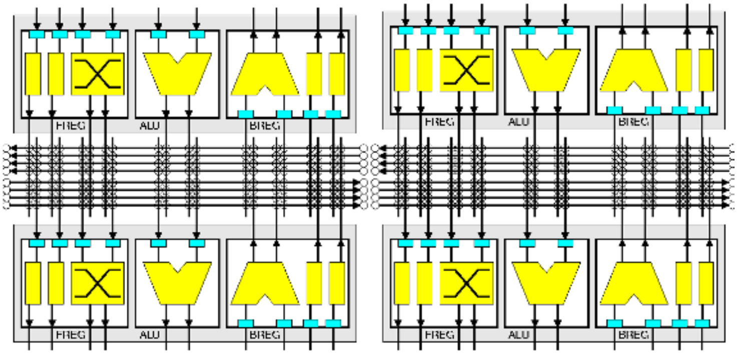
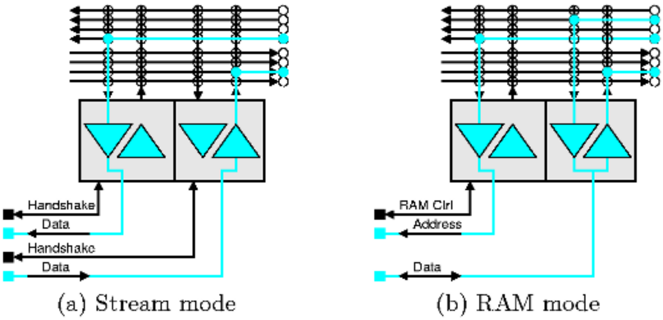


Fig. 2. ALU PAEs and routing network.

The XPP array is provided as a parameterizable soft core. The number of PAEs, as well as the width of the data path, can be adapted to different application needs. A demonstrator chip – XPP64-A – has been fabricated, featuring an XPP array of 64 ALU PAEs, 16 RAM PAEs, and 4 I/O elements and has a data path width of 24 bits.

All PAE objects communicate through a packet-oriented network, which routes two types of packets: data packets and event packets. PAE objects are self synchronizing. An operation is performed as soon as all necessary operands are available at the inputs, and the result is forwarded to the next PAE as soon as the previous result has been consumed. A token-oriented data flow is maintained, while the hardware protocol ensures no data is lost.



(a) Stream mode (b) RAM mode

Fig. 3. XPP I/O element operating modes

Data is fed to the device and the results read back using I/O elements, a special type of PAE, connected to horizontal routing channels. Each I/O element has two I/O ports and can be configured either as streaming port or as RAM interface. The two configuration modes are detailed in Fig. 3.

In streaming mode, each of the two ports of an I/O element can be configured as input or output, the packet handling being performed through an asynchronous ready/acknowledge protocol, which allows the interfacing with devices operating on a different clock domains, such as an A/D converter. However, the entire XPP chip runs on a single clock and is fully synchronous.

In RAM mode, an external synchronous SRAM can be connected to an I/O element. One of the two ports is configured as output and provides the RAM address and the control signals, while the other is bidirectional and used for data transfer. The size of the external RAM is only limited by the data width of the XPP implementation, e.g. 16 M words for 24 bits.

The operation of an XPP device consists in applying one or more data/event streams at the input and reading the results as soon as they are available. Due to the self synchronizing nature of the architecture, no explicit scheduling of operations is necessary, which simplifies application development significantly.

3 XPP Development Environment

In order to exploit the capabilities of the XPP architecture, the circuit configurations are described using *Native Mapping Language* (NML), a proprietary language developed by *PACT XPP Technologies*. NML is a structural language which includes reconfigurable primitives that give designers full access to all the hardware features. Configurations are defined by instantiating and placing (optional) PAE modules then specifying their connections.

The XPP design flow is shown in Fig. 4. The XMAP tool performs the placement of the PAEs and the configuration of the routing network as defined by the NML description. The resulting file (*.xbin) is used as a configuration for the XPP architecture. The rest of the design flow is for debug purposes and comprises a simulator (XSIM) and a visualizer (XVIS) that analyzes the state of the XPP array throughout the simulation process.

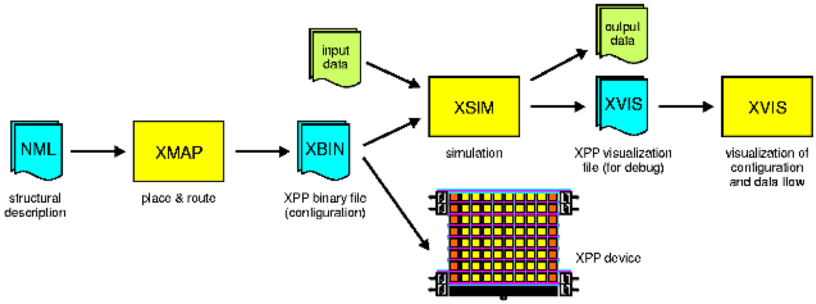

input data
output data
NML
structural description
XMAP
place & route
XBIN
XPP binary file (configuration)
XSIM
simulation
XVIS
XPP visualization file (for debug)
XVIS
visualization of configuration and data flow
XPP device

Fig. 4. XPP native development chain.

As a further aid in development, a vectorizing C compiler is available [2], which is able to translate a subset of the C language into NML modules. Particularly suitable are the computation intensive loops, which can be vectorized and executed in a pipelined fashion. However, for programs that contain control statements, the partitioning into control and computation intensive parts has to be made manually.

The control-oriented code sequences will be mapped on a microcontroller and will include API calls to functions that exchange data with the reconfigurable array, which will execute only the computation intensive tasks. The XPP array is connected to the controller's main bus and acts as a reconfigurable accelerator for data-flow algorithms.

Besides the C approach, another popular method for system specification and design is to use the Simulink environment [8]. Simulink comes with a wide range of predefined library modules, which accelerates the development of new algorithms and shortens the development cycle considerably. The simulator is based on multi-rate time-discrete model of computation.

A typical design flow starts with the simulation and debugging of the new design and is followed by its implementation on a target system, usually an embedded processor. The design is then translated to an equivalent C description using Real-Time Workshop (RTW), a Simulink add-on which generates C code for embedded target architectures.

In order to integrate the XPP architecture into the Simulink design flow, both aspects have been considered: co-simulation and code generation for a specific target platform. The goal is to create a custom IP library of XPP modules that implement popular DSP algorithms and can be instantiated like standard Simulink library modules. This integration will enable designers to accelerate the development of applications that include blocks mapped on an XPP architecture.

4 XPP Co-simulation in Simulink

Simulink provides a very powerful mechanism for extending its functionality by supporting user-defined modules to implement new algorithms, which opens unlimited possibilities of co-simulation with other environments. To allow the co-simulation of XPP modules within Simulink, such a custom module has been developed, which acts as a wrapper for the XPP simulator (XSIM).

4.1 Communicating with the XPP Simulator

XSIM is a cycle-accurate simulator for the XPP architecture and can operate in stand alone interactive mode or as a client in conjunction with a controlling process (server). When in client mode, it communicates with the controlling process through TCP/IP to exchange messages, debug visualization data, and I/O streams, as shown in Fig. 5.

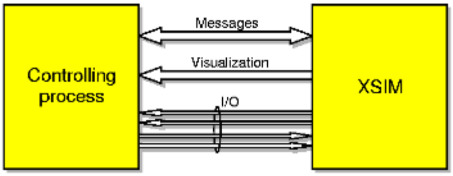


Fig. 5. XPP in client mode.

The following list shows a typical simulation flow with XSIM in client mode:

- Open two TCP/IP servers sockets, one for messages and one for visualization, start XSIM as a child process, and wait for XSIM to connect to the servers.
- Load a configuration file.
- Define I/O ports and external RAMs in accordance with the configuration file. A separate TCP/IP connection will be created for each port.
- Write data to input ports, advance the simulation, and read output data when available. Read also the debug visualization data and write the content to a file. This step is performed repeatedly.
- At the end of simulation, quit the simulator and close all TCP/IP connections.

The messages are in human readable format and easy to parse. Part of the interface are messages for loading configuration files, configuring ports, advancing and stopping the simulation, controlling breakpoints, set and read port and RAM data, etc. Visualization data files can become very large during simulation and can be disabled if not needed.

4.2 Custom Modules in Simulink

The Simulink library can be extended through the mechanism of S-functions [9]. They define the properties of the new custom module by implementing a number of callback functions which are called by the Simulink framework at various stages during simulation. This ensures a standard interface with Simulink and defines the complete functionality of the module.

Typically, S-functions can be developed using Matlab, C, C++, FORTRAN or ADA and are compiled as a dynamically linked library which exports functions called by the Simulink framework. For further details on S-functions, refer to [9]. Because of its high flexibility and efficiency, we have chosen to implement our XPP wrapper using C++. There are almost no limitations for the functionality of a module, as long as the standard interface is observed.

For a fixed-step simulation, as it is the case in all discrete systems, a minimum of four callback functions have to be implemented, as shown in Fig. 6. For more complex modules with dynamic run-time defined behavior, more functions have to be implemented. For our XPP wrapper module, we have determined that the four-function model is sufficient.

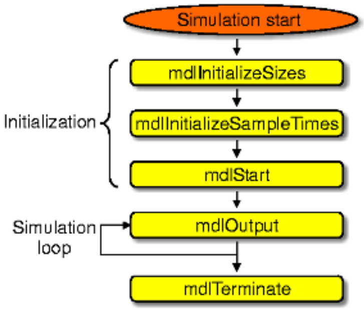


Fig. 6. Simulink callback functions

The number of input and output ports, their size and type, as well as the sampling rates are declared in the initialization phase. The actual functionality is implemented in the *mdlOutput* function, which is called repeatedly in the simulation loop, once every sampling period. If objects have been allocated at initialization or during the simulation, *mdlTerminate* is the place to perform clean-up. More details can be found in [9].

4.3 The XPP Wrapper Module

In order to ensure a consistent interface with the XPP simulator, a C++ class has been developed, which encapsulates the data exchange and the message protocol, handling all the interactions between Simulink and the simulator. The class offers methods for sending commands to XSIM, reading the status, declaring data and event ports, read visualization data, and send/receive data and event samples.

In each phase of the simulation – initialization, simulation loop, and termination – the Simulink framework calls the standard functions in Fig. 6 that the custom module exports. Besides Simulink-specific code, these functions contain calls to the methods of the wrapper class to implement the functionality required for co-simulation. The individual simulation steps are detailed in Fig. 7.

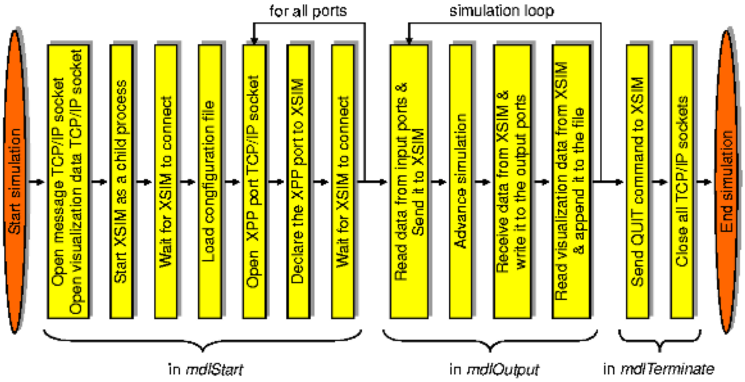


Fig. 7. Detailed simulation dataflow

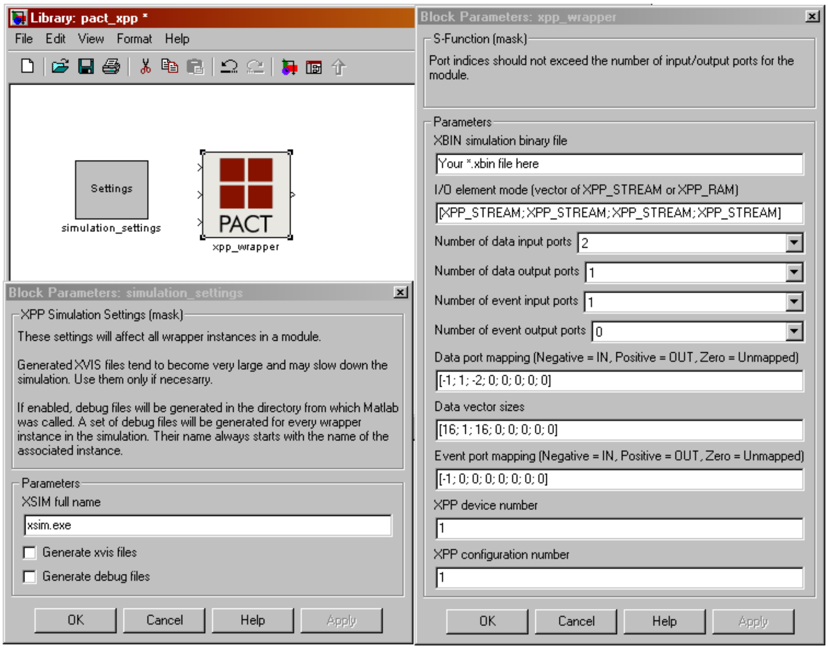
A potential problem lies in the fact that the XPP architecture processes data as a stream in a self timed and self synchronizing manner, being natively suitable for the integration with an environment that supports an untimed dataflow model of computation. The processed data becomes available at the output a certain number of cycles after all necessary data samples have been provided at the input.

In the case of an 64-point FFT for instance, all 64 complex input samples have to be provided before the result can be computed and the first result sample made available at the output. We say in this case that data is processed in blocks. Moreover, the computation delay is not of concern and the applications should not rely on a specific value thereof. Everything that needs to be known is the size of the input and output data blocks, without any reference to timing. For some applications, the size of the result is not fixed, but depends on the input data itself.

This aspects pose a problem to the integration with an environment which only supports a discrete-time model of computation, such as Simulink, where each module has to generate some output every sampling cycle. Our solution was to let the XPP array process one block of data every sampling cycle. Block sizes greater than one are supported by using vector signals.

In the case of a digital filter, the block size is one, since one input sample is enough to produce one result sample. For an FFT, the block size is the number of points on which the FFT is performed. No result samples are produced unless all input samples are available. Since port sizes in Simulink are fixed during a simulation, the block sizes of the data expected and generated by the XPP array have to be design-time constants. XPP modules that produce dynamically sized output data, e.g. a Huffman encoder, are not supported.

The XPP wrapper module behaves like a built-in Simulink module and is included in a dedicated Simulink library, from where it can be instantiated in new designs. In the same library there is also a special module without ports, which is used for specifying simulation-specific settings. One such a module has to be instantiated in every design. However, multiple wrapper modules can be instantiated. In order to support different port mapping, sizes, and configurations, the wrapper module is generic and run-time parameterizable.



Block Parameters: simulation_settings

XPP Simulation Settings (mask)

These settings will affect all wrapper instances in a module.

Generated XVIS files tend to become very large and may slow down the simulation. Use them only if necesarry.

If enabled, debug files will be generated in the directory from which Matlab was called. A set of debug files will be generated for every wrapper instance in the simulation. Their name always starts with the name of the associated instance.

Parameters

XSIM full name: xsim.exe

□ Generate xvis files

□ Generate debug files

Block Parameters: xpp_wrapper

S-Function (mask)

Port indices should not exceed the number of input/output ports for the module.

Parameters

XBIN simulation binary file: Your *.xbin file here

I/O element mode (vector of XPP_STREAM or XPP_RAM): [XPP_STREAM;XPP_STREAM;XPP_STREAM;XPP_STREAM]

Number of data input ports: 2

Number of data output ports: 1

Number of event input ports: 1

Number of event output ports: 0

Data port mapping (Negative = IN, Positive = OUT, Zero = Unmapped): [-1; 1; -2; 0; 0; 0; 0; 0]

Data vector sizes: [16; 1; 16; 0; 0; 0; 0; 0]

Event port mapping (Negative = IN, Positive = OUT, Zero = Unmapped): [-1; 0; 0; 0; 0; 0; 0; 0]

XPP device number: 1

XPP configuration number: 1

Fig. 8. Settings for the wrapper library

The parameters can be specified upon instantiation in a design by double-clicking on the generic wrapper module. Fig. 8 shows the parameters for both the wrapper and the configuration module, such as the configuration file, the number of data and event ports, the mapping of the XPP I/O elements to the Simulink ports, and the block size of the data ports. A test is performed when the parameters are changed, to ensure the consistency of values.

Based on the generic wrapper module, specialized instances are created for every application available on XPP, such as FFT, digital filters, Viterbi decoders, Rake receivers, OFDM receivers, etc, which are made available as an IP library for signal processing. Although permitted, designers will not instantiate the generic wrapper module directly, but use the specialized modules instead.

5 Conclusions

We have presented the XPP reconfigurable architecture for multimedia and wireless developed at *PACT XPP Technologies*, together with its associated design flow. As an extension to the existing design flow, we have developed a solution for integrating the architecture in a Simulink design flow. Of the two aspects involved in the integration, namely the co-simulation and the automated code generation for a target platform, this paper dealt with the former, while the latter will make the object of another paper.

In order to enable the co-simulation, a special Simulink wrapper module has been developed, which communicates with the XPP simulator. The wrapper is generic and can be parameterized for specific configurations which are part of an IP library. Including the XPP IP blocks early in the design process helps reducing the development time and increase productivity. In conjunction with the automated code generation, this allows a complete application to be designed and prototyped in a matter of hours.

References

1. PACT XPP Technologies: The XPP White Paper. (2002)
2. PACT XPP Technologies: XDS User Manual: Using the Mapper, the Simulator, and the Visualizer. (2002)
3. Baumgarte, V., May, F., Nückel, A., Vorbach, M., Weinhardt, M.: PACT XPP - a self-reconfigurable data processing architecture. The Journal of Supercomputing **26** (2003)
4. Hartenstein, R., Kress, R., Reinig, H.: A new FPGA architecture for word-oriented datapaths. Field-Programable Logic; 4th International Workshop (1994)
5. Ebeling, C., Cronquist, D.C., Franklin, P.: RaPiD - reconfigurable pipelined datapath. Field-Programable Logic; 6th International Workshop (1996)
6. Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P.: Baring it all to software: Raw machines. IEEE Computer (1997) 86–93
7. Helmschmidt, J., Schüler, E., Rao, P., Rossi, S., di Mateo, S., Bonitz, R.: Reconfigurable signal processing in wireless terminals. Design, Automation and Test in Europe Conference (2003) 244–249
8. The MathWorks: Simulink Reference. (2003)
9. The MathWorks: Writing S-Functions. (2003)
10. Becker, J., Thomas, A., Vorbach, M., Baumgarte, V.: An industrial/academic configurable system-on-chip project (CSoC): Coarse-grain XPP-/Leon-based architecture integration. Design, Automation and Test in Europe Conference (2003) 1120–1121

An FPGA Based Coprocessor for the Classification of Tissue Patterns in Prostatic Cancer

M.A. Tahir, A. Bouridane, and F. Kurugollu

Queens University of Belfast
BT71NN, Northern Ireland
a.tahir@qub.ac.uk

Abstract. This paper discusses the suitability of reconfigurable computing to speedup medical image classification problems. As an example of the speedup offered by reconfigurable logic, a multispectral computer vision system for automatic diagnosis of prostatic cancer is implemented. Different parallel architectures for various steps in automatic diagnosis are proposed and implemented in Field Programmable Gate Arrays (FPGAs). The first step of the algorithm is to compute Grey Level Cooccurrence Matrix (GLCM). The second step involves the normalisation of GLCM. The third step of the algorithm is to compute texture features from the normalised GLCM. The last step is concerned with image classification using linear discriminant analysis (LDA). Finally, the performance of the proposed system is assessed and compared against a microprocessor based solution. The results obtained clearly show that the proposed solution compares favorably.

1 Introduction

Image processing applications usually require the processing of large amounts of data, especially after the introduction of multispectral and scanscope images [1, 2]. In multispectral images, instead of analyzing conventional grey scale or RGB color images, multiple bands of a object are created using light from different parts of the spectrum [1]. Figure 1 shows two bands of a textured multispectral medical image. In scanscope images, for a $15mm * 15mm$ tissue section, approximately 2.5 gigabytes (GB) of data must be acquired. Much effort has therefore gone into designing high performance architectures tailored to image processing applications. The aim of this paper is to develop an FPGA based co-processor for the classification of tissue patterns in prostatic cancer using multispectral medical images. Figure 2 shows the steps involved for prostate cancer classification [1].

The first step of the algorithm is to compute GLCM to extract Haralick texture features [3,4]. Step 2 involves the normalization of GLCM. The third step of the algorithm is to compute texture features from normalized GLCM. Since different regions in tissue section images can be classified as cancer or normal using

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 771–780, 2004.

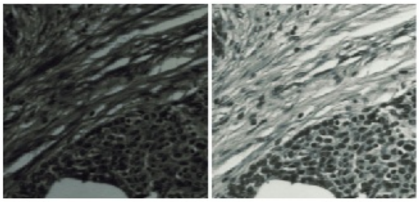


Fig. 1. Two bands of a multispectral image showing pathological section of a prostate biopsy.

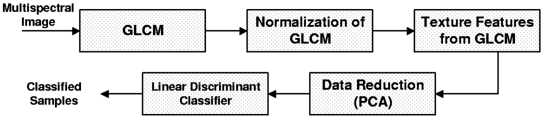


Fig. 2. Algorithm for the classification of prostate tissue cancer.

texture features, these features are measured in a large number of sub-regions inside each band of the multispectral image. Thus, for each subregion, the computation of GLCM is required along with the computation of texture features. Step 4 is a data reduction step which is used to avoid curse-of-dimensionality problem. Step 5 involves the use of a classifier for classification [1,5].

The overall calculations for the computation of GLCM and texture features are computationally intensive. For an image of size 5000*5000 with 16 bands, the time required is 350 seconds using Pentium 4 machine running at 2.4 GHz. 75% of the total time spent is for the calculation of GLCM, 5% for the normalisation, 19% for the calculation of texture features while 1% is for the classification using classical discrimination method. There are different images for each patient and even the number of patients can vary. The Von Neumann style of fetch-operate-writeback computation fails to exploit the inherent parallelism in computing GLCM and Haralick features. Therefore, a hardware intensive parallel implementation is needed. Furthermore, by implementing this application on FPGAs, hardware solutions are also provided for various important algorithms in image processing such as GLCM, Haralick Texture Features, and Classification using discriminant analysis.

The target hardware for this work is a Celoxica RC1000-PP PCI based FPGA development board equipped with a Xilinx XCV2000E Virtex FPGA having 19,200 slices and 655,360 bits of block RAM, and four banks of static RAM with 2MB each [7,8].

The paper is organized as follows. Section 2 reviews GLCM, Haralick texture features and linear discriminant classifier. Section 3 describes the proposed system model. Experiments and discussion are presented in Section 4. Section 5 concludes the paper.

2 GLCM, Haralick Texture Features, Data Reduction, and Linear Discriminant Classifier

Grey Level Co-Occurrence Matrix (GLCM): GLCM, first introduced by Haralick [3], is a powerful technique for measuring texture features. The texture-context information is specified by the matrix of relative frequencies $P_{i,j}$ with two neighboring pixels separated by a displacement d and an angle θ. The GLCM is calculated with the following equation [9];

$$\begin{aligned} P(i,j,d,\theta) = \#\{(x_1,y_1)(x_2,y_2)|f(x_1,y_1)=i, f(x_2,y_2)=j, \\ |(x_1,y_1)-(x_2,y_2)|=d, \angle((x_1,y_1),(x_2,y_2))=\theta\} \end{aligned} \quad (1)$$

where # is the number of occurrences inside the window sizes where the intensity level of a pixel pair changes from i to j, the location of the first pixel is (x_1, y_1) and that of the second pixel is (x_2, y_2), d is the distance between the pixel pair, θ is the angle between the two pixels. The co-occurrence matrix so defined is not symmetric. If the GLCM is calculated with symmetry, then only angles up to 180^0 need to be considered. A symmetric co-occurrence matrix can be computed by the expression $P(i,j,d,\theta)' = (P(i,j,d,\theta) + P(i,j,d,\theta)^T)$ / 2 where $P(i,j,d,\theta)^T$ is the transpose of $P(i,j,d,\theta)$. Probability estimates are obtained by dividing each entry in $P(i,j,d,\theta)'$ by the sum of all possible intensity changes with the distance d and direction θ.

Haralick Features: Let $P(i,j,d,\theta)$ is a (normalised) frequency of occurrence of grey level pair (i,j) at distance d and angle θ and N_g be the number of gray levels.

$$f_1(d,\theta) = ASM = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} P(i,j,d,\theta)^2 \quad (2)$$

$$f_2(d,\theta) = \mu = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} iP(i,j,d,\theta) \quad (3)$$

$$f_3(d,\theta) = \sigma = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g}((i-\mu)^2 P(i,j,d,\theta))^{1/2} \quad (4)$$

$$f_{4(d,\theta)} = Corr = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} \frac{(i-\mu)(j-\mu)P(i,j,d,\theta)^2}{\sigma^2} \quad (5)$$

$$f_{5(d,\theta)} = Cont = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g}(i-j)^2 P(i,j,d,\theta) \quad (6)$$

$$f_{6(d,\theta)} = Diss = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g}(i-j) P(i,j,d,\theta) \quad (7)$$

$$f_{7(d,\theta)} = Ent = -\sum_{i=1}^{N_g}\sum_{j=1}^{N_g} P(i,j,d,\theta)\log P(i,j,d,\theta) \quad (8)$$

where ASM, μ, σ, Corr, Cont, Diss, Ent are angular second moment, mean, variance, correlation, contrast, dissimilarity, and entropy respectively.

Data Reduction: Normally, the total number of features used in the classification process using multispectral images is greater than 100. The accurate estimation of statistical parameters requires the rate (number of features / number of samples) to be as low as possible. Therefore, a data reduction step, such as Principal Component Analysis (PCA), is used [5]. Mathematically;

$$\begin{pmatrix} fv_1 \; fv_2 \; . \; . \; fv_k \end{pmatrix} = \begin{pmatrix} f_1^{'} \; f_2^{'} \; . \; . \; f_d^{'} \end{pmatrix} * \begin{pmatrix} c_{11} & c_{12} & . \; . & c_{1k} \\ c_{21} & c_{22} & . \; . & c_{2k} \\ . & . & & . \\ c_{d1} & c_{d2} & . \; . & c_{dk} \end{pmatrix} \tag{9}$$

where $f_1^{'} f_2^{'} .. f_d^{'}$ = standardized feature vector with dimension d, $fv_1 fv_2 .. fv_k$ = reduced feature vector with dimension k, and c_{11}.... is the Component Score Coefficient Matrix [10,11] obtained after applying PCA on training samples to reduce the dimension. The standardized feature vector $f_1^{'}$ is obtained by subtracting the original feature vector f_1 with mean variable and dividing by the standard deviation variable. Both mean and standard deviation variables are calculated for each feature variable during the training stage. In order to simplify the calculations during the testing process (i.e. to avoid division and substraction in equation 10), the equation 10 can be reduced to equation 11 as follows

$$fv_1 = f_1^{'} * c_{11} + f_2^{'} * c_{21} + ... + f_d^{'} * c_{d1} \tag{10}$$

$$fv_1 = \frac{f_1 - m_{f_1}}{std_{f_1}} * c_{11} + \frac{f_2 - m_{f_2}}{std_{f_2}} * c_{21} + ... + \frac{f_d - m_{f_d}}{std_{f_d}} * c_{d1}$$

$$fv_1 = f_1 * c_{11}^{'} + f_2 * c_{21}^{'} + ... + f_d * c_{d1}^{'} - m_{f_1} * c_{11}^{'} - m_{f_2} * c_{21}^{'} - m_{f_d} * c_{d1}^{'}$$

$$fv_1 = f_1 * c_{11}^{'} + f_2 * c_{21}^{'} + ... + f_d * c_{d1}^{'} - g_1 \tag{11}$$

where $c_{d1}^{'} = \frac{c_{d1}}{std_{f_d}}$ and $g_1 = m_{f_1} + m_{f_2} + ... + m_{f_d}$.

$$fv_2 = f_1 * c_{21}^{'} + f_2 * c_{22}^{'} + ... + f_d * c_{d2}^{'} - g_2 \tag{12}$$

$$.........$$

$$fv_k = f_1 * c_{d1}^{'} + f_2 * c_{d2}^{'} + ... + f_d * c_{dk}^{'} - g_k \tag{13}$$

where, $(g_1, g_2, ... , g_k)$ are constants. The simplification of equation 9 results in an efficient hardware avoiding many division and subtraction.

Linear Discriminant Analysis: Linear discriminant analysis (LDA) is a classical statistical approach for classifying samples of unknown classes, based on training samples with known classes. For a two class problem, linear discriminant analysis can be defined mathematically as: if $fv_1 * y_1 + fv_2 * y_2 + .. + fv_k * y_k > 0$ then, the sample belongs to class1 else it belongs to class2. $y_1, ...y_k$ are Canonical Discriminant Function Coefficients obtained during training [10,12].

3 Proposed System Architecture

The proposed system architecture for the classification of prostate cancer is shown in Figure 3. Different parallel architectures for each stage are proposed (Figures 4 to 7). At its most basic level, the programming model for our image processing machine is a host processor (typically a PC running on 2.4 GHz Pentium 4-based system, programmed in C++). The host machine works as a

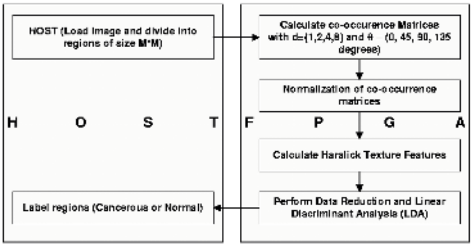

HOST (Load image and divide into regions of size M*M)
Calculate co-occurence Matricse with d={1,2,4,8} and θ (0, 45, 90, 135 degrese)
Normalization of co-occurrence matrices
H O S T
F P G A
Calculate Haralick Texture Features
Perform Data Reduction and Linear Discriminant Analysis (LDA)
Label regions (Cancerous or Normal)

Fig. 3. System architecture.

control unit for loading input data required for each stage in the FPGA external memory. In the first stage, the input to the FPGAs are the different bands of multispectral images. Each band is divided into regions of size *N*N*. The input to the second stage and third stage are *B*R* co-occurrence matrices with $\theta = \{0^o, 45^o, 90^o, 135^o\}$ where B is the number of bands in multispectral images and R is the number of regions for each band. The input to the fourth and last stage are the features obtained from stage 3. The output from stage 4 are the different regions in the image that are classified as stroma and cancer.

Proposed Architecture for Calculating GLCM: The block diagram for calculating GLCMs on FPGA is shown in Figure 4(a) [13]. The inputs to the FPGAs are image regions of size *N*N* loaded into memory banks 0,1,2,3 for bands B_j, B_{j+1}, B_{j+2}, B_{j+3}, respectively. The main reason of distributing different images (bands) in memory banks is to achieve a maximum parallelism. Pixel i is read simultaneously from each memory bank for different bands. Neighboring pixels are then accessed simultaneously. Thus, all neighboring pixels are accessed in 4 clock cycles. The memory addresses for all 16 GLCMs (bands B_j, B_{j+1}, B_{j+2}, B_{j+3} with $\theta = 0, 45, 90, 135$ i.e. 4*4) are calculated in parallel followed by an updating of the number of occurrences of pixels in the co-occurrence matrix. The process is repeated for each pixel of the entire image. After the calculation of GLCMs, the normalisation of GLCM is required. The flow diagram for the normalisation of GLCMs is shown in Figure 4(b). For simplicity reasons, the normalisation of only one GLCM (for band j and θ=0) is illustrated. All other GLCMs are also normalised simultaneously. In order to normalize the GLCM, element $P(i, j, d, \theta)$ is first added with $P^T(i, j, d, \theta)$. The result is then divided by the expression (two times the sum of all possible intensity changes with direction θ) to compute the estimation of probability [9,13]. Since, the above expression is constant, a constant multiplier is used instead of a divider by taking the reciprocal of the expression. The final result is stored in the external memory. If there are more elements in GLCM, the process is repeated. Other GLCMs are also normalised simultaneously. Thus, all GLCMs are calculated and normalised in parallel. The process of calculating and normalising GLCM is then looped for the remaining bands and regions of the multispectral images.

Proposed Architecture for Haralick Texture Features Computation: The block diagram of the proposed architecture for extracting the texture

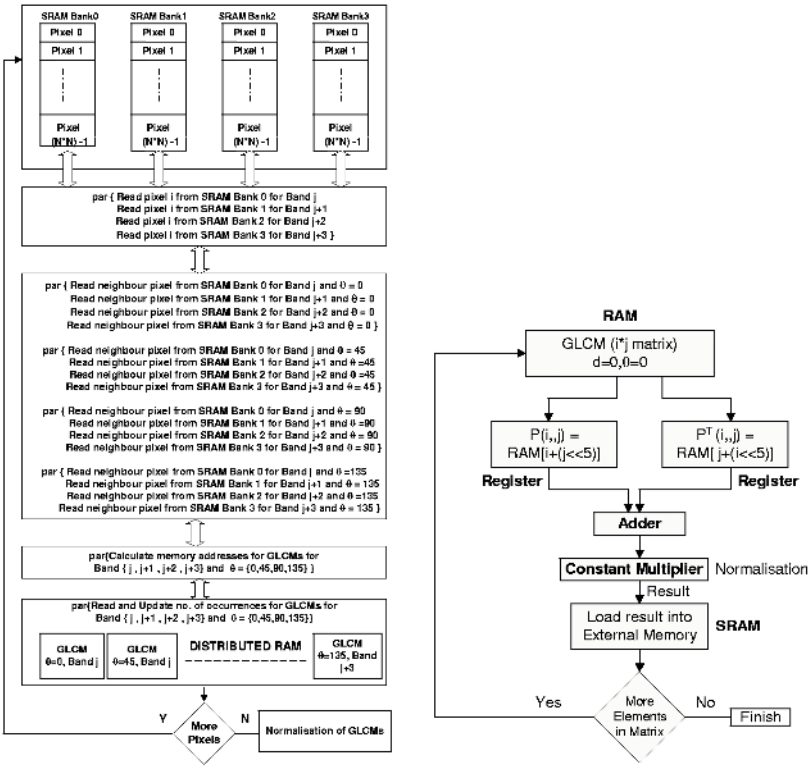

SRAM Bank0
SRAM Bank1
SRAM Bank2
SRAM Bank3
Pixel 0
Pixel 1
Pixel (N*N) -1
par { Read pixel i from SRAM Bank 0 for Band j
Read pixel i from SRAM Bank 1 for Band j+1
Read pixel i from SRAM Bank 2 for Band j+2
Read pixel i from SRAM Bank 3 for Band j+3 }
par { Read neighbour pixel from SRAM Bank 0 for Band j and θ = 0
Read neighbour pixel from SRAM Bank 1 for Band j+1 and θ = 0
Read neighbour pixel from SRAM Bank 2 for Band j+2 and θ = 0
Read neighbour pixel from SRAM Bank 3 for Band j+3 and θ = 0 }
par { Read neighbour pixel from SRAM Bank 0 for Band j and θ = 45
Read neighbour pixel from SRAM Bank 1 for Band j+1 and θ =45
Read neighbour pixel from SRAM Bank 2 for Band j+2 and θ =45
Read neighbour pixel from SRAM Bank 3 for Band j+3 and θ = 45 }
par { Read neighbour pixel from SRAM Bank 0 for Band j and θ = 90
Read neighbour pixel from SRAM Bank 1 for Band j+1 and θ =90
Read neighbour pixel from SRAM Bank 2 for Band j+2 and θ = 90
Read neighbour pixel from SRAM Bank 3 for Band j+3 and θ = 90 }
par { Read neighbour pixel from SRAM Bank 0 for Band j and θ =135
Read neighbour pixel from SRAM Bank 1 for Band j+1 and θ = 135
Read neighbour pixel from SRAM Bank 2 for Band j+2 and θ =135
Read neighbour pixel from SRAM Bank 3 for Band j+3 and θ = 135 }
par{Calculate memory addresses for GLCMs for Band { j , j+1 , j+2 , j+3} and θ = {0,45,90,135})
par{Read and Update no. of occurrences for GLCMs for Band { j , j+1 , j+2 , j+3} and θ = {0,45,90,135})
GLCM θ=0, Band j
GLCM θ=45, Band j
DISTRIBUTED RAM
GLCM θ=135, Band j+3
Y
More Pixels
N
Normalisation of GLCMs
RAM
GLCM (i*j matrix) d=0,θ=0
P(i,,j) = RAM[i+(j<<5)]
P^T (i,,j) = RAM[j+(i<<5)]
Register
Register
Adder
Constant Multiplier
Normalisation
Result
Load result into External Memory
SRAM
Yes
More Elements in Matrix
No
Finish

Fig. 4. (a) Block diagram for calculating GLCM on FPGA. (b)Flow Chart for the normalisation of GLCMs on FPGA.

features on FPGA is shown in Figure 5 [14]. Each processing unit, which operates in parallel, consists of a set of adders and multipliers as shown in Figure 6 and is used to calculate the features described in equations 2-8 for a particular angle θ. The final processing unit, consisting of adders and shift registers, is used to calculate the average of each feature for different angle θ. This results in 7 features as mentioned in section 2 for region R and band B. These features are then stored in SRAM (Bank 0). This process is looped Y times where $Y = R*B$. Figure 6 shows the block diagram of the processing unit indicated in Figure 5. There are two stages in this block diagram. Mean, contrast, dissimilarity, and entropy are calculated in the first stage and four Processing Elements (PEs) are used for their calculations. These PEs, consisting of parallel multipliers and adders, operate simultaneously. In order to speed up the computation, the constants $(i - j)$ and $(i - j)^2$ of equations 6 and 7 are pre-computed and stored in ROMS. Log tables, which are stored in Block RAMS, are used for the calculation of log function involved in the entropy computation. Angular second moment, variance and correlation are calculated in the second stage as variance and correlation depend upon the mean value.

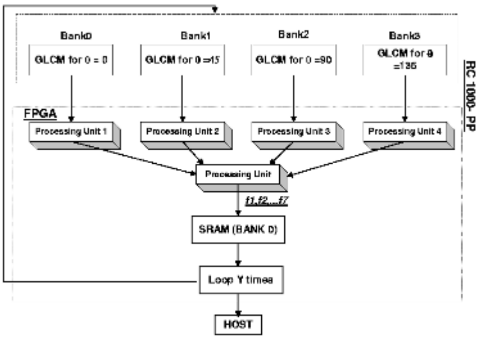


Fig. 5. Block diagram for extracting Haralick features in FPGA.

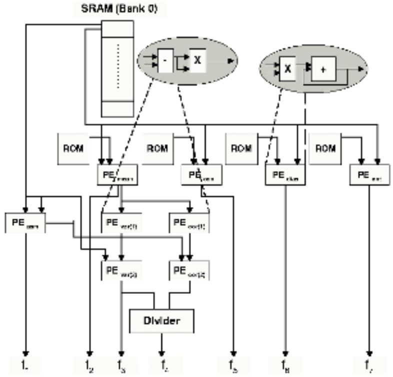


Fig. 6. Block diagram of Processing Unit 1 as shown in Figure 5. The other processing units 2-4 have the same architecture and are executed in parallel.

Proposed Architecture for data reduction and linear discriminant analysis: Figure 7 shows the hardware design for data reduction and LDA with $k = 5$. The feature vector is distributed into 4 different banks of SRAMs. Each feature is multiplied by k different constants. Thus, 20 PEs (4 features in one iteration * 5 constants) are executed in parallel to perform the multiplication and accumulation. The coefficients stored in the ROM are also distributed such that they are not duplicated. The process is repeated F times where the value of F is $(numberoffeatures)/4$. Once this is done, the results from PE1, PE6, PE11, and PE16 are added using adder1 and then constant g_1 is subtracted to compute fv_1 as mentioned in equation 11. The results of $fv_2......fv_k$ can be obtained in the similar way. The constant multipliers are used for LDA and the results from the constant multipliers are added to determine whether region i is cancerous or normal. The process is then repeated if there are more regions to be classified.

The proposed architecture has been implemented by using Handel-C [7]. Handel-C is a truly innovative C-like language for implementing algorithms in

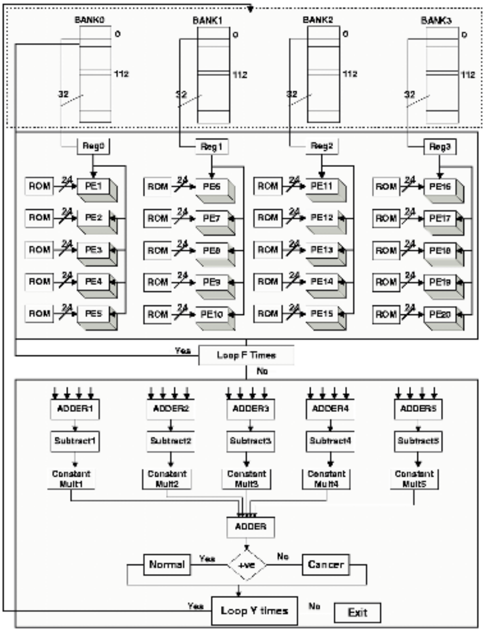


Fig. 7. Hardware Architecture for data reduction and linear discriminant analysis. The value of F is (No. of features) /4, while the value of is the number of samples to be classified.

hardware. The output from Handel-C is a file that is used to create the configuration data for the FPGA.

4 Experiments and Discussion

The original testing material consists of textured multispectral images taken at 16 spectral channels (from 500nm to 650) [1]. For testing the classification accuracy, the features used for training are computed on the FPGA co-processor. 304 different samples have been used to carry out the analysis. They have been assessed by two highly experimented pathologists and labelled onto 2 groups: 128 cases of Stroma (muscular normal tissue), 176 cases of Prostatic Carcinoma: PCa(abnormal tissue development corresponding to cancer). The assessment of classification results has been made using Leave-One-Out method [5,10]. Table 1 shows the overall classification error. This error is the same when the same features are re-calculated using C++ and classified using LDA.

Table 2 shows the execution time comparison between μP-based and FPGA-based implementation for various image sizes for the calculation of GLCM, Texture Features, and data reduction (PCA) & LDA. The clock speed of the FPGA architecture for calculating GLCM is 50MHz, the clock speed for calculating

Table 1. Classification error.

Classified as	PCa	STROMA	Error
PCa	171	5	2.8%
STROMA	3	125	2.3%
Overall			2.55%

GLCM texture features is 45MHz, and the clock speed for performing data reduction (PCA) and LDA is 44MHz. The clock speed for Pentium 4 is 2.4GHz. The result shows that the performance of FPGA in calculating GLCM is approximately 5 times faster than that of Pentium 4 PC even though the PC platform has a clock speed which is 45 times faster. This improvement in the performance is mainly due to the calculation of different GLCMs in parallel. Also, the result shows that the performance of FPGA in calculating Haralick Texture Features is approximately 7 times faster than Pentium 4 PC. This is mainly due to the parallel architecture of FPGA that results in more multiplications and additions on every clock cycle than Pentium 4. Finally, the performance of FPGA in performing PCA&LDA is 11 times better than Pentium 4 PC as 20 multiplications and additions are performed in parallel in this case. Table 3 shows the area used by different FPGA architectures. The speed-grade of the FPGA used for the experimental results is 6.

Table 2. Comparison of Execution Time between μ-P and FPGA implementation for GLCM, Haralick Texture Features, and PCA&LDA. S = Speed-Up.

Image Size	GLCM			Texture Features			PCA&LDA		
(16 Bands)	Time in sec			Time in sec			Time in msec		
	μ-P	FPGA	S	μ-P	FPGA	S	μ-P	FPGA	S
512*512	3.0	0.63	4.75	0.710	0.097	7.3	0.54	0.049	11.0
1024*1024	11.9	2.5	4.75	2.840	0.388	7.3	2.16	0.196	11.0
2048*2048	47.6	10.0	4.75	11.360	1.550	7.3	8.65	0.785	11.0

Table 3. Area used for calculating GLCM, GLCM Texture Features, and PCA&LDA.

	GLCM	Texture Features	PCA&LDA
Number of occupied Slices	59%	86%	73%
Number of slice flip flops	4%	16%	9%
Total number of 4 input LUTs	57%	70%	63%
Number of bonded IOBs	65%	59%	65%
Number of block rams	-	76%	-

5 Conclusion and Future Work

Reconfigurable architectures have many applications in image classification. Depending on the classifier, reconfigurability can assist in speeding up classification process. An FPGA based multispectral computer vision system for diagnosis of prostate cancer is proposed in this paper. Different parallel architectures for various steps in automatic diagnosis have been proposed and implemented on FPGAs. The FPGA-based version is 5 times faster when compared with μ-processor for calculating GLCMs, 7 times faster for calculating texture features while 11 times faster for dimension reduction and linear discriminant analysis. This was mainly achieved due to various parallel architectures used for different steps in automatic diagnosis. Future work is concerned with an implementation of these proposed architectures using new Celoxica RC2000-PP PCI based FPGA development board in order to further improve the performance both in terms of area and speed. This board is equipped with a Xilinx Virtex II FPGA and six banks of static RAM with 2MB each.

References

1. M. A. Roula et al. A Multispectral Computer Vision System for Automatic Grading of Prostatic Neoplasia. IEEE International Symp. on Biomedical Imaging, (2002).
2. URL: http://www.scanscope.com.
3. R. M. Haralick, K. Shanmugam, and I. Dinstein. Textural Features for Image Classification. IEEE Trans. on Systems, Man, and Cybernetics, **3(6)**, (1973) 610–621.
4. R.W. Conners and C.A. Harlow. A Theoretical Comaprison of Texture Algorithms. IEEE Trans. on Pattern Analysis and Machine Intelligence, **2**, (1980), 204-222.
5. R.O. Duda, P. E. Hart, D.G. Stork. Pattern Classification, Willey-Interscience, Second Edition, (2001).
6. A. Sharma. Programmable Logic Handbook, PLDs, CPLDs and FPGAs, McGraw-Hill, (1998).
7. URL: http://www.celoxica.com.
8. URL: http://www.xilinx.com.
9. K. Wikantika, A.B. Harto, and R. Tateishi, The use of spectral and textural features from Landsat TM image for land cover classification in mountainous area. Proceedings of the IECL Japan workshop, Tokyo, (2001).
10. URL: http://www.spss.com.
11. H. H. Harman. Modern factor analysis, 3rd ed., Chicago: University of Chicago Press, (1976).
12. M. M. Tatsuoka, Multivariate analysis, New York: John Wiley and Sons, (1971).
13. M. A. Tahir, A. Bouridane, F. Kurugollu, and A. Amira. An FPGA based coprocessor for calculating grey level cooccurrence matrix. Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems, (2003).
14. M. A. Tahir et al. An FPGA based co-processor for GLCM texture features measurement. Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems, (2003).

Increasing ILP of RISC Microprocessors Through Control-Flow Based Reconfiguration

Steffen Köhler, Jens Braunes, Thomas Preußer,
Martin Zabel, and Rainer G. Spallek

Institute of Computer Engineering
Department of Computer Science
Dresden University of Technology
D-01062 Dresden, Germany
{stk,braunes,preusser,zabel,rgs}@ite.inf.tu-dresden.de

Abstract. This work introduces a new concept of enhancing a RISC microprocessor with a tightly coupled reconfigurable ALU array, a vector load/store unit and a control flow manipulation unit. These units implement coarse-grain reconfigurable structures by means of switchable contexts. Context activation is performed event-driven according to the instruction pointer of the RISC microprocessor. The synchronous operation of the context controlled functional units enables instruction level parallelism (ILP) comparable to complex VLIW processors, without introducing instruction overhead. The reconfigurable units can be adapted to the application demands exploiting parallelism more coarse-grain than common instruction-level functional units. To evaluate the concept, a standard ARM RISC microprocessor was chosen to be tightly coupled to these reconfigurable units. Architecture description and simulation were performed using RECAST, a reconfiguration-enabled architecture description language and simulation tool-set. The software environment also includes a retargetable, parallelizing C compiler based on the SUIF compiler kit. First experiments executing DSP algorithms have indicated, that the proposed architecture can exploit more of the potential application parallelism than conventional VLIW processors.

1 Introduction

The data processing for today's media and telecommunication applications can be efficiently provided by application-specific digital signal processors. Due to high implementation and mask costs the performance of these DSP is not always acceptable to be only sufficient for a very limited scale of special algorithm implementation. Another important fact is the lack of available/usable instruction level parallelism leading to an insufficient average utilization of the functional units in state-of-the-art VLIW or SIMD architectures [16]. This indicates a low functional execution density and thus a waste of chip area. Reconfiguration-enabled functional units have shown a way towards more flexible and efficient implementations of application-specific microprocessor designs [4][5][6]. On the

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 781–790, 2004.

other hand, the flexibility gained from reconfiguration causes additional costs for reconfigurable components as compared to an ASIC implementation. The configuration memory and the interconnect network consume a significant amount of chip area. Also, in many cases a not negligible number of reconfigurable blocks may remain unutilized because of application irregularity or resource constraints. The additional effort necessary for reconfiguration has to be compensated by savings concerning the hard-wired components.

In this paper we introduce the ARRIVE (ARm microprocessor with Reconfigurable Instruction-flow controlled Vliw Extension) architecture, which can be considered a tightly-coupled coarse-grain reconfigurable VLIW extension of a standard RISC microprocessor. In contrast to common VLIW microprocessors, the functional units are controlled by configuration contexts rather than addressed by the VLIW instruction decoder. The switching of configuration contexts is based on the control flow by observing the microprocessor's instruction fetch phase and its instruction pointer value. Thereby, we can achieve the synchronization of the execution within the RISC microprocessor core and the reconfigurable units, effectively gaining a high degree of pseudo-VLIW ILP. Through the application of this technique, a significant amount of hardware resources (VLIW decoder, instruction memory) can be saved and made available to the reconfigurable functional units.

The paper is organized as follows: Section 2 outlines typical microprocessor-based coarse-grain reconfigurable architectures. In section 3, we introduce the ARRIVE architecture, which consists of an ARM microprocessor coupled tightly to coarse-grain reconfigurable functional units. Sections 4 and 5 are dedicated to the ADL-based architecture description and compilation framework. Section 6 discusses first experimental results. Finally, section 7 concludes the paper.

2 Related Work

In the past, much research work has been carried out on coupling a standard RISC microprocessor with a fine-grain reconfigurable array [7] [8]. Because of the low efficiency in terms of area overhead and poor routability of fine-grain reconfigurable devices, coarse-grain reconfigurable arrays have become an attractive alternative. This is especially true for obtaining high-bandwidth word-level parallelism as required in many DSP applications. The greater structural regularity and data width of processing elements (PEs) and their interconnect cause a massive reduction of configuration data and time. Thus, we want to outline some typical examples of coarse-grain reconfigurable architectures coupled with a RISC processor.

REMARC [1] is a reconfigurable accelerator coupled tightly to a MIPS-II RISC processor. The reconfigurable device consists of an 8×8 array of PEs or nano processors. A nano processor has its own instruction RAM, a 16-bit ALU, a 16-entry data RAM, and 13 registers. It can communicate directly to its four neighbors and via a horizontal and a vertical bus to the nano processors in the same row or column. Different configurations of a single nano processor are held in the instruction memory in terms of instructions. A global program counter is used as an index to a particular instruction inside the instruction memory.

PACT SMeXPP [2] is based on the XPP hierarchical array of coarse-grain *Processing Array Elements* (PAEs). These are grouped into a single or multiple *Processing Array Clusters* (PACs). Each PAC has its own configuration manager, which can reconfigure the associated PAEs while neighboring PAEs are processing data. In addition, the SMeXPP is equipped with an additional ARM processor core, that is loosely coupled via a high-speed bus interface with the reconfigurable array.

MorphoSys [3] combines a RISC processor with a reconfigurable cell array consisting of 8×8 (subdivided into 4 quadrants) identical 16-bit PEs. Each PE has an ALU-multiplier and a register file and is configured through a 32-bit context word. The interconnection network comprises three hierarchical levels enabling nearest neighbor, intra-quadrant, and inter-quadrant connections.

The analysis of the existing concepts suggests that the communication between the RISC processor and the reconfigurable array should be performed using additional instructions. In case of a tightly-coupled functional unit, these instructions are often implemented as an ISA extension. ILP is not targeted in the examples mentioned above. Although it is possible to enhance performance by grouping instructions into a VLIW instruction word [4][5][6] or by dynamically scheduled superscalar execution [9], there will always be a drawback due to the increased instruction overhead. Another important issue is the partitioning problem in a loosely-coupled reconfigurable system. Application development might experience problems in case of a hardly partitionable algorithm, thus introducing a lot of communication overhead. It is obvious that the performance gain of a reconfiguration-enabled microprocessor system is limited in the considered implementations.

3 The ARRIVE Architecture

The ARRIVE architecture extends an ARMv4 RISC core [15] by three coarse-grain reconfigurable functional units (RFUs). These units are dedicated to DSP application acceleration, including support for arithmetic parallelism, data-path parallelism and hardware-based zero-overhead loop unrolling (hardware loops). In detail, the ARRIVE architecture includes a multi-context reconfigurable ALU array (RALU), a multi-context vector load/store unit (VLSU) and a single-context control-flow manipulation unit (CFMU). The RFUs are data-path coupled to the lower eight mode-independent registers of the ARM core. From the processing model's view the ARRIVE architecture can be considered a highly parallel VLIW DSP. In contrast to previous work, ARRIVE is not an ISA extension. The processing operations of the RFUs are only specified within the configuration contexts. As the number of frequently executed instructions inside inner loops is low, ILP can be easily mapped to a limited number of configuration contexts. The configuration context change is performed cycle-wise according to

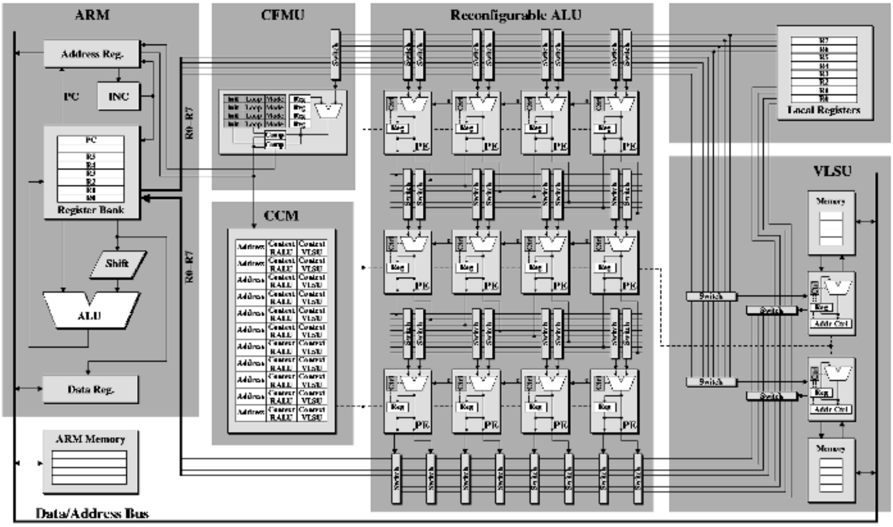


Fig. 1. ARRIVE architecture overview

the ARM instruction pointer. A context assignment table is implemented in the context configuration manager (CCM), which is capable of changing the configuration context on a matching instruction pointer value.

The Reconfigurable ALU Array is composed of coarse-grain context-controlled processing elements (PEs). Horizontal routing is provided through 8-bit wide busses. The inputs and outputs of the PEs are connected to these routing busses via a configurable switched matrix. Vertical routing is maintained only through the PEs. The PEs can be configured to execute 16 different 8-bit wide arithmetic or logic functions. Additionally, a bypass and an idle operation are implemented for routing and power-save reasons. Each PE also includes a pipeline data register. A designated carry logic can be utilized to chain the PEs horizontally into multiple of 8-bit ALU operation. The inputs of the top row are directly connected to the source register bus, whereas the outputs of the bottom row interface the destination register bus. The size of the array is currently assigned - but not limited to - 8×8 PEs, as it is required for our benchmark applications. Referring to section 2, other array architectures may also be advantageous for different application domains.

The Vector Load/Store Unit (VLSU) provides a flexible and scalable interface to dedicated external memories. It enables parallel data load/store of the ARM core registers or the RALU/VLSU local registers. Through special address registers the VLSU supports typical DSP memory transfers including pre-/post-increment/-decrement, write-back and offset address modes.

The Control Flow Manipulation Unit (CFMU) can directly modify the address register of the ARM core, thus influencing the instruction fetch process.

Two configurable counters and comparators are included to support the mapping of loops with fixed bounds.

The Configuration Context Manager (CCM) The configuration manager matches the value of the ARM address register in the instruction fetch phase with the entries in the context mapping table. In case of a positive match, it activates the associated context. Context control is implemented using a triple-stage pipeline similar to the ARM core pipeline. The independent context numbers for the RALU and the VLSU are passed to the functional units by separate configuration context busses.

4 Architecture Description

To evaluate our architecture, we developed a new reconfiguration-enabled architecture description language (ADL) and tool-set. Similar to other approaches [11][12], it supports mixed behavioral and structural modeling, enhancing description power by additional aspects for reconfigurable systems. Whereas memory hierarchy and pipeline structure are described structurally, the instruction set is modelled on a behavioral level. In matters of syntax, the designed ADL leans rather towards [11] using a C-like notation. The architecture description can be divided into three main sections describing the memory hierarchy, the pipeline structure and the instruction set. Optionally, these sections may be preceded by another one declaring references to external libraries. The description of the memory hierarchy is straight forward as it enumerates the memory components and parameterizes their attributes in a fashion much like [12]. The same holds for the description of the pipeline, which defines the available pipeline stages and their own inherent behavior independent from the instruction context executed. Typically, the instruction fetch and decode as well as the default flow of instruction contexts through the pipeline are defined here. It needs to be mentioned that the structural composition of the pipeline is not necessarily static. Specific instructions may override their routing even depending on the data they process. Additionally an instruction context may be split and routed to multiple pipeline stages, which allows the modelling of VLIW architectures. The instruction set is modelled in a hierarchical fashion as is done in [11]. The instruction set hierarchy is, however, described as inheritance tree rather than being defined in terms of a formal grammar.

For each pipeline stage as declared in the pipeline section, an instruction may define a specific behavior. The combination with the behavior possibly inherited from another instruction yields the effective behavior of an instruction as it is to be observed when executed on the architecture model. Figure 2 illustrate how this may serve the brevity of a description. An important feature of the designed ADL is that the behavioral descriptions can only be composed from a fixed set of operational statements. The behavioral semantics can thus not be blurred by vague code in some programming language or even references to external functions. Nevertheless is the inclusion of external libraries

```
// ARM Instruction Set
ISA {
  ...
  // Data Processing Instructions
  Data : Cond "00<o2_i:1><5><rn:4>
             <rd:4><-:7><rm_rs:1>
             <-:4>","$$ $rd" {
    ...
    // AND Instruction
    AND : Data "0000<scc:1>" {
    EX {
        if (ex_cycle==alu_cycle) {
          result=src1 & src2;
          ...
          if (rd==15) {
            branch_cycle=1;
            ADDRESS=result;
            => EX;
          }
          else R[rd]=result;
        }
      }
    }
  }
}
```

```
// Reconfigurable ALU
RALU (Config,op1,op2,cin,
      res0,res1,cout) {
    // ADDC op1,op2
    if (Config==0x0) {
        tmp=(op1)(0,9)+op2+cin;
        res0=tmp(0,8);
        cout=tmp(8,1);
    }
    // SUBC op1,op2
    else if (Config==0x1) {
        tmp=(op1)(0,9)+(~op2)+1;
        result=tmp(0,8);
        cout=tmp(8,1);
    }
    // NOT op1
    else if (Config==0x2) {
        res0=~op1;
        cout=cin;
    }
    // AND op1,op2
    else if (Config==0x3) {
        result=op1 & op2;
        cout=cin;
    }
}
```

Fig. 2. ADL description of the instruction set and the reconfigurable ALU

supported. These can, however, only be used to provide custom implementations for structurally isolated models as the arbitration of a bus or the decoding of instructions. The simulation is currently performed by an interpreting simulator implemented in C++. Beside the architectural description, it receives the program and the configuration data as binary code. Its generation is part of the compilation framework, which is described in the next section. The event-driven simulator core can be plugged into a visual user interface. Its capabilities go beyond the mere visualization of the architectural state and even allow for the cycle-accurate debugging of an architecture.

5 Compilation Framework

Based on the SUIF compiler toolkit [10], we have developed a versatile compiler framework [14], that has been customized for the ARRIVE architecture. The framework, called Reconfiguration Enabled Compiler And Simulation Toolset (RECAST) consists of a profiler, retargetable compiler, a mapping module and a simulator (Fig. 3).

Frontend: For the processing of the C source, we use the frontend that comes with the SUIF compiler kit. After standard analysis and some architecture-independent transformation and optimization stages, the algorithm is now represented by the SUIF *Intermediate Representation* (IR) in terms of an abstract syntax tree.

Candidate Identifier: To find the most time-consuming parts of the application, which should be accelerated by execution within the reconfigurable array,

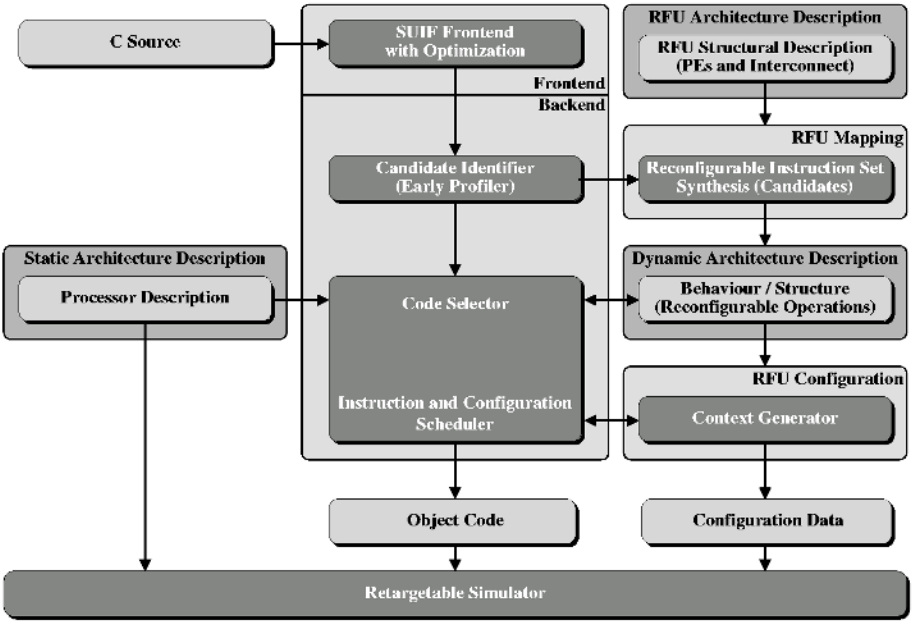

C Source
SUIF Frontend with Optimization
Frontend
Backend
Candidate Identifier (Early Profiler)
Code Selector
Instruction and Configuration Scheduler
RFU Architecture Description
RFU Structural Description (PEs and Interconnect)
RFU Mapping
Reconfigurable Instruction Set Synthesis (Candidates)
Dynamic Architecture Description
Behaviour / Structure (Reconfigurable Operations)
RFU Configuration
Context Generator
Static Architecture Description
Processor Description
Object Code
Configuration Data
Retargetable Simulator

Fig. 3. Compilation framework

a profiler stage is included to estimate the run-time behavior of the application. In contrast to other profiler-driven concepts, early profiling is performed on the intermediate representation instead of requiring fully-compiled object code.

Synthesis of Reconfigurable Instructions: The partitioning of the algorithm takes the profiling data into consideration. In the present implementation, the mapping module generates a description for subtrees of the IR. The mapping results can be influenced by a parameter set. This includes the maximal pipeline depth, the minimal clock frequency and the maximal area consumption. For synthesis, a set of predefined, parameterizable modules is used. These modules were previously evaluated in terms of implementation costs for our reconfigurable units.

Code Selection: Based on the estimated run-time behavior as provided by the early profiler, the candidates for reconfigurable execution can be evaluated and selected so that high speedups are achieved and the number of reconfiguration contexts is minimized. The design parameters annotated to the reconfigurable instruction candidates are used to ensure that resource constraints and design requirements are met.

Scheduling Phase: Interlocked with the selection phase, it attaches the selected configuration contexts to the hardwired ARM instructions and creates the configuration manager event table. The main goal is to find a common schedule that is optimal in terms of utilization of all units and processing parallelism, avoiding

Table 1. FFT schedule mapped to the ARRIVE architecture

Addr	ARM	CFMU	CTXT	RALU	CTXT	VLSU	
0	MOV						
4	MOV						
8	MOV	INIT L0					
12	MOV						
16	MOV						
20	MOV	INIT L1					
24	ADD				1:	LDR twiddle++	
28	MOV				2:	LDR (e1)	LDR (e2)
32	NOP		1:	MUL MUL SUB ADD			
36	NOP	INIT L2	2:	MUL MUL ADD SUB			
40	NOP		3:	ADD SUB ADD SUB	3:	LDR (e1+d)	LDR (e2+d)
44	NOP		1:	MUL MUL SUB ADD	4:	STR (e1)	STR (e1)+d!
48	NOP	LOOP L2	2:	MUL MUL ADD SUB	5:	STR (e2)	STR (e2)+d!
52	NOP		3:	ADD SUB ADD SUB			
56	NOP				4:	STR (e1)	STR (e1)+d!
60	ADD	LOOP L1			5:	STR (e2)	STR (e2)+d!
64	MOV	LOOP L0					

resource conflicts (especially memory and register access related) as much as possible. Software pipelining is also applied to maximize ILP [6].

Through its separated control-flow, data-processing and transfer-oriented reconfigurable functional units, the ARRIVE concept can be considered a load/store architecture, which is inherently compiler-friendly. Thus, the configuration contexts can be obtained by the application of sophisticated code generation methods similar to RISC and VLIW microprocessors [13].

6 Experimental Results

The ARRIVE model has been verified against a 512-point FFT, which is a typical DSP benchmark application. It is implemented as a radix-2 butterfly-based triple-nested loop with 16-bit accuracy. The optimized inner loop contains 21 ARM instructions, executing in 34 clock cycles. Through the utilization of the additional coarse-grain reconfigurable units, the FFT inner-loop execution time can be reduced to 3 clock cycles as shown in table 1. The 16-bit MUL/ADD and MUL/SUB operations are mapped to the reconfigurable array, executing in a single clock cycle and consuming 40 processing elements. Detailed operation layout information is given in figure 4. The init/loop operation addresses of the hardware-accelerated loops are stored into the CFMU start/end address configuration registers, thus allowing the observation and manipulation of the instruction pointer. Every time the control-flow reaches a loop address, the dedicated index register is updated according to the configured increment/decrement operation. Execution of the unaccelerated FFT takes 99608 cycles, whereas it

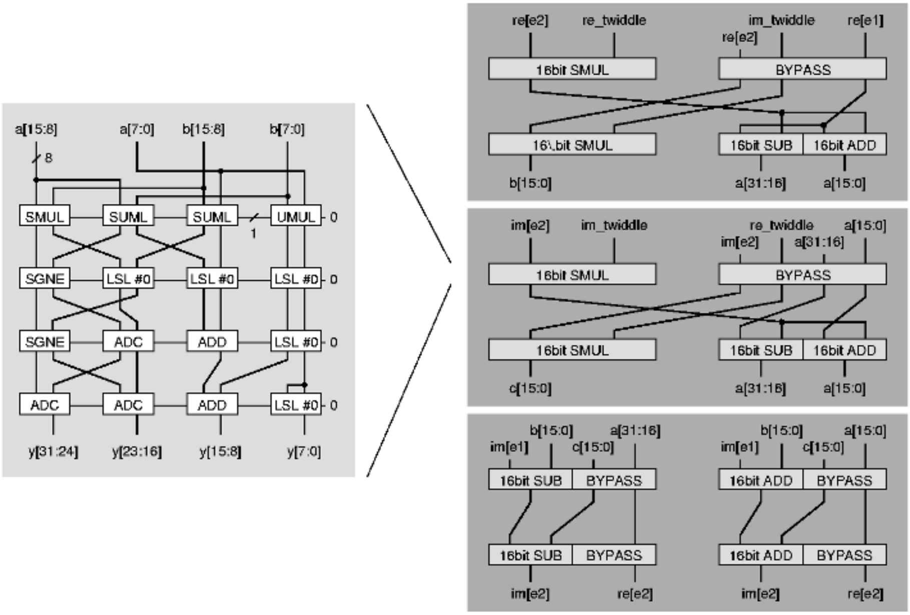


Fig. 4. FFT inner loop RALU operations

can be performed in 8995 cycles on the ARRIVE architecture, assuming all memories are modelled as single clock cycle accessible SRAM.

7 Conclusions

In this paper, we have presented the ARRIVE architecture as an extension of a standard RISC microprocessor architecture. Computing performance has been obtained through tightly-coupled context-controlled reconfigurable functional units, which are synchronized to the microprocessor core in a pseudo-VLIW manner. Spatial computation is performed inside a reconfigurable array, thus increasing ILP through coarse-gain parallelism. As a result, higher functional density can be obtained compared to state-of-the-art VLIW processors [16]. Additionally, the implementation expenses for the reconfigurable units can be kept low by achieving chip area savings, resulting from the reduction of the number of hard-wired units, the decrease of the instruction memory size and the simplification of the instruction decoder. The ARRIVE architecture is also designed taking low-power considerations into account. Beside the idle mode of the processing elements, the avoidance of a large high-bandwidth VLIW decoder is the main argument for a power efficient architecture. Ongoing research will investigate further applications out of the DSP/telecommunication/multimedia domain to be mapped to the ARRIVE architecture. Furthermore, we plan the development of a synthesizable VHDL- model for a prototype evaluation.

References

1. Miyamori, T., Olukotun, K.: REMARC: Reconfigurable Multimedia Array Coprocessor. In: Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA'98), ACM Press (1998)
2. V. Baumgarte, F. May, A. Nückel, M. Vorbach, M. Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. In: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2001). (2001)
3. H. Singh, M.H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, F.E.C. Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. IEEE Transactions on Computers 49 (2000)
4. L. Pozzi, Methodologies for the Design of Application-Specific Reconfigurable VLIW Processors. Ph.D. Thesis, Politechnico di Milano, 2000
5. C. Iseli, SPYDER - A Reconfigurable Processor Development System. Ph.D. Thesis, Ecole Polytechnique Federale, Lausanne, 1996
6. F. Barat, M. Jayapala, P. Op de Beeck, G. Deconinck. Software Pipelining for Coarse-Grained Reconfigurable Instruction Set Processors. ASP-DAC/VLSI Design, 2002
7. J. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In: J. Arnold and K. Pocek, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp. 24–33, Napa, CA, April 1997.
8. S. Sawitzki, A. Gratz, R.G. Spallek. CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism. Proceedings of the 5th Australien Conference on Parallel and Real-Time Systems (PART'98), pp. 213–224, 1998
9. Z.A. Ye, A. Moshovos, S. Hauck, N. Shenoy, P. Banerjee. CHIMAERA: Integrating a Reconfigurable Functional Unit into a High-Performance, Dynamically-Scheduled Superscalar Processor. IEEE Transactions on VLSI Systems, 2002
10. R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.A.M. Anderson, S.W.K. Tjiang, S.W. Liao, C.W. Tseng, M.W. Hall, M.S. Lam, J.L. Hennessy: SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. SIGPLAN Notices 29 (1994) 31–37
11. A. Hoffmann, H. Meyr, R. Leupers. Architecture Exploration for Embedded Processors with LISA. Kluwer Academic Publishers, 2002.
12. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In: Proceedings of the European Conference on Design, Automation and Test (DATE 99). (1999) 485–490
13. S. Köhler, J. Braunes, S. Sawitzki, R.G. Spallek. Improving Code Efficiency for Reconfigurable VLIW Processors. In: 16th International Parallel and Distributed Processing Symposium, 9th Workshop on Reconfigurable Computing (RAW'02), Ft.Lauderdale, 2002.
14. J. Braunes, S. Köhler, R.G. Spallek. RECAST: An Evaluation Framework for Coarse-Grain Reconfigurable Architectures. In: 17th International Conference on Architecture of Computing System, Augsburg, 2004
15. S. Furber. ARM System-on-Chip Architecture. Addison-Wesley, 2000.
16. Philips Semiconductors: TriMedia TM-1000. Programmable Media Processor, 1998

Using of FPGA Coprocessor for Improving the Execution Speed of the Pattern Recognition Algorithm for ATLAS – High Energy Physics Experiment

Christian Hinkelbein, Andrei Khomich, Andreas Kugel, Reinhard Männer, and Matthias Müller

Institute of Computer Science V, University of Mannheim,
B6, 23-29, 68131, Mannheim, Germany
{hinkelbein, khomich, kugel, maenner, mmueller}@ti.uni-mannheim.de
http://www-li5.ti.uni-mannheim.de/fpga

Abstract. Pattern recognition algorithms are used in experimental High Energy physics for getting parameters (features) of particles tracks in detectors. It is particularly important to have fast algorithms in trigger system. This paper investigates the suitability of using FPGA coprocessor for speedup of the TRT-LUT algorithm – one of the feature extraction algorithms for second level trigger for ATLAS experiment (CERN). Two realization of the same algorithm have been compared: C++ realization tested on a computer equipped with dual Xeon 2.4 GHz CPU, 64-bit, 66 MHz PCI bus, 1024 Mb DDR RAM main memories with Red Hat Linux 7.1 and hybrid C++ – VHDL realisation tested on same PC equipped in addition by MPRACE board (FPGA-Coprocessor board based on Xilinx Virtex-II FPGA and made as 64-bit, 66 MHz PCI card developed at the University of Mannheim). Usage of the FPGA coprocessor can give some reasonable speedup in contrast to general purpose processor only for those algorithms (or parts of algorithms), for which there is a possibility to fulfil calculations with a major degree of parallelism. In case of TRT-LUT algorithm it is the most time consuming parts and using of FPGA coprocessor can give us speed-up by factor more then two for hybrid FPGA/CPU realisation in comparison with CPU only implementation.

1 Introduction

ATLAS [1] is one of the general-purpose experiments at Large Hadron Collider (LHC) which will start operation in 2007. It will detect the end-products of proton-proton collisions at a centre-of-mass energy of around 14 TeV.

Triggering is one of the greatest challenges at hadron collider experiments. At the LHC beams will be colliding every 25 ns and the time available for second level trigger algorithms will be about 10 ms. To make things worse, the *pp* interaction leading to the interesting physics process (referred to as the physics

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 791–800, 2004.

event) will be accompanied by several minimum bias interactions occurring simultaneously (~ 20 at the LHC design luminosity). The task of a trigger system is to select rare events and to suppress background events as effiefficiently as possible. One of the most demanding tasks is usually the track reconstruction from measured points (detector hits) of particle trajectories. The ATLAS trigger strategy foresees a reduction of the event rate at several levels: Level-1 (LVL1) and High Level trigger (HLT) which in turn is subdivided on Level-2 (LVL2) and Event Filter (EF) [2].

There are different kinds of algorithms for the different sub-detectors and for different subtasks of the trigger. Results of the feature extraction algorithms is the relevant track features (ϕ, η, p_T, E_T – starting angle, momentum and energy of particle). This document describes the possible acceleration of one of LVL2 feature extraction algorithms for Transition Radiation Tracker (TRT-LUT algorithm) with FPGA coprocessors help.

The FPGA coprocessor – MPRACE (Multi Purpose Reconfigurable Accelerator / Computing Engine) [3] developed at the University of Mannheim was used for this task.

MPRACE is an FPGA-Coprocessor based on Xilinx Virtex-II FPGA and made as 64-bit, 66 MHz PCI card. The main features of this board are:

- Dedicated PCI-to-Local bus bridge PLX9656 with support for 64-bit, 66 MHz PCI operation and 32-bit, 66 MHz local bus operation.
- High capacity FPGA, Xilinx Virtex-II. In first version XC2V3000-4BF957C is used (14 336 slices (28 672 LUTs/FFs), 12 DCMs, 96 Multipliers, 1 728 Mbit of internal block RAM (96 RAM blocks×18 kBit), 684 I/Os)
- High bandwidth memory system: 4 banks ZBT, each 512 k×36 bits (8 Mbytes in total).

The following conventions are used in the paper: the coordinate system has its origin at the interaction point. The z-axis is parallel to the beam direction and x and y or the radius r and the azimuthal angle ϕ are used to denote positions in the transverse plane. Instead of the polar angle θ the pseudorapidity $\eta = -\ln(\tan(\theta/2))$ is used to specify directions inside the detector.

2 Detector Geometry

Transition radiation is produced when a relativistic particle traverses an inhomogeneous medium, in particular the boundary between materials of different electrical properties. This radiation hence offers the possibility of particle identification at highly relativistic energies.

The combined straw tracker and transition radiation detector, the Transition Radiation Tracker (TRT), provides tracking and contributes to the electron identification. These are straw tubes which are used in region where the track density is relatively low. Its purpose is two-fold – firstly to make a large number of measurements of charged particle position, and secondly to assist in the identification of these particles.

The geometrical layout of the TRT used for this work is described below. More details can be found in ATLAS Inner Detector TDR [4].

The TRT consists of a central barrel part and two end-cap sections. The sensitive elements are straws of internal diameter 4 mm with a single sense wire running down the centre. The barrel comprises 52 544 straws parallel to the beam axis in 73 layers; the end-caps comprise 319 488 radial straws in 18 multiplane wheels. The barrel has an inner radius of 56 cm and outer radius of 107 cm. The straws have an electrical break at $z = 0$. In order to reduce the occupancy and improve the measurement, each half of the barrel is read out separately.

The TRT straws provide a two-dimensional position measurement, r-ϕ in the barrel and z-ϕ in the end-caps. Typically the TRT provides 36 measurements on every track in the entire covered η range ($\sim$ 20 000 hits per event).

In the mechanical layout for the detector, the barrel consists of three concentric cylindrical rings. Each ring is formed from 32 independent identical mechanical modules. The modules are not projective and have a different shape from ring to ring [5]. Layers of straws are grouped into three types of modules. In each layer, straws are at approximately constant distance (about 6.8 mm), the $\Delta\phi$ covered by each module is the same ($2\pi/32$), and the number of straws per layer in each module varies from layer to layer (from 15 to 29 straws per layer). The total number of straws is 1 642 per 3 modules, or 52 544 $\times$ 2 for the two halves of the detector.

In the simulation of the detector the geometry is different for the barrel. In the simulation the straws in each layer are arranged in a concentric cylinder. Within a layer all straws have equal distance in ϕ. For all layers the straw distance is approximately $r\phi = 6.8$ mm. From layer to layer the first straws have small ϕ offsets. The distance between two layers is 6.8 mm, there are 75 layers, two of which are empty [4]. This geometry, used for simulation and reconstruction, does not contain any symmetry.

3 Algorithm Description

TRT-LUT algorithm [6] consists of a track candidate search followed by track-fits performed to select the best candidate and to determine the track parameters. The purpose of the candidate search is to reduce the number of point combinations to be investigated at the track fitting stage.

Since all particle trajectories to search for can be calculated in advance a histogramming method based on the Hough Transform is well suited for the initial track search in TRT [6]. The Hough Transform is a standard method in image analysis that allows recognition of global patterns in an image space by recognition of local patterns in a transformed parameter space and the pattern matching problem can be addressed with easily performed peak detection [7]. Algorithm is based on the idea that every hit in the three-dimensional detector image can belong to a number of possible (predefined) tracks characterized by different parameters. All such tracks (or roads) are stored in Look-Up Table (LUT). Thus every hit increases the “probability” for the existence of these tracks by one (histogramming). The histogram for a single track consists of a "bow-tie" shaped region of bins with entries with a peak at the centre of the region. The bin at the peak of the histogram will, in the ideal case, contain all the hits from the track. The roads corresponding to the other filled bins share straws with the peak bin, and so contain sub-sets of the hits from the track. The histogram for a more complex event consists of a superposition of the entries from the individual tracks. The bins containing the complete set of points from each track can be identified as local maxima in the histogram. After a clean-up step followed by a fit the final tracks are selected.

LUT based Hough Transform algorithm for TRT was implemented in C++. It was integrated and investigated in ATLAS Level-2 reference software [6]. The algorithm was implemented in VHDL for MPRACE board as well. This implementation was integrated in to ATLAS Level-2 reference software for performance study and comparison with C++ implementation.

The entire algorithm as included in the ATLAS Level-2 reference software consists of the following steps (Fig. 1):

Initial Track Finding: Utilizes an LUT-based Hough Transform using histogramming to find potential track candidates. In the barrel, the Hough Transform is performed from (R, ϕ) space to $(\phi, 1/p_\mathrm{T})$ space. The LUT consist of 81 920 $(\phi \times 1/p_\mathrm{T} = 1024 \times 80)$ predefined roads. All predefined roads point to the origin and are computed as exact circles in the x-y projection to be accurate enough for low-p_T tracks in magnetic field. The road width increases linearly from 4.5 mm at layer one (numbering from the innermost layer outwards) to 6.8 mm at layer 42 and is then constant and equal to 6.8 mm from layer 42 to layer 73. With this definition, ∼65 straws are assigned to each road. The predefined roads overlap by 30% - 50% in $1/p_\mathrm{T}$ and ϕ. This overlapping of the roads prevents the loss of hits from a track with a trajectory which could otherwise pass between two predefined roads, but can lead to multiply reconstructed tracks, which have to be eliminated in subsequent steps. Each straw is assigned to ∼120 roads (max.130).

Thresholding and Local Maximum Finding: Selects potential track candidates and eliminates multiple reconstructed tracks.

Track Splitting: Removes hits incorrectly assigned to a track, and splits tracks that have been erroneously merged. In this step the pattern of hits associated to a track candidate is analysed. If potential track candidate contains N_is consecutive layers without a hit, the track is split into two separate candidates either side of the gap. If one of the candidates contains more than N_thr hits, it is retained. If both candidates pass this threshold, the track segment which starts at the lowest radius is retained. The result of the track splitting step is a candidate that consists of a sub-set of the straws in a road. For this step "bin-ordered" LUT is constructed (each bin correspond to road). The list of straws lying within the road is stored in the LUT.

Track Fitting and Final Selection: Performs a fit in the r-ϕ(barrel) or z-ϕ(end-caps) plane using a third order polynomial. The algorithm uses only the straw position (i.e. the drift time information is not used). The final track candidates are selected during this step.

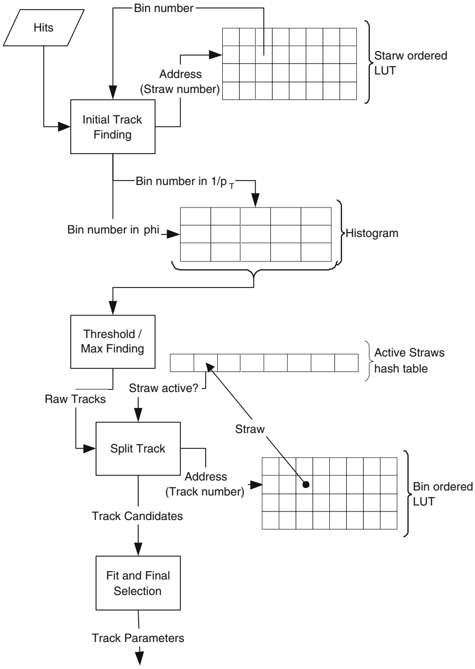


Fig. 1. TRT-LUT algorithm

Profiling of a C++ implementation of the TRT-LUT algorithm shows that most of the computing time is spent with access of LUTs, incrementing of 8-bit numbers, and a local maximum finding. CPU-only implementation of Histogramming (or Initial Track Finding) and Thresholding / Local Maximum Finding require 62% of the total processing time. These two steps are good candidates for an FPGA implementation because of both steps can exploit an 80-fold parallelism (80 $1/p_{\mathrm{T}}$ blocks) and make use of the fast internal dual port block RAM. The main difference between an FPGA implementation and a CPU implementation of the Hough transformation is that the loop over all predefined roads for one straw (one row in LUT), which is executed once per active straw and increments ∼120 histogram counters, is executed sequentially in the CPU and in parallel in the FPGA.

A schematic view of the VHDL implementation of the initial track finding step of the algorithm is shown in Fig. 2. This implementation takes advantage of both the external SRAM and the internal RAM blocks.

The initial track finding works as follows. The array of active straws (hits) prepared in host memory and transferred over PCI bus to MPRACE by DMA.

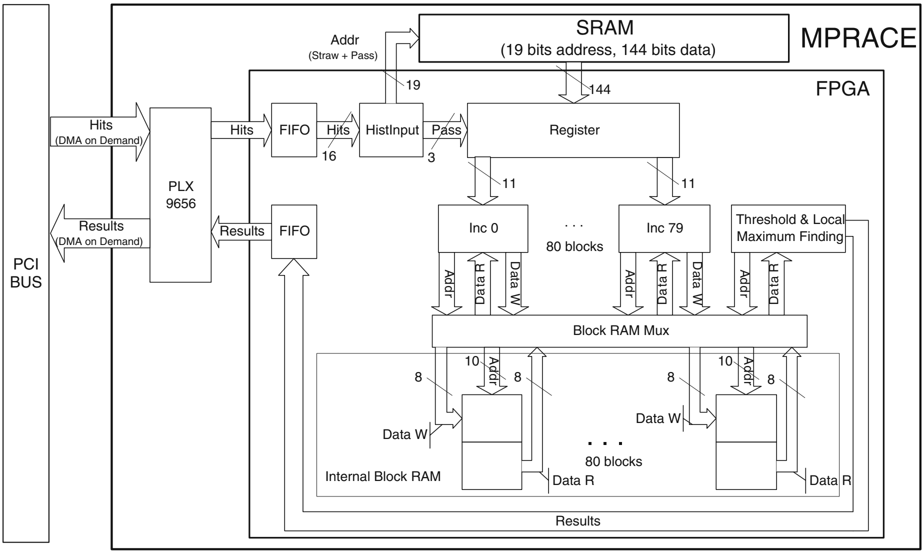


Fig. 2. FPGA initial track finding

For each hit straw, counters are incremented for all roads containing that straw. The histogram counters for all TRT straws which have to be incremented are stored in the "straw-ordered" LUT. The construction of the LUT with overlapping roads in ϕ and $1/p_{\mathrm{T}}$ guarantees that each active straw contributes to exactly one or two pre-defined ϕ values of a certain predefined $1/p_{\mathrm{T}}$. For the barrel, there are 80 "$1/p_{\mathrm{T}}$-blocks" with 1024 possible ϕ each, thus 81 920 patterns are predefined. Therefore the one or two values out of 1024 possible values, which have to be incremented, are stored in 11 bits (10+1), because the (possibly) two values are always directly neighboured.

The "straw-ordered" LUT is stored in the SRAM. An address space of 16 bits for the straws in the "straw-ordered" LUT is required (if symmetry of the detector not in use). The SRAM itself has 19 bits addresses, of which 16 are used to identify the current straw. The LUT stores for each straw address 880 bits - histogram counters addresses (80×11 bits). The word length of the SRAM is only 144 bits; therefore seven steps (passes) with the same straw address (16 bits) and changing pass addresses (3 bits) are needed for transfer from SRAM (LUT) all information corresponding to one straw to FPGA. Using of the detector symmetry can significantly reduce requirements for size and word length of the memory used for LUT. The histogram is stored in the internal RAM blocks in the FPGA. The implementation profits from a true Dual Port RAM (internal RAM Blocks) to allow a fast read-modify-write cycle during one external SRAM cycle. Therefore histogramming for each straw is executed in seven passes with up to three clock cycles each, whereas a CPU implementation requires looping sequentially over 130 histogram bins.

After the histogramming step the maximum finding step first applies a threshold filter and then search for local maximum. The maximum finding has to be applied in two dimensions (ϕ and $1/p_{\mathrm{T}}$). This is done in parallel for 80 $1/p_{\mathrm{T}}$-blocks and sequentially for 1024 ϕ-blocks. Resulting are the histogram counters with values above the threshold and which are local maximum in both ϕ and $1/p_{\mathrm{T}}$. The output information contains the predefined ϕ and $1/p_{\mathrm{T}}$ values of the track (predefined from the LUT) and the number of active straws corresponding to the track. These results stored in output FIFO and transferred over PCI bus to host memory by DMA.

If the track splitting step is also done in the FPGA, the results of the local maximum finding step are stored internally in the FPGA for this step.

A schematic view of the FPGA implementation of the track splitting step looks similar to initial track finding and is shown in Fig. 3.

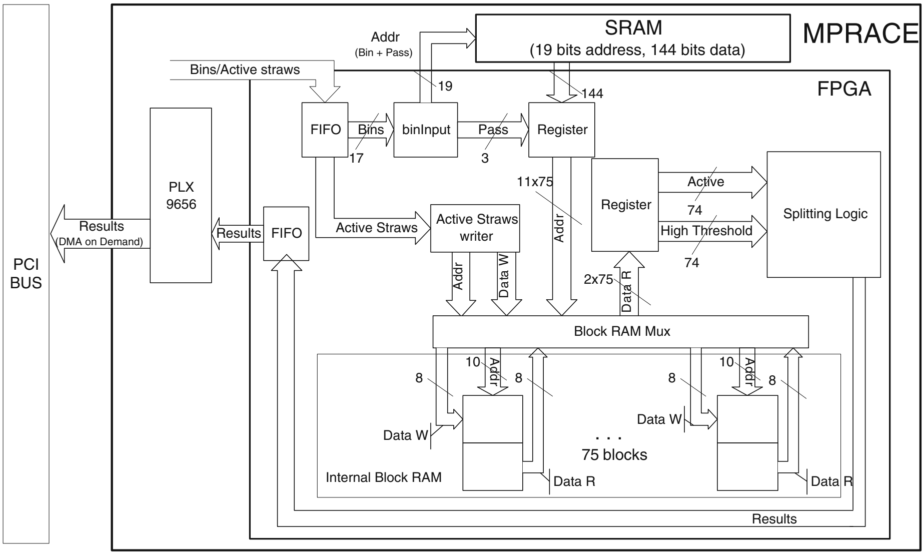


Fig. 3. FPGA initial track finding

The track splitting step uses the 75 internal RAM blocks (each block for one layer/plane) for a hash-table, which stores for each straw in this layer/plane if it is active or not ("active straws" hash-table), and the threshold information (and optionally the drift-time). The SRAM is used to address the bin-ordered LUT with histogram bin numbers above the threshold which were a local maximum. The LUT outputs all straws which belong to the pattern definition layer by layer. The bin-ordered LUT stored 825 bits (75 layers×11 bits for straw number in layer). The word length of the SRAM is 144 bits; therefore six steps (passes) with the same bin address and changing pass addresses are needed for transfer from SRAM (LUT) all information corresponding to one pattern to FPGA.

Straw numbers from LUT are used as address for “active straws” hash-table. We have two 74 bit words from this hash-table: one of them give us information about active straws, second – about high threshold hits. This information is used for perform track splitting and the result is a 28 bit word which contain number of hits, number of high threshold hits, fist hit layer and last hit layer for track candidate. This result is passed to the host CPU over PCI bus by DMA for the track fit.

It is possible to put both LUTs (straw-ordered and bin-ordered) in to SRAM memory of the MPRACE board if information about detector symmetry will be in use. For barrel the 32-fold symmetry allows to store only $32\ \phi \times 80\ 1/p_{\mathrm{T}} = 2\,560$ bins. Therefore we need 20 bit address at the SRAM: 16 bits for “straw-ordered” LUT (52 544 straws), 3 bits for passes and 1 bit for selecting one of LUTs (“bin-ordered” or “straw-ordered”). Without symmetry we need 21 bit address: 17 bits for “bin-ordered” LUT, 3 bits for pass and 1 bit for LUT selecting.

4 Execution Time Measurement Results

Measurements have been made on a computer equipped with dual Xeon 2.4 GHz CPU, 64-bit, 66 MHz PCI bus, 1024 Mb DDR RAM main memories with Red Hat Linux 7.1 and MPRACE board. But only one CPU has been used by algorithm. It is supposed that the algorithm will be included in ATLAS Trigger software which has several algorithms for different subdetectors and any of these algorithms can be run in parallel on one workstation. There is no advantage to have multithreaded algorithms in this case.

Only initial track finding step of the TRT-LUT algorithm was implemented in VHDL. Software was compiled with GNU gcc-2.95; Mentor Graphics LeonardoSpectrum and Xilinx ISE4 were used for synthesis and place and route. FPGA design is synchronised by 64 MHz clock signal. A data file containing approx. 156 events was used.

Identical results are obtained for the initial track finding step for both the CPU and FPGA implementations.

The TRT consists of two subdetector types, the barrel and the end-cap. Both measure essentially in two dimensions, the barrel in $r - \phi$ and the end-cap in $z - \phi$. The algorithms for barrel and end-cap differ, but the principles are identical. For the performance measurements presented in this paper only the barrel algorithm was used. The TRT barrel is composed of two identical (in the scope of this document) parts, the left and right barrel half. The following numbers refer to one half barrel.

The total number of straws is 52 500. The resolution required for the Initial Track Finding is defined by 1024 bins in ϕ space and 80 bins in $1/p_{\mathrm{T}}$ space leading to total search space 81 920 patterns. Information about barrel symmetry is not used. Measurements results are shown in Table 1.

For higher speed-up “Track Splitting” part of algorithm should be implemented in FPGA too. We expect that FPGA realisation of “Track Splitting” gives at least the same speed-up as “Histogramming / Maximum Finding”. In

Table 1. Execution times on a Xeon 2.4 GHz PC for the TRT-LUT CPU only and hybrid implementation. (The total time is not the sum of the average times shown in the table, but rather is the average of the total times per event.)

Number of events: 156
Average number of hits in event: 2 167.413
CPU: Pentium-IV XEON 2.4 GHz
FPGA board: MPRACE (VIRTEX-II), 64 MHz, 64-bit, 66 MHz PCI
Pattern-ordered LUT with 80 $1/p_{\mathrm{T}}$ *and 1024* ϕ *values predefined*
Threshold: $N_{\mathrm{thr}} = 12$*; Isolation:* $N_{\mathrm{is}} = 9$
Average number of track candidates per event: 57

Task	Platform	Time(μs)
CPU-only Implementation		
Fill event	CPU	107.77
Histogramming	CPU	2314.03
Threshold/Max Find	CPU	376.76
Histo/Thresh/Max	**CPU**	**2694.72**
Track Splitting	CPU	1246.18
Track Fitting	CPU	151.91
Final Selection	CPU	25.05
Copy tracks	CPU	145.62
Extract time	**CPU**	**4371.24**
Hybrid Implementation		
Fill event	CPU	115.23
Data converting	CPU	24.62
prepare write buffer	CPU	81.55
DMA write/Histogramming	PCI/FPGA	433.04
prepare read buffer	CPU	23.44
Max Finding/DMA read	FPGA/PCI	22.42
output formatting	CPU	17.71
Histo/Thresh/Max	**FPGA/CPU**	**612.77**
Track Splitting	CPU	1274.87
Track Fitting	CPU	155.54
Final Selection	CPU	24.80
Copy tracks	CPU	149.71
Extract time	**FPGA/CPU**	**2332.92**

this case we will have “Track Splitting” done in ~285 μs instead of 1 246 μs and full extract time will be ~1 350 μs. It is gives us speed-up ~3.2.

5 Conclusion

Improving the speed of tracking algorithms is extremely important for future hadron collider experiments, where the presence of many pile-up interactions simultaneously recorded with the interesting physics event leads to very high hit occupancy in tracking detectors. Faster algorithm allows physicists not only to obtain data sooner, but it allows to have lower Level-1 and Level-2 thresholds with same events rate, and, therefore, to study more interesting physics (like b-physics).

One of the possible approaches is use of hybrid FPGA/CPU systems for such algorithms where most time consuming parts of algorithm are accelerated by FPGA platforms. We could demonstrate that FPGAs are well suitable for these tasks. Usage of the FPGA coprocessor can give some reasonable speedup as contrasted to general purpose processor only for those algorithms (or parts of algorithms), for which one there is a possibility to fulfil calculations with a major degree of parallelism. In case of TRT-LUT algorithm it is the most time consuming parts and using of FPGA coprocessor can give us speed-up by factor ∼2–3 for hybrid FPGA/CPU realisation in comparison with CPU only implementation. It allows to make search for low-p_{T} tracks in entire TRT detector at low luminosity ($10^{33}\,\mathrm{cm}^{-2}\mathrm{s}^{-1}$) in reasonable time what is particularly important for some special physics programs which can be done on ATLAS (for example b-physics).

References

1. ATLAS Collaboration: ATLAS technical proposal. CERN/LHCC 94-13, CERN, Geneva (1994)
2. ATLAS Collaboration: ATLAS detector and physics performance TDR. CERN/LHCC 99-14, CERN, Geneva (1999)
3. Kugel, A.: MPRACE, preliminary documentation. CERN intranet, CERN (2002) http://akugel.home.cern.ch/akugel/mpRace/.
4. ATLAS Inner Detector Community: ATLAS Inner Detector Technical Design Report. CERN/LHCC 97-16, CERN, Geneva (1997)
5. Clarke, P., Falciano, S., Le Du, P., Lane, J., Abolins, M., Schwick, C., Wickens, F.: Detector and readout specifications, and buffer-RoI relations, for the level-2 studies. ATL-DAQ-99-014, CERN (1999)
6. Hinkelbein, C., Kugel, A., Männer, R., Müller, M., Sessler, M., Singpiel, H., Baines, J., Bock, R., Smizanska, M.: Pattern recognition in the TRT for the ATLAS B-Physics trigger. ATL-DAQ-99-012, CERN (1999)
7. Illingworth, J., Kittler, J.: A survey of the Hough transform. Comput. Vision Graphics, Image Processing **44** (1988) 87–116

Partial and Dynamically Reconfiguration of Xilinx Virtex-II FPGAs

B. Blodget, C. Bobda, M. Huebner, and A. Niyonkuru

brandon.blodget@xilinx.com, bobda@cs.fau.de,
huebner@itiv.uni-karslruhe.de, niyonkur@hsu-hh.de

Abstract. Current trends show that partial and dynamic reconfiguration can be used in adaptive systems. These systems are able to adapt themselves to the demand of their environment during run-time. This flexibility can be used for many industrial applications. Unfortunately the current documentation for using this methodology isn't sufficient. This gap should be closed with a tutorial that shows the possibilities of this feature and also gives some precious hints for designers.

1 Introduction

Field programmable gate-arrays (FPGAs) are mainly used today for rapid-prototyping purposes. They can be reconfigured many times for different applications. Modern state-of-the-art FPGA devices like Xilinx Virtex FPGAs [1] additionally support a partial dynamic run-time reconfiguration which reveals new aspects for the designer who wants to develop future applications demanding adaptive and flexible hardware. Especially in the domain of mobile computing high-end mobile communication applications will benefit from the capabilities of the new generation of reconfigurable devices. There exist some new approaches deploying Virtex/Virtex-II FPGAs in multimedia applications using their capabilities for a dynamic function-multiplex showing new ways for the efficient deployment of partial run-time reconfiguration [2] [4]. A new approach to create systems that are able to manage configuration is run-time systems. These systems use the flexibility of an FPGA by changing the configuration partially [3]. Only the necessary functions are configured in the chip's memory. A function can be replaced by another function while other parts stay operative. To solve the problem of substitution and I/O management the configuration needs a main module to control the tasks. With such a system it is possible to save resources like output pins and energy because of outsourcing configuration data [4]. The need of chip area becomes smaller and therefore the power consumption can be reduced. Nevertheless the power requirements of such applications will grow with increasing rate of configuration [5]. Creating such a system has two aspects: Reducing amount of chip area and reducing power consumption by designing a control system which manages the content of FPGAs configuration in an intelligent way to minimize reconfiguration rate. Additionally this management can control the on chip intercommunication bus to prevent an overhead of bus size. The development of such

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 801–810, 2004.

a system assumes a deep knowledge about internal dependencies inside the FPGA. A tutorial should open this possibility for interested researchers and developers.

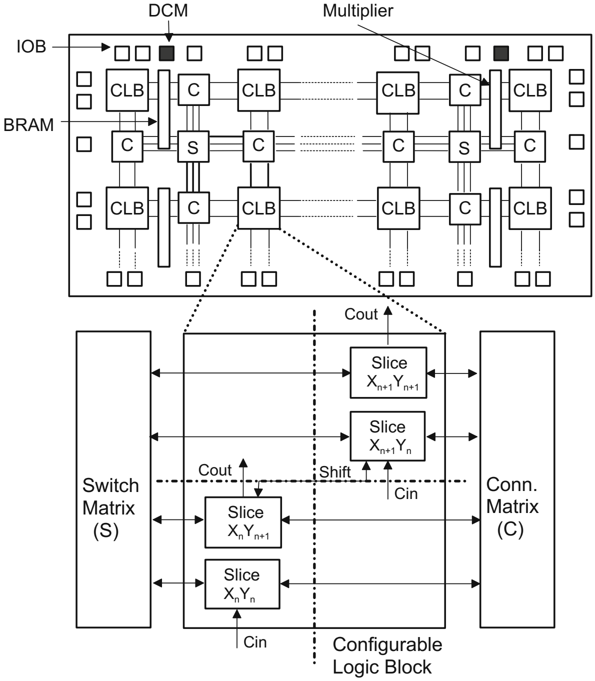


Fig. 1. Architecture of Xilinx Virtex FPGAs

2 Internal Structure of Xilinx Virtex FPGAs

The key-component of each Xilinx Virtex FPGA is the CLB, which consists of four slices each of them providing two 4-input function generators, carry logic, arithmetic logic gates, function multiplexers and two storage elements (figure 1) [10][12]. The four slices are directly linked to the global switching matrix and to the fast connection matrix, so that routing is done between slices and not at the CLB-level. Thus, the view of a Virtex FPGA using e.g. Xilinx Floorplanner shows the slices available. When implementing partial reconfiguration, it is important to place reconfigurable modules horizontally on a four-slice boundary. This means that the leftmost slice number ('X'-index) for any reconfigurable module must be 0, 4, 8… Vertically, the reconfigurable module fits into the full height of the device [11]. Section 4 (Placement Constraints) provides more details how to do it.

3 Modular Design Flow

The Modular Design Flow is the method generally used to enable partial and dynamic reconfiguration. Originally, it is intended to allow many designers to work on the same design project in parallel. Xilinx recommends it to implement partial reconfiguration [11].

One specific requirement for partial reconfiguration is there must be as many top-level design projects as the number of reconfigurable modules planned; each top-level includes a different alternate reconfigurable module. The top-level designs include fixed modules and reconfigurable modules all instantiated as "black boxes". The corresponding attribute for VHDL-design entry is: "*attribute box_type of <module name>: component is "BLACK_BOX*" which is defined in the Xilinx *unisim* library. Communication between fixed and reconfigurable modules is implemented by instantiating as many bus macros as required to assure signal integrity during partial reconfiguration. For that, the bus macro component and its implementation file (*bm_4b_v2.nmc*) are provided. Each top-level design can be handled as a single design project and therefore synthesized in a separate directory.

In order to get a clear but also essential organization of the design files, Xilinx recommends setting up a well-defined directory structure. An example of such a structure can be found in Xilinx Application Notes 290 [11]. The also provided reference design (e.g. calctop for VHDL-based designs) shows all steps and the corresponding commands needed to fully implement partial reconfiguration using Modular Design Flow.

4 Placement Constraints

Placement constraints are one of the central points in the partial reconfiguration. They are used exclusively to allocate device area for modules in a given configuration or bitstream. The placement constraints can be done only on the basis of the module's size which can be estimated after compiling the top-level design containing the module as described in the initial budgeting phase. The Xilinx PACE tool can then be used to graphically visualize the module. This is helpful since it provides a good visual estimation on the bounding-box in which the module can be placed. Use the PACE toolkit to constraint the module to be placed in the estimated bounding-box to a given location.

PACE generates (or modifies, if there is one) an ".ucf" file in which the placement constraints are described in the correct notation (e.g. ***AREA_GROUP = usergroup RANGE=SLICE_X3Y1: SLICE_X33Y33*** for the Virtex II and II-Pro and ***AREA_GROUP = usergroup RANGE=CLB_R3C1: CLB_R33C33*** for the Virtex, Virtex E and Spartan).

It is very important to note that all the pins used by a given module should be locked in the area (set of columns) occupied by that module. Otherwise, bus macros should be used to keep the integrity of the signals across module boundaries, especially when driving a signal from one module to a configurable one. Most designers spend 80% of the time on solving such issues which are really not related to the partial reconfiguration itself. Unfortunately, most of the boards on the market were not designed for partial reconfiguration purpose. We recommend studying the pin assignment on a board and only buying it, if it matches the computation-flow of your application.

5 JBits

JBits [13] is an Application Program Interface (API) to the Xilinx configuration bitstream. It is designed for dynamic modification of Virtex-II bitstreams. JBits contains Java classes which allow having access to internal resources by modifying parts of a partial or complete configuration bitstream. Modification of bitstreams generated by Xilinx design tools or bitstreams read back from a device is possible.

Designers are able to access resources (e.g. CLBs, IOBs, Block RAM and PIPs) and may modify them before writing the bitstream back to the FPGA. One example use of JBits is to use it to extract partial configuration data from a complete bitstream. This can be done after the Modular Design Flow (see section 3) to generate partial configuration data. This data can be stored in an external memory and used for partial reconfiguration by reading out external Flash memory and sending the data to the Internal Configuration Access Port (ICAP) or the SelectMAP™ interface (see section 6). In [4] this technique is used within the reconfigurable system using static design flow (figure 2). In this example JBits is only used to extract data from an existing bitstream without modification of the data.

JBits can modify bitstreams very quickly, which enables it to be used in runtime reconfiguration (RTR) applications. Using the Java Run Time Reconfiguration (RTR) design flow (figure 2), the configuration of the FPGA can be modified during runtime. An application uses JBits for modifying existing or read back configuration data. Modifications are done and exported to XHWIF (Xilinx HardWare InterFace) for communication to the hardware. The XHWIF interface can be used for communication with a FPGA board. It provides several methods to describe the FPGA based board and to send data to and from the board. The powerful XHWIF API adds a layer of abstraction to the hardware. This enables the simple porting of applications to new hardware. XHWIF also provides clock stepping and readback commands which enable debugging on hardware.

6 ICAP Interface

ICAP is an acronym for Internal Configuration Access Port. This component was introduced in the Xilinx Virtex-II devices. It is also present in Xilinx Virtex-II Pro devices. The ICAP provides configuration access to the FPGA logic. This in essence enables self-reconfiguration of Virtex-II devices. Care obviously must be taken when using the ICAP not to reconfigure the circuitry that is controlling the ICAP. Thus the ICAP does not allow full reconfigurations, only partial reconfigurations. The ICAP interface is a subset of the SelectMAP™ Interface. The process for configuring the device and reading back from the device using the ICAP is essentially the same as it is for SelectMAP. Xilinx Application Notes 138 and 151 describe the Virtex™ series configuration architecture and the SelectMAP interface [9][10]. Table 1 describes the ICAP ports.

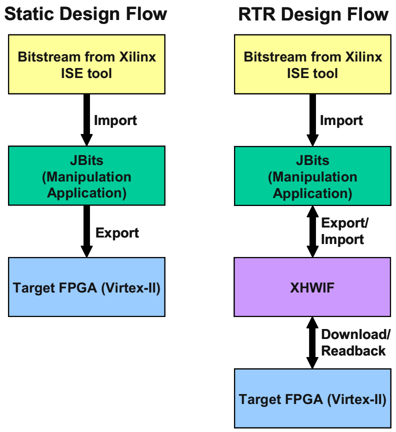


Fig. 2. Design Flow with Jbits

The ICAP interface is simplified because it only has to support partial reconfigurations. It does not have to support full configurations or multiple modes like SelectMAP. The pins that are missing from the ICAP interface are the mode pins (M2, M1, M0), DONE, INIT, and PROGRAM. The SelectMAP CS pin has been renamed CE on the ICAP. This pin still performs the same functions and it defaults to active low just like in the SelectMAP interface. The SelectMAP bi-directional D[0:7] port is split into two ports in the ICAP interface. These two ports are I[0:7] and O[0:7]. The Xilinx Embedded Developer Toolkit (EDK) provides tools for implementing FPGA designs containing embedded processors. The EDK version 6.2 contains a peripheral called *opb_hwicap*. This peripheral wraps the ICAP with additional logic that can read and write frames to a BRAM. The *opb_hwicap* interfaces with the CoreConnect™ On-Chip Peripheral Bus (OPB)[7]. This peripheral with its associated drivers abstracts away the details of the ICAP configuration interface and enables a self-reconfiguring platform [8].

7 Troubleshooting

When compiling a partial reconfigurable design, they are some troubles which may appear. We have listed some of them below and provide some tips on how to solve them.

In the initial phase
Error: At least on inactive module...
Solution: at top level, each module should be defined as black box
In the Synthesis phase when defining bus macros
Error: a default value is assigned, a signal should be assigned...
Solution: In port map of bus macro, the signal must be grouped.

Table 1.

ICAP Port	**Description**
I[0:7]	The I0 through I7 pins function as an input data bus to the ICAP port. Configuration data is written to this bus. The I0 pin is considered the MSB bit of each byte.
O[0:7]	The O0 through O7 pins function as an output data bus from the ICAP port. Readback data is read from this bus. The O0 pin is considered the MSB bit of each byte. This bus also provides status information while the device is being configured.
BUSY	The BUSY output indicates when the FPGA can accept another byte. If BUSY is Low, the FPGA reads the data bus on the next rising CCLK edge where both CE and WRITE are asserted Low. If BUSY is High, the current byte is ignored and must be reloaded on the next rising CCLK edge when BUSY is Low.
CE	Enables the ICAP port. Although not specified in the Libraries guide, the CE pin is active low by default. FPGA Editor shows that it is possible to invert this signal to make it active high. This pin is equivalent to the SelectMAP \CS pin.
WRITE	Like the CE pin the WRITE pin is active low by default. When asserted Low, the WRITE signal indicates that data is being written to Data Input bus (I[0:7]). When asserted High, the WRITE signal indicates that data is being read from Data Output bus (O[0:7]). FPGA Editor shows that it is possible to invert this signal to make it active high. This pin is equivalent to the SelectMAP \WRITE pin.
CCLK	The CCLK signal synchronizes all loading and reading of the data bus for configuration and readback.

Error: no pin connected for the output of bus macro
Solution: Define dummy output in the top level entity, and feed the output of bus macro to them. Update the .ucf file accordingly.

In the active phase

Error: multiple drivers...
Solution: Check the project properties of the top level designs and all modules before launching the synthesize tool: the bus delimiter must be set to (), and the I/O buffers must be disabled when synthesizing the modules.

Assembling phase

Error: The STEPPING level for this design is 0.
FATAL_ERROR: Guide: basgitaskphyspr.c:255:1.28.20.1.14.1:137…
Solution: This error is cause by a GLOBAL_LOGIC used implicitly in a sub-module, but locked in the boundary of the device. For example if a BUFG is used in a sub-module this will implicitly declare and use a global logic GLOBAL_LOGIC1_XX. Since GLOBAL_LOGIC1_XX has to be locked in top level design, a fatal error will be given back.

Never declare and use a GCC or GND in the top-level module. All the constant signals (e.g. GND=0; VCC=1) needed at the top-level should be defined in a sub-module and fed to the top-level entity. Remove all the GNDs and VCCs signals declared in the sub-level modules which are not used.

8 Partial Reconfiguration with HandelC on the RC200 Board

One of our experiments on partial reconfiguration was made using the HandelC language on the RC200 board. The structure of the directory previously described is the same. Using HandelC a design can be compiled either to VHDL or to EDIF, which then can be used through the Modular Design Flow. The example below shows how to implement a small partial reconfigurable design using HandelC.

Example: The design consists of an adder in its first configuration and it is partially reconfigured to act as a subtracter. Both modules (adder and subtracter) have to be implemented separately. The following listing shows a HandelC description of an adder.

```
void main()
{
        unsigned 32 res;
        interface port_in(unsigned 32 var_1 with {busformat="B<I>"}) Invar_a();
        interface port_in(unsigned 32 var_2 with {busformat="B<I>"}) Invar_b();
        interface port_out() Outvar(unsigned 32 Res = res with {busformat="B<I>"});
          res = Invar_a.var_1 + Invar_b.var_2;
}
```

The first part of the code is the definition of the interfaces for communication with other modules and the second part realizes the addition with the input values coming from the input interface and the output values sent to the output interface. The modules must now be included into the top-level module and connected together as shown on the following code segment.

```
unsigned 32 operand1, operand2;
unsigned 32 result;
interface adder (unsigned 32 Res)
my_adder(unsigned 32 var_1 = operand1, unsigned 32 var_2 = operand2) with
{busformat="B<I>"};

void main()
{
   operand1 = produceoperand( 0); operand2 = produceoperand( 3);
   result = my_adder.Res;
   dosethingwithresult(result);
}
```

According to the top-level design being implemented, the adder will be replaced by a subtracter. The next question is how to keep the signal integrity of the interfaces. Since there is no bus macro available in HandelC, bus-macros provided by Xilinx may be used as VHDL-component in a HandelC design. Before setting constraints on a HandelC design, it has to be compiled first. Afterwards, the resulting

EDIF-file is opened with any text-editor and the longest pattern that contains the name of the module as declared in the top-level module is selected. This is useful because the HandelC compiler adds some characters to the module name used in the original design. If the original module name is used it will not be recognized in the following steps of the Modular Design Flow.

Experience with the RC200-Board
Since all the interfaces and pin-locking of the RC200-board are hard-coded in the Celoxica-PSL-library, it is not possible to set the required pin-constraints after the design has been compiled to EDIF. Thus, the design has to be compiled to VHDL in order to overcome this issue. In the resulting entities the top-level ports are named with the pin-names of the given device. Note that the pins are not optimally (in term of partial reconfigurability) placed on the RC200-board.

9 Practical Demonstration

We have implemented a self-reconfigurable platform under Linux. The hardware for this demo is the Xilinx ProMedia Board which contains 3 Virtex2Pro XC2VP50 devices. In this demo only one XC2VP50 is used. On the XC2VP50 is a complete running Linux system. The embedded PowerPC is used to run Linux and multiple peripherals are implemented in the FPGA fabric. The ProMedia board has a LM4549A audio chip. We use this chip to stream music from an MP3 player into an audio filter implemented on the same FPGA as the running Linux system. From a laptop we are able to login to the ProMedia board over a TCP/IP connection. Once logged in to the board we can download new DSP filters over the TCP/IP connection and self-reconfigure the FPGA with these filters. The effect of loading the new filters can be heard through the speakers.

Figure 3 shows how the physical design has been partitioned. In this design bus macros are not used since there is no communication between the logic that makes up the embedded Linux system and the filters that are reconfigured. However there is communication between the main Linux system on the right hand side and the Ethernet and SystemAce peripherals on the left hand side via an OPB bus. The DSP filter between these two regions can be reconfigured without disturbing the communication on the OPB bus. This is possible because Virtex 2 [Pro] devices offer glitchless partial reconfiguration. If a configuration bit holds the same value before and after configuration there will be no glitch on the resource that bit controls. Resources to be careful of are SRL16s and LUT Rams because they change dynamically and will be over written when configuration occurs. We worked around this problem by constraining the filters to the top part of the reconfigurable region and allowing only OPB routing (no logic) under the filter region.

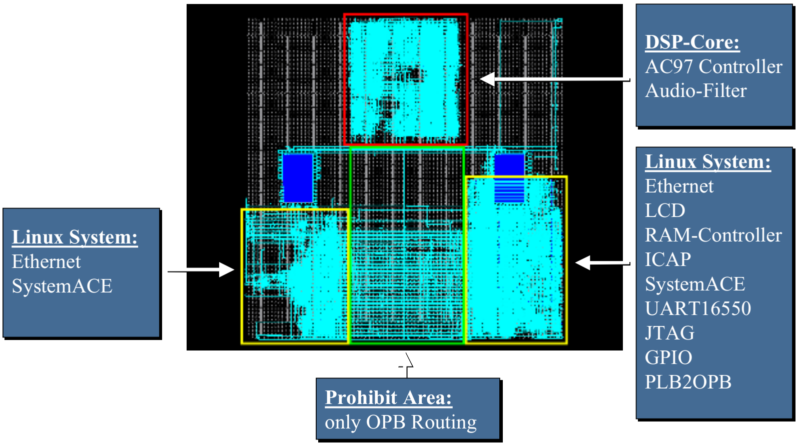


Fig. 3. FPGA Editor view of implemented system

10 Conclusions

Partial and dynamic run-time reconfiguration offers new possibilities for designs with Xilinx Virtex-II FPGAs. This paper shows the desired design flow and novel tools enabling the use of this technique. One example of the benefit for using dynamic reconfiguration is the possibility to use smaller FPGAs by outsourcing configuration data [5]. Other aspects are the adaptivity of systems to the demand of e.g. applications or environment. This technique opens a great field for investigation and the development of new systems.

References

1. www.xilinx.com
2. Y. Ha, B. Mei, P. Schaumont, S.Vernalde, R. Lauwereins, H. De Man; "Development of a Design Framework for Platform- Independent Networked Reconfiguration of Software and Hardware"; Proc. 11th Int´l Conference on Field Programmable Logic and Applications, Belfast, Ireland, 2001M.
3. J.-Y. Mignolet, S. Vernalde, D. Verkest, R. Lauwereins: “Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances”; Int´l. Conf. on Engineering of Reconfigurable Systems and Algorithms; June 25-27 2002, Las Vegas, USA
4. M. Ullmann, B. Grimm, M. Huebner, J. Becker: “An FPGA Run-Time System for Dynamical On-Demand Reconfiguration”, RAW04, Apr. 04, Santa Fé, USA
5. J. Becker, M. Hübner, M. Ullmann: “Real-Time Dynamically Run-Time Reconfiguration for Power-/Cost-optimized Virtex FPGA Realizations”, VLSI03, Darmstadt, Sep. 03
6. J. Becker, M. Hübner, M. Ullmann: “Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations”, SBCCI03, Sao Paulo, Sep. 03
7. IBM web site. http://www.chips.ibm.com/products/coreconnect (2003)

8. B. Blodget, P. James-Roxby, E. Keller, S. McMillan, P. Sundararajan. “A self-reconfiguring platform”, In Proc. of the Intern. Conference on Field Programmable Logic and Applications (FPL2003), Lisbon, Portugal, Sept. 2003.
9. Carmichael, C.: “Virtex FPGA series configuration and readback”. Xilinx Application Note XAPP138, version 1.1, Xilinx, Inc. (1999)
10. “Virtex Series Configuration Architecture User Guide”. Xilinx Application Note XAPP151, version 1.6, Xilinx, Inc. (2003)
11. “Two Flows for Partial Reconfiguration: Module Based or Difference Based”, Xilinx Application Note XAPP290, version 1.1, Xilinx, Inc. (2003)
12. “Virtex-II Platform FPGA User Guide”, version 1.8, Xilinx, Inc. (2004)
13. http://www.xilinx.com/products/software/jbits/

SystemC for the Design and Modeling of Programmable Systems

Adam Donlin[1], Axel Braun[2], and Adam Rose[3]

[1] Xilinx Research Labs, 2100 Logic Drive, San Jose, CA 95124,
[2] Computer Engineering Dept., University of Tuebingen,
Sand 13, 72076 Tuebingen, Germany.
[3] Cadence Design Systems, Manchester, England.

Abstract. The Field Programmable Logic (FPL) community is set to assume an important role within the electronic system level (ESL) community. Programmable technologies are proving to be the correct implementation substrate for the growing majority of system architects who can no longer afford the cost or shoulder the risks associated with submicron ASIC design. In this tutorial we present an overview of SystemC, the dominant and open environment for ESL design and modeling. We focus on presenting the fundamentals of the language and describing an important extension to the language that enables rapid modeling of systems at the transaction level.

1 Introduction

ASIC design has become an overwhelming task, plagued by spiraling cost and complexity, and burdened with issues at both the silicon and system levels. Programmable logic such as FPGAs are a compelling alternative implementation technology that allow the designer to mitigate the silicon-level issues of ASIC design. Yet, as FPGAs and other programmable technology become host to larger systems, the challenges being tackled by the electronic system level (ESL) for contemporary ASIC design community also will be applicable to designers in the FPGA community. This paper describes SystemC, the defacto industry standard modeling environment for ESL. Our aim is to provide a comprehensive overview of two of the main features of the SystemC environment. In section one, we describe the fundamentals of the SystemC language and its approach to modeling. In section two, we discuss an important extension to SystemC: a standard library, defined by Cadence, for modeling systems at the *transaction* level. This library is proposed to the SystemC community for widespread adoption and is in process of being donated to the Open SystemC Initiative (OSCI) for inclusion into the SystemC language.

2 The Fundamentals of SystemC

Many system-level models have been written using the C or C++ language. SystemC standardizes these approaches in one language, based in C++ and

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 811–820, 2004.

implemented as a library that includes a simulation kernel. The library provides all the necessary constructs for hardware specification, software design, and system-level modeling. The governing body of SystemC is the Open SystemC Initiative (OSCI). OSCI holds the license of SystemC under an open software license model and it drives both the standardization and the further development of the language. In addition, a vivid market of commercial products (EDA tools, IP products, etc.) has been established around the SystemC language.

Compared to traditional hardware description languages like VHDL or (System)Verilog, SystemC addresses a broad range of abstraction levels, beginning from low-level hardware modeling on the register-transfer level (version 1.x) up to system-level modeling methodologies and algorithmic specification styles (current version 2.x). The future roadmap for the language integrates real-time software development (version 3.x), and analog, mixed-signal design (version 4.x).

2.1 Basic Modeling Concepts

The structure of the SystemC library is shown in Figure 1. The core language elements (modules, ports, processes, etc.) and the extended hardware-specific data types (including standard C++ types) are the basis blocks of the language. Elementary channels for communication and synchronization (signals, FIFOs, etc.) are built upon these. SystemC allows designers to extend the language's modeling capabilities by adding user-defined or application-specific channels or libraries, e.g. for master/slave-like communication, verification and testbenching, or alternative models of computation.

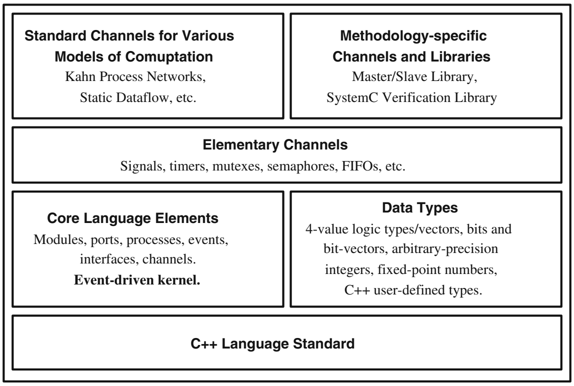

Standard Channels for Various Models of Comuptation
Kahn Process Networks, Static Dataflow, etc.
Methodology-specific Channels and Libraries
Master/Slave Library, SystemC Verification Library
Elementary Channels
Signals, timers, mutexes, semaphores, FIFOs, etc.
Core Language Elements
Modules, ports, processes, events, interfaces, channels.
Event-driven kernel.
Data Types
4-value logic types/vectors, bits and bit-vectors, arbitrary-precision integers, fixed-point numbers, C++ user-defined types.
C++ Language Standard

Fig. 1. SystemC Language and Library Structure

Modules and Hierarchy. Modules are the building blocks of a SystemC design. The `sc_module` class allows us to compose a design from distinct functional units and separate the specification of computation from communication.

Modules may contain other modules, allowing hierarchical composition. A module may also contain SystemC processes which enable concurrency within the model. The behavior of the module is specified in its member functions. Data members of a module can be any kind of variables or SystemC ports. Ports are the connecting points to the interfaces of SystemC channels and facilitate communication between the modules of the model.

Channels, Ports, and Interfaces. To ensure extensibility and a proper separation of functionality and communication, SystemC supports the concept of channels. Channels (derived from `SC_CHANNEL`) are containers for communication. The interfaces (derived from `sc_interface`) of a channel define only the function signatures (prototypes) for the communication and synchronization which is to be implemented within a channel. A process in a module may access these communication functions (e.g. data `read()` or `write()` functions) by 'binding' the channel interfaces to ports of the module and invoking the functions specified in the interface via the bound port object. Figure 2.1 shows this scenario. The interface concept enables easy substitution of one channel implementation for another (e.g. swoping an un-timed for a timed implementation) *without* any modification of the module specifications, provided the interface definition remains unchanged.

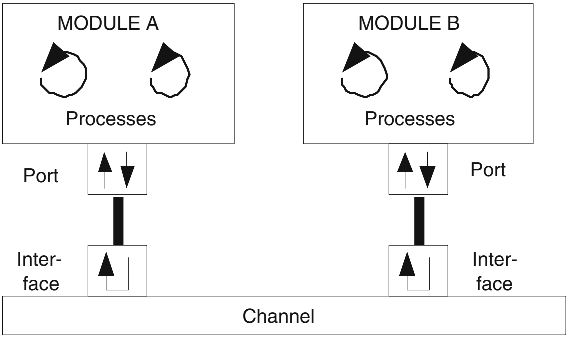


Fig. 2. Module Communication

Besides these 'primitive' channels, SystemC also supports 'hierarchical' channels that can contain their own processes, e.g. for modeling complex communication protocols. SystemC provides several standard channels, such as hardware signals (`sc_signal`), and FIFOs (`sc_fifo`), etc. Primitive and hierarchical channels can also be user-defined. With these features, SystemC can model basic low-level communication like hardware signals as well as extended, abstract communication semantics defined by the designer.

Processes: As mentioned above, the functionality in SystemC designs is expressed in member functions of the respective modules. This functionality is

vitalized by processes, the SystemC constructs that represent concurrency. SystemC provides two types of process: `sc_method` and `sc_thread`:
Methods: When activated, a method process always executes its function body from the beginning to the end. It cannot be interrupted and therefore it doesn't save any internal state. For example the modeling of a finite state machine using methods explicitly requires member variables and switch statements.
Threads: In contrast to a method, a thread process may be suspended by calling `wait()` functions. Whilst suspended, its state (including local variables) is retained and the execution will be continued from the same point after it is reinvoked by the SystemC scheduler. This allows the specification of multi-cycle behavior very easily.
Sensitivity lists (both static and dynamic) are used to inform the SystemC scheduler which specific events (e.g. signal changes, clock signals, user defined, etc.) may activate any given process.
Data Types: As SystemC is based on the C++ programming language, all C++ data types like integer, floating-point data types, etc. can be used within a SystemC model. The SystemC library also introduces a huge variety of additional data types that are designed specially for system and hardware modeling: two-valued and four-valued bits and bit vector types, arbitrary width integer data types and a sophisticated set fixed-point data types.

Putting it all together: SystemC Simulation. OSCI maintains the reference implementation of the SystemC simulator. In order to run a SystemC simulation, the entire model is compiled using a common C++ compiler and linked to the SystemC library (the OSCI event-driven simulation kernel is included in the SystemC library). The resulting executable represents the simulator for the given model and running it performs an actual simulation of the system. During execution, the model may interact with its environment, read and produce data files and, with the help of an integrated trace library, produce standard VCD waveforms. Third party vendors such as Cadence, Mentor and CoWare also offer simulators for SystemC.

The path towards Hardware: SystemC Synthesis. SystemC includes design flows for hardware synthesis. The synthesizable subset of the SystemC language depends strongly on the synthesis tool. The current synthesizable subset is comparable to that of classical hardware description languages. Ongoing standardization activities within the OSCI synthesis working group also cover the specification of a tool and vendor independent synthesizable subset.

3 Transaction Level Modeling

Transaction Level Modeling (TLM) addresses a number of practical ESL design problems. These include: providing an early platform for software development; design exploration; system verification; and the need to use system-level models in block-level verification. RTL tools and techniques are intended for block level implementation and test and are not effective for ESL design. As such, an ESL tool chain must enable design at levels of abstraction above RTL: designs should be captured and manipulated at the *transaction* level. The OSCI Transaction Level Working Group (TLMWG) has defined the following abstraction levels, listed in increasing order of detail:

- **algorithmic level**(AL): purely behavioral, no architectural detail whatsoever.
- **communicating processes**(CP): behavior partitioned into a network of processes connected point-to-point, possibly through buffers.
- **communicating processes with time**(CP+T): CP annotated with high-level performance data.
- **programmers view** (PV): behavior specified memory and register accurate. The CP point-to-point interconnect refined to an abstracted bus or NoC model. The model is sequenced but untimed.
- **programmers view with time** (PV+T): PV with multi-cycle timing estimates annotated.
- **cycle callable** (CC): system behavior modeled with hardware and cycle-true detail. Communication models are protocol-true.
- **register transfer level** (RTL).

Of these abstraction levels, TLM applies at the levels between AL and RTL. SystemC has supported TLM since Version 2.0 but the lack of established standards and methodologies meant each TLM effort had to invent its own methods and APIs. In addition to ‘reinventing the wheel’, the methodologies each differed slightly, making IP exchange difficult. An industry standard for TLM will increase the productivity of software engineers, architects, implementation and verification engineers. However, the improvement in productivity promised by such a standard can only be achieved if the standard meets a number of criteria: it must be easy to use, efficient and thread-safe; it must enable reuse between projects and abstraction levels; it must model hardware, software and hybrid designs easily; and it must enable the design of generic components such as routers and arbiters. This section describes the proposed OSCI TLM standard, as originally defined by Cadence in [5]. Widespread adoption of this proposal will enable the productivity improvements promised by TLM.

3.1 The TLM Proposal

Three key concepts are used in the proposal: interfaces; blocking vs non-blocking methods; and bi-directional vs uni-directional transactions.

Interfaces : The emphasis on interfaces rather than implementation flows from the fact that SystemC is a C++ class library, and that C++ (when used properly) is an object oriented language. Interfaces are the key to enabling interoperability and reuse. Before we can define a TLM standard, we first need to rigorously define the key interfaces, and then we can go on to discuss the various ways these may be implemented in a TLM design. In SystemC, all interfaces should inherit from the class `sc_interface`.

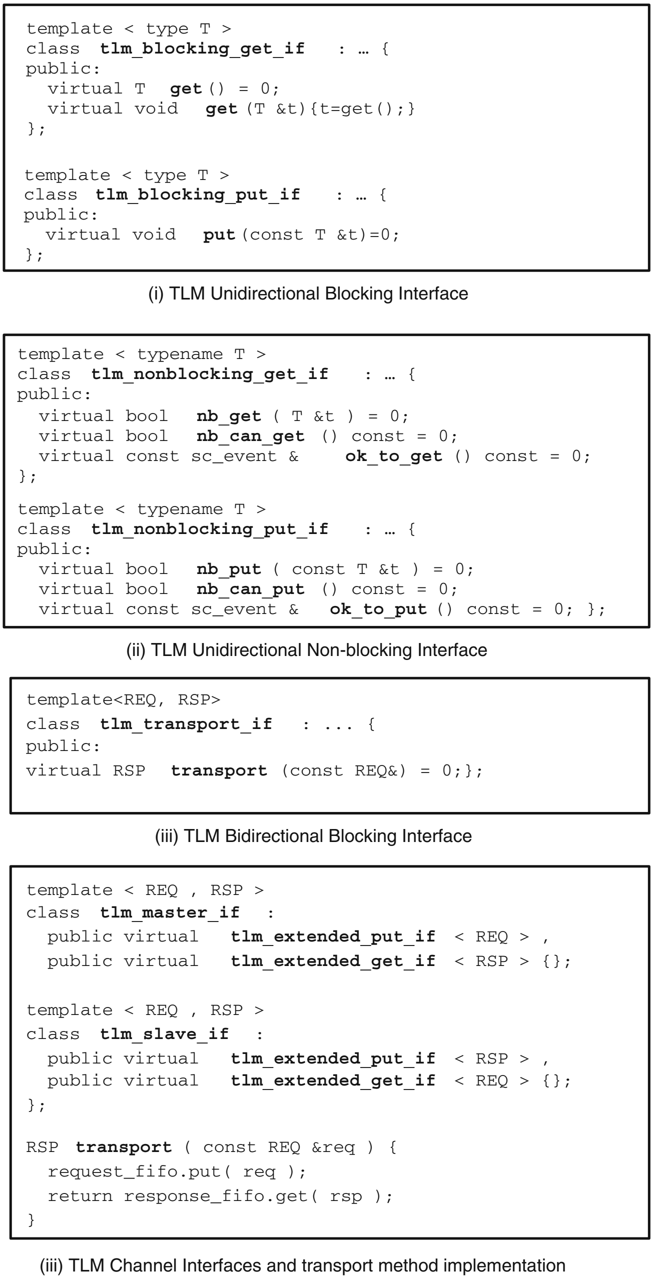
Blocking and Non Blocking: As discussed in the fundamentals section, SystemC offers two concurrency constructs: `SC_THREAD` and `SC_METHOD`. OSCI defines `SC_METHOD` to be a*non-blocking* construct: it cannot invoke the simulator's '`wait()`' function. `SC_THREAD` is a blocking process construct as it may call `wait()`. This distinction is important to TLM as some models are more naturally captured with a blocking style whereas others have a natural fit with a non-blocking style. The non-blocking style avoids simulator context switch penalties and therefore gains simulation performance. This issue is discussed in some depth in [4]. In the TLM API, we clearly delimit which methods are blocking (they may call '`wait()`') and which methods are non-blocking (they are guaranteed not to call '`wait()`').

Bidirectional vs Unidirectional: Some transactions (e.g., a read across a bus) are clearly bidirectional whilst others are clearly unidirectional (e.g., a packet based communication mechanism). Complex protocols may be deconstructed into a sequence of bidirectional or unidirectional transfers. For example, a bus with address, control and data phases may look like a simple bidirectional read/write bus at a high level of abstraction, but more like a sequence of pipelined unidirectional transfers at a more detailed level. Any TLM standard must have both bidirectional and unidirectional interfaces, provide a common look and feel to both interfaces, and show clearly how the two may inter-relate.

3.2 The Core TLM Interfaces

The Unidirectional Interfaces. Figure 3(i) and (ii) show the unidirectional interfaces, both of which are based on the `sc_fifo` interfaces of SystemC 2.1. `sc_fifo` has been used for many years and, as a result, it is well understood and proven reliable in concurrent systems. Furthermore, interfaces based on `sc_fifo` are amenable to well known static scheduling optimizations. The unidirectional interface classes are split into blocking and non-blocking classes. Non-blocking methods are distinguished with the prefix "`nb_`". For convenience, we supply two forms of `get` and a default implementation of the pass-by-reference form. Therefore, an implementer of the interface need only supply one.

The non-blocking interfaces may 'fail' since they are not allowed to `wait()` for the correct conditions for the calls to succeed. Since the blocking functions are allowed to call '`wait()`', they never 'fail'. Hence `nb_put` and `nb_get` must return a bool to indicate whether the invocation succeeded. We also supply `nb_can_put` and `nb_can_get` which enquire whether a transfer *will* be successful without actually moving any data. These methods enable polling `put()` and `get()` methods to be implemented. We also supply event functions which enable an `SC_THREAD` to wait until it is likely that the access succeeds or a `SC_METHOD` to be woken up because the event has been notified. These event functions enable an interrupt driven approach to using the non-blocking interface. However, in the general case, even if the relevant event has been notified, we still need to check the return value of the non-blocking method. For example, many threads



```
template < type T >
class  tlm_blocking_get_if   : … {
public:
  virtual T  get() = 0;
  virtual void  get(T &t){t=get();}
};

template < type T >
class  tlm_blocking_put_if   : … {
public:
  virtual void  put(const T &t)=0;
};
```

(i) TLM Unidirectional Blocking Interface

```
template < typename T >
class  tlm_nonblocking_get_if   : … {
public:
  virtual bool  nb_get ( T &t ) = 0;
  virtual bool  nb_can_get  () const = 0;
  virtual const sc_event &    ok_to_get () const = 0;
};

template < typename T >
class  tlm_nonblocking_put_if   : … {
public:
  virtual bool  nb_put ( const T &t ) = 0;
  virtual bool  nb_can_put  () const = 0;
  virtual const sc_event &   ok_to_put () const = 0; };
```

(ii) TLM Unidirectional Non-blocking Interface

```
template<REQ, RSP>
class  tlm_transport_if   : ... {
public:
virtual RSP  transport (const REQ&) = 0;};
```

(iii) TLM Bidirectional Blocking Interface

```
template < REQ , RSP >
class  tlm_master_if   :
  public virtual   tlm_extended_put_if  < REQ > ,
  public virtual   tlm_extended_get_if  < RSP > {};

template < REQ , RSP >
class  tlm_slave_if   :
  public virtual   tlm_extended_put_if  < RSP > ,
  public virtual   tlm_extended_get_if  < REQ > {};
};

RSP  transport ( const REQ &req ) {
  request_fifo.put( req );
  return response_fifo.get( rsp );
}
```

(iii) TLM Channel Interfaces and transport method implementation

Fig. 3. TLM Interfaces and Channels

may have been notified that a fifo is no longer full but only the first to 'wake up' is guaranteed to have room to write before the fifo is full again.

Bidirectional Blocking Interface. The bidirectional blocking interface, shown in Figure 3(iii), is used to model transactions where there is a tight one to one, non-pipelined binding between the request going in and the response coming out. This is typically true when modeling from a software programmers point of view, when for example a read can be described as an address going in and the read data coming back. The signature of the transport function can be seen as a merger between the blocking `get()` and `put()` functions. This is by design, since then we can produce implementations of `tlm_transport_if` which simply call the `put()` and `get()` of two unidirectional interfaces.

3.3 TLM Channels

One or more of the interfaces described above can be implemented by any channel that a user cares to design. However, we have identified two channels common to a large number of modeling contexts. We provide implementations of them as part of the core TLM proposal.
`tlm_fifo<T>`: The `tlm_fifo<T>` class implements all the unidirectional interfaces above, and is based on the implementation of `sc_fifo`. In addition to the `sc_fifo` functionality, `tlm_fifo` can be zero or infinite sized, and implements the fifo interface extensions discussed in [5].
`tlm_req_rsp_channel<REQ,RSP>`: This class contains two fifos, one for the request going from initiator to target and one for the response from target to initiator. We provide direct access to these fifos by exporting the four fifo interfaces. The channel implements three additional interfaces. The first two combine the unidirectional requests and responses into convenient master and slave interfaces. In addition to this, it implements the `transport()` method of the `tlm_transport_if<REQ,RSP>` as shown in Figure 3(iv). This simple function provides a key link between the bidirectional and sequential world (represented by the transport function) and the timed, unidirectional world (represented by `tlm_fifo<T>`).

3.4 Core of TLM Library

The ten methods, grouped into the five interfaces above, will form the basis of the OSCI TLM standard. With it we may build models of software and hardware, generic routers and arbiters, pipelined and non pipelined buses, and packet based protocols. We can also model at multiple levels of timing and data abstraction and provide channels to connect *between* abstraction levels. Users may design their own channels, implementing some or all of the TLM interfaces. In addition to the core interfaces, the two standard channels (`tlm_fifo<T>` and `tlm_req_rsp_channel<REQ,RSP>`) are included and enable modeling of a wide variety of timed systems and an easy to use bridge between the untimed and timed domains.

3.5 The Structure of an OSCI-TLM Model

It is not possible to give a detailed example of a TLM model in this paper but in this section we shall discuss the abstract structure of a single master, single slave TLM. We refer the reader to [4,5] for a detailed treatment of both the philosophy and technology of the TLM library. Figure 4 shows the common use of the TLM interfaces in a bus oriented model with a master node initiating a transaction to a slave node. The structure may be applied through each of the abstraction levels and a discussion of the refinement of a TLM based on this structure can be found in [5], alongside many more examples of using the TLM interface classes for common system structures.

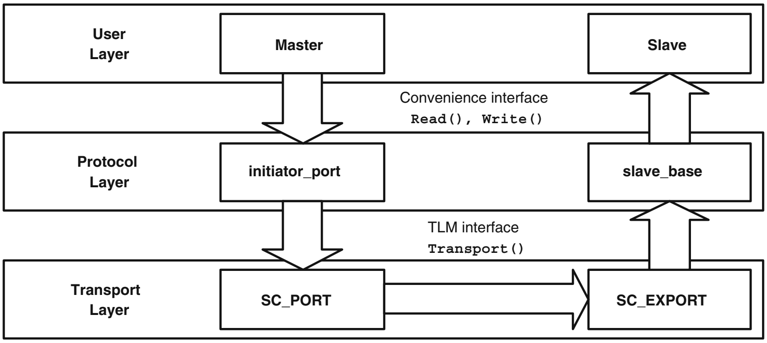


Fig. 4. Common API Layering in a TLM

We have separated out the API utilization according to three layers: user, protocol and transport. This separation exists to serve two distinct groups of engineers. The first group (the “application-level” engineers) understand the application domain very well, but they are not particularly expert in C++ nor are they interested in the finer details of the TLM transport layer or the signal level protocol used to communicate between modules. The second group (the “system-modeling” engineers) do not necessarily understand the application domain well, but its members do understand the underlying communication protocols and the C++ techniques used to model them. Because of the difference in relative expertise and interests, it is often useful (but by no means compulsory) to define a protocol-specific boundary between the two groups. In Figure 4, we identify this boundary as the ‘convenience interface’. It allows the first group of engineers to create comprehensive models of the system behavior without concerning themselves with detailed protocol and modeling issues. The structure shown in Figure 4 assumes a bus-based system and exposes two `read()` and `write()` methods. It is these methods that constitute the convenience interface and, through them, the ‘application-modeling’ engineers may express interaction of the system modules they specify. Below the user layer, the protocol and transport layer may be implemented and optimized by the second group of engineers. At these levels, they may further optimize the model's implementation for efficient exploration and analysis of the system . By maintaining coherence with the respective TLM interfaces, the implementations may also be swopped out for alternative implementations as required.

4 Summary

ESL and the technologies which enable it are of growing relevance to the programmble logic community. As ASIC design cost and complexity continue to grow, more and more system-level ASIC design starts will be displaced into programmable technologies. FPGAs are the dominant programmable logic technology and this tutorial has offered an overview of SystemC, the dominant ESL modeling technology, to this important design community. The authors of this paper firmly believe that, as FPGAs and other programmable technologies assume their role in system-class designs, the design and modeling environment offered by SystemC will become a central component of the FPGA designer's tool flow. We have provided an overview of two key components of ESL with SystemC: the fundamentals of the language and the upcoming transaction-level extensions to the language. For further discussion of these issues we direct the reader to [2,1] and [4,5] respectively. A breadth of information on SystemC and OSCI may also be found at [6].

Acknowledgments. Transaction level modeling has been a keen focus for the members of the OSCI Transaction Level Modeling Working Group(TLMWG). We wish to acknowledge the various members of the TLMWG who have contributed to the philosophy and technology of Section 3.

References

1. Grötker, T., Ciao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers, Boston Dordrecht London (2002)
2. Black D.C., Donovan J., "SystemC: from the Ground Up", Kluwer Academic Publishers (2004).
3. Doulos Ltd., "SystemC Golden Reference Guide.", Doulos Ltd., Ringwood (2002).
4. Burton M. and Donlin A., "Transaction Level Modeling: Above RTL Design and Modeling." Whitepaper of the OSCI Transaction Level Modeling Group (2003).
5. Rose A., Swan S., Pierce J.L., Fernandez J.M., "Transaction Level Modeling in SystemC" Cadence Whitepaper for the OSCI Transaction Level Modeling Group.
6. The Open SystemC Initiative, http://www.systemc.org/
7. OSCI, The SystemC Language Reference Manual,
http://www.systemc.org/, (2003).

An Evolvable Hardware Tutorial

Jim Torresen

Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
jimtoer@ifi.uio.no
http://home.ifi.uio.no/∼jimtoer

Abstract. Evolvable Hardware (EHW) is a scheme - inspired by natural evolution, for automatic design of hardware systems. By exploring a large design search space, EHW may find solutions for a task, unsolvable, or more optimal than those found using traditional design methods. During evolution it is necessary to evaluate a large number of different circuits which is normally most efficiently undertaken in reconfigurable hardware. For digital design, FPGAs (Field Programmable Gate Arrays) are very applicable. Thus, this technology is applied in much of the work with evolvable hardware. The paper introduces EHW and outlines how it can be applied for hardware design of real-world applications. It continues by discussing the main problems and possible solutions. This includes improving the scalability of evolved systems. Promising features of EHW will be addressed as well, including run-time adaptable systems.

1 Introduction

The number of transistors becoming available for designers continue to increase as Moores law seems to be valid for the development of new computer hardware. Earlier we have seen a limit in the *size* of hardware devices. However, we may very well soon see a limit in *designability*. That is, designers are not able to apply all the transistors in the largest integrated circuits becoming available. To overcome this problem, new and more automatic design schemes would have to be invented. One such method is evolvable hardware (EHW).

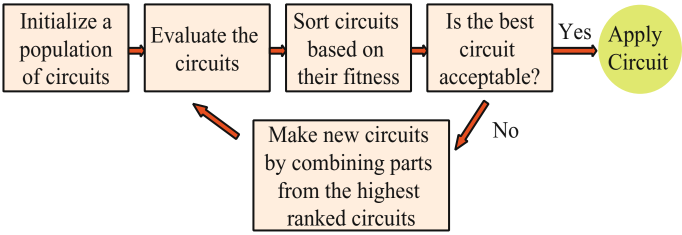

Initialize a population of circuits
Evaluate the circuits
Sort circuits based on their fitness
Is the best circuit acceptable?
Yes
Apply Circuit
No
Make new circuits by combining parts from the highest ranked circuits

Fig. 1. The algorithm for evolving circuits.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 821–830, 2004.

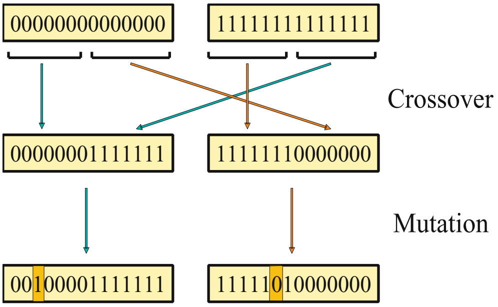


Fig. 2. The genetic algorithm operators.

It was introduced for about ten years ago as a new way of designing electronic circuits [4]. Instead of manually designing a circuit, only input/output-relations are specified. The circuit is automatically designed using an adaptive algorithm inspired from natural evolution. The algorithm is illustrated in Fig. 1. In this algorithm, a set (population) of circuits – i.e. circuit representations, are first randomly generated. The behavior of each circuit is evaluated and the best circuits are combined to generate new and hopefully better circuits. Thus, the design is based on incremental improvement of a population of initially randomly generated circuits. Circuits among the best ones have the highest probability of being combined to generate new and possibly better circuits. The evaluation is according to the behavior initially specified by the user. After a number of iterations, the fittest circuit is to behave according to the initial specification. The most commonly used evolutionary algorithm is genetic algorithm (GA) [3]. The algorithm – which follows the steps described above, contains important operators like crossover and mutation of the circuit representations for making new circuits. The operations are very similar to those found in natural evolution as seen in Fig. 2.

Each individual – representing a circuit description, in the population is often named *chromosome* or genotype and is represented by an array of bits. Each bit in the array is often called a *gene*. Thus, each chromosome contains a representation of a circuit with a set of components and their interconnections. In crossover, the parameters of the pairwise selected circuits are exchanged to generate – for each couple, two new offspring – preferably fitter than the parents. As an alternative to crossover operation, there is usually some probability for conducting cloning instead. Then, the two offspring circuits are equal to the two parent circuits. Further, the *best* circuit may as well be directly copied into the next generation (called elitism). Mutations may also occur and involves randomly inverting a few genes in the chromosome. This make the chromosomes slightly different from what could be obtained by only *combining* parent chromosomes.

When the number of offspring circuits equals the number of circuits in the parent population, the new offspring population is ready to become the new parent population. The original parent population is deleted. Thus, one loop in Fig. 1 is named one *generation*. Randomness is introduced in the selection of parents to be mated. Not only the fittest circuits are selected. However, the probability of a circuit being selected for breeding decreases with decreasing fitness score.

A circuit can be represented in several different ways. For digital circuits however, gate level representation is most commonly used. That is, the representation contains a

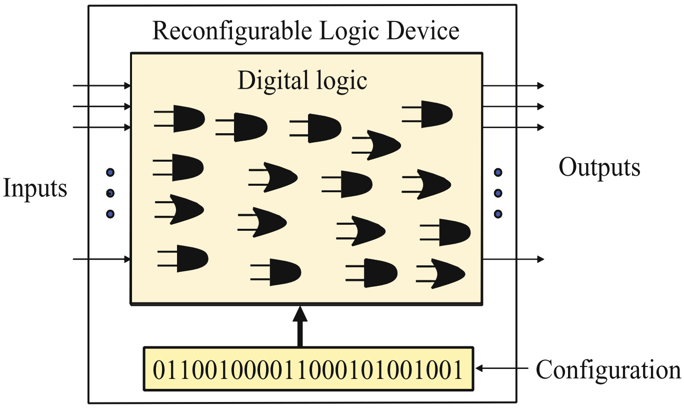


Fig. 3. Illustration of an Field Programmable Logic Device (FPLD).

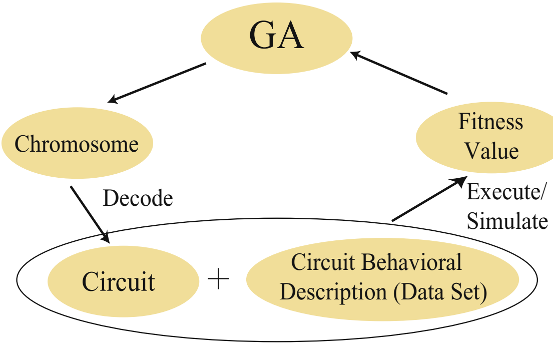


Fig. 4. The cycle of evolving a circuit.

description of what kind of gates are applied and their interconnections. This is coded into a binary configuration bitstream applied to configure a reconfigurable logic device as seen in Fig. 3. This is usually either a commercial device like an Field Programmable Gate Array (FPGA) or a part of an Application Specific Integrated Circuit (ASIC). Each new circuit would have to be evaluated for each generation. This can be undertaken more efficiently by measuring the performance on a real reconfigurable device compared to using simulation. Thus, reconfigurable technology is an important technology for the development of systems based on evolvable hardware.

In addition to the evolutionary algorithm (GA), a circuit *specification* would have to be available. This is often a set of training vectors (input/output mappings) assembled into a *data set*. The operation of GA together with the data set are given in Fig. 4. The most computational demanding part of GA is usually the evaluation of each circuit – typically named fitness computation. This involves inputing data to each circuit and computing the error given by the deviation from the specified correct output.

There are a number of aspects to consider when evolving hardware. Many roads lead to evolved systems. This paper will give an overview of the large variety of schemes, parameter settings and architectures available. However, evolving systems have a number of limitations. One of the main problems with evolution seems to be the scalability of the systems. As the complexity of a system increases, the chromosome string length must be increased. However, a larger number of generations is required as the string length increases. This often makes the search space too large to be able to find a well performing system. Thus, this paper will go into this important problem and present how it is tried to be solved.

On the other hand, there are several new features provided with evolution. Since the systems are automatically designed, we are able to have the evolution going on at runtime in parallel with normal operation. This could be applied to modify the system if the environment changes or errors occur in the system. Work on such run-time adaptable systems will be described. Evolvable hardware has been applied to a number of real-world applications showing the applicability of the promising approach of evolving hardware systems.

The next section contains a classification of EHW research based on a given classification framework [32]. This is followed by a discussion of the main problems in Section 3. Section 4 includes a presentation of the online adaptivity provided with EHW. Conclusions are given in Section 5.

2 A Framework for Classifying EHW

EHW research is rapidly diverging. Thus, to understand the EHW field of research, a classification framework would be beneficial. This is presented below. The many degrees of freedom in EHW could be represented in a multi-dimensional space. However, here a list format is prefered.

Table 1. Characteristics of EHW applied to real-world applications.

Application	EA	TE	AR	BB	THW	FC	EV	SC
Adaptive Equalizer [18]	GA	D	CD	Neuron	Custom	ONL	On-chip	S
Ampl. and Filter Design [15]	GA	A	CD	T/R/L/C	Custom	OFL	Off-chip	S
Analog Circuit Synthesis [12]	GP	A	CD	R/L/C	Custom	OFL	Off-chip	S
Character Recognition [31]	GA	D	CD	Gate	Comm.	OFL	Off-chip	S
Clock Adjustment [27]	GA	D	CT	Gate	Custom	ONL	Off-chip	S
Digital Filter Design [16]	GA	D	CD	Gate	–	OFL	Off-chip	S
Gene Finding [39]	GA	D	CD	Gate	Comm.	OFL	Off-chip	S
IF Filter Tuning [17]	GA	A	CT	Filter	Custom	ONL	Off-chip	S
Image Compression [22]	GA	D	CT	Pixel	Custom	ONL	On-chip	D
Image Compression [23]	GA	D	CD	Gate	Comm	ONL	On-chip	D
Multi-spect. Image Rec. [20]	GA	D	CT	Function	Comm.	OFL	Off-chip	S
Number Recognition [7]	GA	D	CD	Gate	Comm.	OFL	Off-chip	S
Prosthetic Hand [9]	GA	D	CD	Gate	Custom	ONL	Complete	S
Road Image Rec. [33]	GA	D	CD	Gate	Comm.	OFL	Off-chip	S
Robot Control [10]	GA	D	CD	Gate	Comm.	ONL	Complete	D
Robot Control [28]	GA	D	CD	Gate	Comm.	ONL	Off-chip	S
Sonar Classification [38]	GA	D	CD	Gate	Comm.	OFL	Off-chip	S

Evolutionary Algorithm (EA). A set of major algorithms exists:

- **Genetic Algorithm (GA)**
- **Genetic Programming (GP)**
- **Evolutionary Programming (EP)**

The major difference between GA and GP is the chromosome representation. GA organizes the genes in an array, while GP applies a tree of genes. Both schemes apply both crossover and mutation, while EP – which has no contraints on the representation, uses mutation only.

Technology (TE). Technology for the target EHW:

- **Digital**
- **Analog**

Architecture (AR). The architecture applied in evolution:

- **Complete Circuit Design (CD)** Complete circuit evolution where building block functions (see below) and their interconnections are evolved.
- **Circuit Parameter Tuning (CT)** The architecture is designed in advance and only a set of configurable parameters are evolved.

Building Block (BB). The evolution of a hardware circuit is based on connecting basic units together. Several levels of complexity in these building blocks are possible:

- **Analog comp. level.** E.g. transistors, resistors, inductors and capacitors.
- **Gate level** E.g. OR and AND gates.
- **Function Level** E.g. sine generators, adders and multipliers.

Target Hardware (THW). In EHW, the goal is to evolve a circuit. The two major alternatives for target hardware available today are:

- **Commercially available devices.** FPGAs are most commonly used. Field-Programmable Analog Arrays (FPAA) are available as well. They use the same programming principle as FPGAs, but they consist of reconfigurable analog components instead of digital gates.
- **Custom hardware.** ASIC (Application Specific Integrated Circuit) is a chip fully designed by the user.

Fitness Computation (FC). Degree of fitness computation in hardware:

- **Offline Fitness Computation (OFL).** The evolution is simulated in software, and only the elite chromosome is written to the hardware device (sometimes named extrinsic evolution).
- **Online Fitness Computation (ONL).** The hardware device gets configured for each chromosome for each generation (sometimes named intrinsic evolution).

Evolution (EV). Degree of evolution undertaken in hardware:

- **Off-chip evolution.** The evolutionary algorithm is performed on a separate processor.
- **On-chip evolution.** The evolutionary algorithm is performed on a separate processor incorporated into the chip containing the target EHW.

- **Complete HW evolution.** The evolutionary algorithm is implemented in special hardware – i.e. it is not running on a processor.

Scope (SC). The scope of evolution:

- **Static evolution.** The evolution is finished before the circuit is put into normal operation. No evolution is applied during normal operation. The evolution is used as a circuit optimizing tool.
- **Dynamic evolution.** Evolution is undertaken while the circuit is in operation and this makes the circuit online adaptable.

Table 1 summarizes the characteristics of the published work on EHW applied to real-world applications. The applications are mainly in the areas of classification and control when complete circuit design is applied. There are also some examples of circuit parameter tuning. A major part of them are based on digital gate level technology using GA as the evolutionary algorithm. However, promising results are given for analog designs, where evolution is used to find optimal parameters for analog components. About half of the experiments are based on custom hardware – or simulation of such. It is more common to put only the fitness computation (ONL), than the whole evolution (On-chip/Complete), on the *same* chip as the target EHW. This is reasonable, since the fitness computation is – as mentioned earlier, the most computational demanding part of the evolution.

Even though there are promising results in evolving systems, there are several obstackles as well that will be discussed in the next section.

3 Evolvable Hardware Scalability

There are several problems concerning evolution and scalability. In this section, two of the major ones will be discussed.

3.1 Chromosome String Length

As mentioned in the introduction, there has been a lack of schemes to overcome the limitation in the chromosome string length [14,37]. A long string is required for representing a complex system. However, a larger number of generations are required by genetic algorithms as the string length increases. This often makes the search space *too* large and explains why only small circuits have been evolvable so far. Thus, work has been undertaken trying to diminish this limitation. Various experiments on *speeding up* the GA computation have been undertaken [1]. The schemes involve fitness computation in parallel or a partitioned population evolved in parallel. Experiments are focussed on speeding up the GA computation, rather than dividing the application into subtasks. This approach assumes that GA finds a solution if it is allowed to compute enough generations. When small applications require weeks of evolution time, there would probably be strict limitations on the systems evolvable even by parallel GA.

Other approaches to the problem have used variable length chromosomes [7]. Another option, called function level evolution, is to apply building blocks more complex than digital gates [19]. Most work is based on fixed functions. However, there has been work in Genetic Programming for *evolving* the functions — called Automatically Defined Functions (ADF) [11].

An improvement of artificial evolution — called co-evolution, has been proposed [6]. In co-evolution, a part of the data which defines the problem co-evolves simultaneously with a population of individuals solving the problem. This could lead to a solution with a better generalization than a solution based only on the initial data. A variant of co-evolution — called cooperative co-evolutionary algorithms, has been proposed by De Jong and Potter [8,21]. It consists of parallel evolution of sub-structures which interact to perform more complex higher level structures. Complete solutions are obtained by assembling representatives from each group of sub-structures together. In that way, the fitness measure can be computed for the top level system. However, by testing a number of individuals from each sub-structure population, the fitness of individuals in a sub-population can be sorted according to their performance in the top-level system. Thus, no explicit local fitness measure for the sub-populations is applied in this approach. However, a mechanism is provided for initially seeding each GA population with user-supplied rules. Darwen and Yao have proposed a co-evolution scheme where the subpopulations are divided without human intervention [2].

Incremental evolution for EHW was first introduced in [30]. The approach is a divide-and-conquer on the evolution of the EHW system, and thus, named *increased complexity evolution*. It consists of a division of the *problem* domain together with incremental evolution of the hardware system. Evolution is first undertaken individually on a set of basic units. The evolved units are the building blocks used in further evolution of a larger and more complex system. The benefits of applying this scheme is both a *simpler* and *smaller* search space compared to conducting evolution in one single run [35].

When considering *manual* hardware design, much *a priori* knowledge is applied [29]. E.g. designing a large circuit would be almost impossible if the designer had to design each sub-circuit from scratch every time it is used [13]. Further, connections are with a mixture of buses and bit-wires. Thus, in the future it is expected to see *intelligent* (or controlled) evolution as an alternative to the normal unconstrained evolution.

3.2 Fitness Computation

To be able to conduct online fitness computation, fast switching between different configurations in a population should be possible. With the available FPGA technology, only a single configuration can be stored at a time within the device. Reloading a new configuration takes too much time to be of interest for evolution. Thus, there has been work on implementing a *user defined FPGA* inside an ordinary FPGA [25,30]. By providing a set of different configuration registers, it is possible to store a population of configurations within an FPGA and perform configuration switching in a single clock cycle [34].

4 Online Adaptation in Real-Time

Most living creatures are able to learn to live in an environment and adapt if some change occur. Artificial systems are far from such an adaptivity. Research on evolutionary methods has – with a few exceptions, been based on one-time evolution. However, there is a wish of being able to design online adaptable evolvable hardware. The hardware would then have to be able to reconfigure its configuration dynamically and autonomously, when operating in its environment [5]. This would be by dynamic evolution.

One possible approach to such a system is to use *two* parallel units. A primary unit is applied for normal runtime operation of an application, while a secondary unit keeps evolving in parallel. If the performance of the secondary unit becomes better than the primary unit, they are exchanged. Thus, the unit giving the best performance at the moment is enabled.

A system for autonomous evolution of robot control has been proposed [10]. A mobile robot learns to track a red ball without colliding with obstacles through dynamic evolution. Thus, all information about the environment is collected concurrently with evolution. This is undertaken through building a model of the environment. Another application with dynamic evolution is image compression [22]. During compression of a given image, evolution search for an optimal set of templates for doing pixel prediction.

So far there is not much work on applying dynamic evolution. This is mainly due to the problem of evolution speed and fitness computation. A set of individuals in the population would have to be evaluated – one at a time, for each generation which is normally time consuming. To make fitness computation you need a feedback from the environment on the performance. This could be difficult to obtain. However, if these problems can be overcome, there is a large potential in being able to provide online-adaptable systems.

5 Conclusions

This paper has introduced EHW and contained a study of the characteristics of EHW applied to real-world applications. Further, problems and new features related to evolution are discussed. For more information about evolvable hardware, there are a couple of special journal issues [36,26] and some books [40,24]. The two main conferences in the field are International Conference on Evolvable Systems: From Biology to Hardware (Springer LNCS publisher) and NASA/DoD Conference on Evolvable Hardware (IEEE publisher).

References

1. E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.
2. P. Darwen and X. Yao. Automatic modularization by speciation. In *Proc. of 1996 IEEE International Conference on Evolutionary Computation*, pages 88–93, 1996.
3. D. Goldberg. *Genetic Algorithms in search, optimization, and machine learning*. Addison–Wesley, 1989.
4. T. Higuchi et al. Evolvable hardware: A first step towards building a Darwin machine. In *Proc. of the 2nd International Conference on Simulated Behaviour*, pages 417–424. MIT Press, 1993.

5. T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, B. Manderick, and T. Furuya. Evolvable hardware and its applications to pattern recognition and fault-tolerant systems. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware: The evolutionary Engineering Approach*, volume 1062 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 1996.
6. W.D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990.
7. M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi. A pattern recognition system using evolvable hardware. In *Proc. of Parallel Problem Solving from Nature IV (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 761–770. Springer-Verlag, September 1996.
8. K.A. De Jong and M.A. Potter. Evolving complex structures via co-operative coevolution. In *Proc. of Fourth Annual Conference on Evolutionary Programming*, pages 307–317. MIT Press, 1995.
9. I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, and T. Higuchi. An evolvable hardware chip and its application as a multi-function prosthetic hand controller. In *Proc. of 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 182–187, 1999.
10. D. Keymeulen et al. On-line model-based learning using evolvable hardware for a robotics tracking systems. In *Genetic Programming 1998: Proc. of the Third Annual Conference*, pages 816–823. Morgan Kaufmann, 1998.
11. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, 1994.
12. J. R. Koza et al. *Genetic Programming III*. San Francisco, CA: Morgan Kaufmann Publishers, 1999.
13. J.R. Koza et al. The importance of reuse and development in evolvable hardware. In J. Lohn et al., editor, *Proc. of the 2003 NASA/DoD Conference on Evolvable Hardware*, pages 33–42. IEEE, 2003.
14. W-P. Lee, J. Hallam, and H.H. Lund. Learning complex robot behaviours by evolutionary computing with task decomposition. In A. Birk and J. Demiris, editors, *Learning Robots: Proc. of 6th European Workshop, EWLR-6 Brighton*, volume 1545 of *Lecture Notes in Artificial Intelligence*, pages 155–172. Springer-Verlag, 1997.
15. J.D. Lohn and S.P. Colombano. A circuit representation technique for automated circuit design. *IEEE Trans. on Evolutionary Computation*, 3(3):205–219, September 1999.
16. J. F. Miller. Digital filter design at gate-level using evolutionary algorithms. In W. Banzhaf et al., editors, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99)*, pages 1127–1134. Morgan Kaufmann, 1999.
17. M. Murakawa et al. Analogue EHW chip for intermediate frequency filters. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98*, pages 134–143. Springer-Verlag, 1998. Lecture Notes in Computer Science, vol. 1478.
18. M. Murakawa et al. The grd chip: Genetic reconfiguration of dsps for neural network processing. *IEEE Transactions on Computers*, 48(6):628–638, June 1999.
19. M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. In *Proc. of Parallel Problem Solving from Nature IV (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 62–71. Springer-Verlag, September 1996.
20. R. Porter et al. An applications approach to evolvable hardware. In *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, 1999.
21. M.A. Potter and K.A. De Jong. Evolving neural networks with collaborative species. In *Proc. of Summer Computer Simulation Conference*. The Society for Computer Simulation, 1995.

22. Sakanashi et al. Evolvable hardware chip for high precision printer image compression. In *Proc. of 15th National Conference on Artificial Intelligence (AAAI-98)*, 1998.
23. L. Sekanina. Toward uniform approach to design of evolvable hardware based systems. In R.W. Hartenstein et al., editors, *Field-Programmable Logic and Applications: 10th International Conference on Field Programmable Logic and Applications (FPL-2000)*, pages 814–817. Springer-Verlag, 2000. Lecture Notes in Computer Science, vol. 1896.
24. L. Sekanina. *Evolvable Components: From theory to Hardware Implementations*. Springer-Verlag, 2004. ISBN 3-540-40377-9.
25. L. Sekanina and R. Ruzicka. Design of the special fast reconfigurable chip using common F PGA. In *Proc. of Design and Diagnostics of Electronic Circuits and Sy stems - IEEE DDECS'2000*, pages 161–168, 2000.
26. M. Sipper and D. Mange. Special issue on from biology to hardware and back. *IEEE Trans. on Evolutionary Computation*, 3(3):165–250, September 1999.
27. E. Takahashi et al. An evolvable-hardware-based clock timing architecture towards gigahz digital systems. In *Proc. of the Genetic and Evolutionary Computation Conference*, 1999.
28. A. Thompson. Exploration in design space: Unconventional electronics design through artificial evolution. *IEEE Trans. on Evolutionary Computation*, 3(3):171–177, September 1999.
29. J. Torresen. Exploring knowledge schemes for efficient evolution of hardware. In *Proc. of the 2004 NASA/DoD Conference on Evolvable Hardware*.
30. J. Torresen. A divide-and-conquer approach to evolvable hardware. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98*, volume 1478 of *Lecture Notes in Computer Science*, pages 57–65. Springer-Verlag, 1998.
31. J. Torresen. Increased complexity evolution applied to evolvable hardware. In Dagli et al., editors, *Smart Engineering System Design: Neural Networks, Fuzzy Logic , Evolutionary Programming, Data Mining, and Complex Systems, Proc. of ANNIE'99*, pages 429–436. ASME Press, November 1999.
32. J. Torresen. Possibilities and limitations of applying evolvable hardware to real-world application. In R.W. Hartenstein et al., editors, *Field-Programmable Logic and Applications: 10th International Conference on Field Programmable Logic and Applications (FPL-2000)*, volume 1896 of *Lecture Notes in Computer Science*, pages 230–239. Springer-Verlag, 2000.
33. J. Torresen. Scalable evolvable hardware applied to road image recognition. In J. Lohn et al., editor, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 245–252. IEEE Computer Society, Silicon Valley, USA, July 2000.
34. J. Torresen and K.A. Vinger. High performance computing by context switching reconfigurable logic. In *Proc. of the 16th European Simulation Multiconference (ESM 2002)*, pages 207–210. SCS Europe, June 2002.
35. Jim Torresen. A scalable approach to evolvable hardware. *Journal of Genetic Programming and Evolvable Machines*, 3(3):259–282, 2002.
36. X. Yao. Following the path of evolvable hardware. *Communications of the ACM*, 42(4):47–79, 1999.
37. X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. In T. Higuchi et al., editors, *Evolvable Systems: From Biology to Hardware. First International Conference, ICES 96*, volume 1259 of *Lecture Notes in Computer Science*, pages 55–78. Springer-Verlag, 1997.
38. M. Yasunaga et al. Evolvable sonar spectrum discrimination chip designed by genetic algorithm. In *Proc. of 1999 IEEE Systems, Man, and Cybernetics Conference (SMC'99)*, 1999.
39. M. Yasunaga et al. Gene finding using evolvable reasoning hardware. In P. Hadddow A. Tyrrel and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Fifth International Conference, ICES'03*, volume 2606 of *Lecture Notes in Computer Science*, pages 228–237. Springer-Verlag, 2003.
40. R. Zebulum et al. *Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms*. CRC Press, 2001. ISBN 0849308658.

A Runtime Environment for Reconfigurable Hardware Operating Systems*

Herbert Walder and Marco Platzner

Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
walder@tik.ee.ethz.ch

Abstract. We present a runtime environment that partially reconfigures and executes hardware tasks on Xilinx Virtex. To that end, the FPGA's reconfigurable surface is split into a varying number of variable-sized vertical task slots that can accommodate the hardware tasks. A bus-based communication infrastructure allows for task communication and I/O. We discuss the design of the runtime system and its prototype implementation on an reconfigurable board architecture that was specifically tailored to reconfigurable hardware operating system research.

1 Introduction and Related Work

Reconfigurable hardware operating systems (RHWOS) are a rather new line of research. Analogous to classical software operating systems, RHWOS allow to compose applications out of communicating tasks. Such hardware tasks represent digital circuits with coarse-grained functionality. A RHWOS executes a set of hardware tasks on a reconfigurable device in a truly-parallel multitasking manner. While the use of an operating system leads to overheads in terms of required area and execution time, a RHWOS also offers a number of benefits. A reconfigurable architecture controlled by an operating system can easily execute dynamic task sets and utilize modern high-density FPGAs. Further, an operating system introduces a minimal programming model and, thus, leads to improved productivity and portability.

Concepts for a RHWOS and the swapping of hardware tasks were first discussed in [1]. Basic multitasking services were investigated in [2]. Especially, the problem of finding placements for hardware tasks on a reconfigurable surface has received attention, e.g, in [3] [4]. More general aspects and services of a RHWOS were discussed in [5] and [6]. There are only few reports on practical implementations of a RHWOS on real FPGA technology, e.g., in [7] and [6]. Due to the limited partial reconfiguration capabilities of current FPGAs, most approaches follow a one-dimensional resource model and partition the reconfigurable surface into a number of fixed-size slots that can hold hardware tasks.

In this paper, we present the design and the implementation of a RHWOS runtime environment on real FPGA hardware (Xilinx Virtex-II). Like related projects, our runtime environment structures the reconfigurable area into vertical task slots. Contrary to related work, our RHWOS works with variable-sized task slots and can thus better adapt the reconfigurable resources to the incoming tasks. A bus-based communication infrastructure is used to connect the hardware tasks with memory resources and I/O.

* This work is supported by the Swiss National Science Foundation (SNF) under grant number 2100-59274.99 and by Xilinx, Inc. by hardware donations.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 831–835, 2004.

2 Reconfigurable Hardware OS Approach

As target architecture, we consider an embedded system consisting of a CPU and a partially reconfigurable FPGA. The CPU runs operating system functions that manage the reconfigurable system resources at runtime. To gain short reconfiguration times, the CPU directly connects to the configuration port of the FPGA. In parallel, a number of wires implement general purpose I/O between CPU and FPGA which allows for high bandwith communication.

An operating system offers an application model in the sense that i) applications are composed out of user tasks and operating system objects and ii) scheduling policies are provided to decide which of the ready tasks should execute. Especially real-time operating systems (RTOS) offer rich sets of objects for task synchronization and communication together with priority-based preemptive scheduling.

We aim at offering similar objects and services to hardware tasks in a RHWOS. An application consists of user hardware tasks and several operating system objects. Many of these objects offer communication services, such as FIFOs, message boxes, and private and shared memory blocks. Task synchronization is supported by semaphore objects, timers, and triggers. Device drivers allow for I/O access.

In contrast to software RTOSs that manage a single computational resource which is either fully allocated by a task or empty at any given point in time, a RHWOS manages a resource which can also be occupied partially and by several tasks concurrently.

A hardware task is a functional entity with a set of ports. The function carried out by a hardware task is invisible for the RHWOS. A *control port* connects a hardware task to the RHWOS modules such as the scheduler. Further, each task may implement a number of *input ports* and *output ports*. These are independent parallel interfaces and allow a task to access operating system objects. Finally, a *clock port* is required to drive the task with a proper clock.

Based on the knowledge of the application structure, the RHWOS schedules and loads hardware tasks to the FPGA and replaces them after they have finished execution. To enable this, an FPGA needs to provide some infrastructure which we denote *runtime environment*. The implementation of such a runtime environment strongly depends on the architectural features of the FPGA, especially on its reconfiguration features.

3 Runtime Environment Based on Xilinx Virtex-II

The most important characteristic of Xilinx Virtex-II is its column-wise chip-spanning partial reconfiguration which asks for a one-dimensional resource model. Consequently, we partition the user task area into a number of vertical task slots. The main feature of our runtime system with respect to resource management is its ability to deal with variable-sized slots. In the following, we emphasize on the partitioning of the reconfigurable surface and the communication infrastructure.

3.1 OS Frames and Variable-Sized Task Slots

The FPGA surface is split into a static and a dynamic region. The static region comprises all operating system modules and is organized into two OS frames located at the left

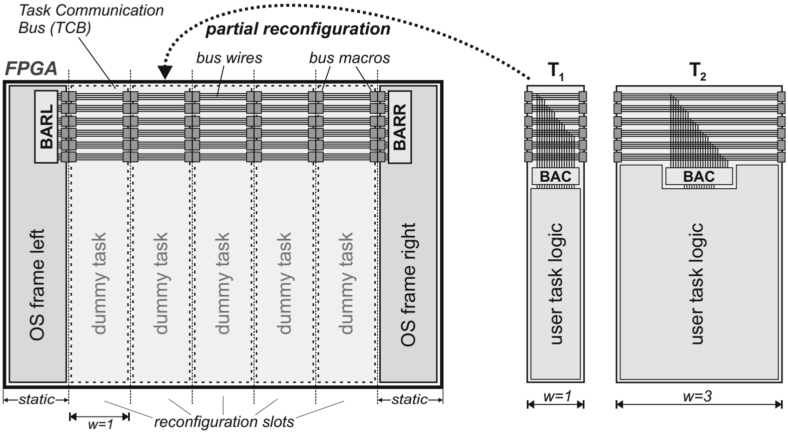


Fig. 1. OS frame (left, right) and Task Communication Bus (TCB)

and right edges of the FPGA. This is shown in Figure 1. The dynamic region comprises logic resources available for user hardware tasks which are dynamically loaded.

Whenever the system is powered up, the FPGA undergoes a full configuration. This initial configuration contains the OS frames with the RHWOS elements and the dynamic area organized into a number of dummy tasks. Dummy tasks are placeholders for user tasks. They do not implement any functionality but establish the communication infrastructure of the runtime environment. Each dummy task implements a part of the overall communication infrastructure consisting of bus macros and bus wires. The width of a dummy task defines a static grid of reconfigurable slots.

When a user task is loaded, it occupies an integer multiple of the width of dummy tasks. Figure 1 shows two user tasks T_1 and T_2 with different widths $w = 1$ and $w = 3$, respectively, which equals 1× and 3× the dummy task width. Tasks with widths $w > 1$ do not necessarily need to implement the bus macros within their area. Implementing the bus macros at the user task borders suffices. After a task with width $w > 1$ has terminated, the occupied area is again filled with dummy tasks to re-establish the communication infrastructure for subsequent user tasks. Dummy tasks as well as user tasks are available as partial bitstreams.

3.2 Task Communication Bus (TCB)

A RHWOS runtime environment has to provide a communication infrastructure for the exchange of data between user tasks and OS objects. There are several ways to implement such a communication infrastructure, with trade-offs between the achieved data throughput and the needed resources. Assigning each task a set of dedicated wires and turning the communication infrastructure into a crossbar would result in the highest-possible throughput. At the same time, a crossbar also leads to enormous area requirements and is not scalable. We rely on a diametrical solution and implement a shared bus structure as communication network. A bus requires less resources but delivers also a lower communication bandwidth. We use two methods to scale the throughput, varying the bus width and using several independent busses. However, the bus approach scales poorly with the FPGA size. Future devices with strongly increased densities will allow to accommodate dozens of user tasks at the same time. Then, the bus would become a severe bottleneck. For such architectures, a two-dimensional partitioning into slots and a 2D communication infrastructure will be better choices [7]. We have decided for a bus structure in this work, as currently neither a 2D partitioning nor a 2D communication network is supported by the available FPGAs.

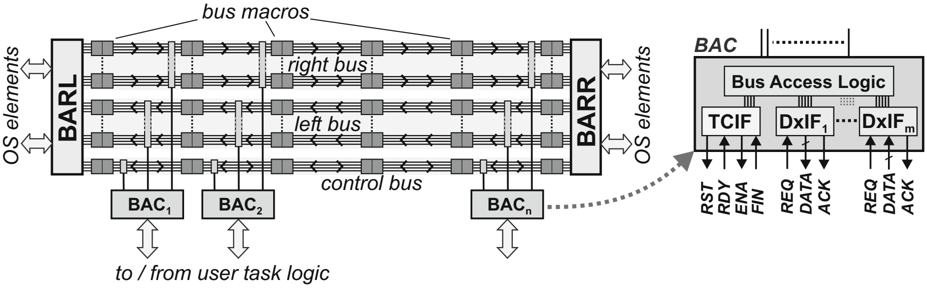


Fig. 2. Task Communication Bus (TCB): structure and elements

Figure 2 shows the bus structure with its elements. The bus system consists of bus wires, bus macros, two *bus arbiters (BARL, BARR)* located in the right and left OS frames, and a *bus access controller (BAC)* in each user task. The bus is split into a left bus, a right bus, and a control bus. The left bus serves read/write accesses to OS elements located in the left OS frame and the right bus accesses OS elements in the right frame, respectively. The left bus is arbitrated by the BARL, and the right bus by the BARR. The control bus comprises request/grant signals connecting each BAC to both arbiters.

The bus access controller (BAC) provides an interface that consists of the *bus access logic* and, at the user task's side, of a *task control interface (TCIF)* and m *data exchange interfaces (DxIF)*. The bus access logic handles the bus reservation and implements the data transaction protocol. Thus, the actual bus protocol is hidden from the user task. The control and data interfaces connect to the task's control and data ports (see Section 2). The TCIF controls the task's state and provides a standard set of signals which each user task is obligated to implement.

4 Prototype Implementation

To realize RHWOS prototypes, we have developed the *XF-Board* platform. For a detailed description of the XF-Board and its features, we refer to [8]. We have successfully tested the full and partial reconfiguration capabilities. The device, a Xilinx Virtex-II XC2V3000-4, contains 14336 slices and is fully configured in 24.3 ms. Although the bus macros dictate a minimal dummy task width of 4 CLBs, we have used a width of 8 CLBs in our tests for both the dummy tasks and the OS frames. A user task of width $w = 1$ is then partially reconfigured in 3.4 ms, a task of width $w = 5$ in 17.3 ms.

5 Conclusion

In this paper, we have presented the design and implementation of a runtime environment for reconfigurable hardware operating systems. The runtime environment is able to dynamically load and execute tasks of variable size. A bus-based communication infrastructure allows for task communication and I/O. Ongoing work includes the thorough test of the developed OS elements and the user tasks as well as a quantitative evaluation of the OS overheads.

References

1. Gordon Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 327–336. Springer, 1996.
2. H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA Coprocessors. In *Proceedings of the 10th International Workshop on Field Programmable Gate Arrays (FPL)*, pages 121–130. Springer, 2000.
3. Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test of Computers*, volume 17, pages 68–83, 2000.
4. Herbert Walder and Marco Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 290–295. IEEE Computer Society, March 2003.
5. Grant Wigley and David Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE CS Press, April 2001.
6. Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287. CSREA Press, June 2003.
7. J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, Vernalde S., and R. Lauwreins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 986–991. IEEE Computer Society, March 2003.
8. Herbert Walder, Samuel Nobs, and Marco Platzner. XF-Board: A Prototyping Platform for Reconfigurable Hardware Operating Systems. In *Proceedings of the 4th International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*. CSREA Press, June 2004.

A Dynamically Reconfigurable Asynchronous FPGA Architecture

Xin Jia, Jayanthi Rajagopalan, and Ranga Vemuri

University of Cincinnati
{jiax, rajagoji, ranga}@ececs.uc.edu

Abstract. This paper presents APL, an Asynchronous Programmable Logic array, as a flexible dynamically reconfigurable multi-application platform with self-reconfigurability. APL employs a Globally-Asynchronous-Locally-Synchronous (GALS) architecture. It consists of Timing Regions (TRs) which operate independently under locally generated clocks and communicate with each other through handshaking asynchronous interfaces. Different applications are mapped into different TRs, so they can run independently. And because of the asynchronous communication, dynamic partial reconfiguration is easily realized with TRs as the basic reconfiguration units. Self-reconfiguration is realized by giving each TR the capability to access the central configuration controller which, in turn, can read/write the configuration memory.

1 Introduction

Future products of reconfigurable computing system are likely to be multi-standard and multi-application. This requires a FPGA to provide multiple clock domains and data communication mechanisms among different clock domains. Dynamic partial reconfiguration is needed for modification of different applications. Furthermore, self-reconfiguration is required for implementation of applications that are data/environment sensitive. Current FPGAs can hardly meet those requirements. Globally-Asynchronous-Locally-Synchronous (GALS) design, which allows independent operation of synchronous islands, is a suitable solution. We propose APL, an Asynchronous Programmable Logic array which adopts GALS architecture, as a flexible dynamically reconfigurable multi-application platform. APL consists of timing regions (TRs) that operate as basic application mapping units and reconfiguration units. Because of the asynchronous properties between TRs, dynamic partial reconfiguration becomes easier and more efficient. Self-reconfiguration is realized by giving each TR the capability to access the central configuration controller which, in turn, can read/write the configuration memory.

Different GALS systems have been proposed by previous researchers [1, 2, 3] and several asynchronous FPGA architectures have been published in the last decade [4, 5, 6, 7]. STACC [5] is a dual-layer GALS FPGA architecture. Our APL design adopts the similar idea from STACC in forming the local clock generator and asynchronous wrapper of a GALS system but provides more efficient routing structure and dynami-

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 836–841, 2004.

cally partial reconfiguration and self-reconfiguration mechanism. PCA [6] contains a programmable logic layer and a network of built-in-facilities, but the wormhole message passing between objects in PCA is expensive in time.

Several FPGAs are self-reconfigurable [8, 9, 10]. They all only allow one reconfiguration generator. APL allows arbitrary number of TRs been configured as reconfiguration generators and therefore, is more suitable to a multi-application scenario.

2 APL Architecture

APL employs a dual-layer structure composed of a timing layer and a logic layer. Both the timing layer and the logic layer adopt hierarchical routing structure which provides various types of routing wires. The whole array is divided into timing regions. A TR consists of a Timing Cell from the Timing Layer and Logic cell from the Logic layer. Each TR is driven by a locally generated clock signal and communicates with other TRs through a 4-phase handshaking protocol.

APL timing cell design is similar to and inspired by STACC [5] timing cell design. The main concern of logic cell design is to determine the logic size and granularity. An appropriate decision must be a compromise of handshaking overhead, reconfiguration time and logic utilization. For the first version of APL, we choose each logic cell to contain 4 logic blocks (LBs). Each LB adopts the same structure as the CLB structure of Xilinx Virtex FPGA [11] which contains four 4-input LUTs. Thus, each logic cell contains a total of 256 SRAM bits.

APL adopts a hierarchical routing structure for both timing layer and logic layer. In hierarchical routing, the routing delays are more predictable. APL only supports bundled-data protocol, and the performance of this protocol is eventually decided by how precisely the interconnect delays can be estimated. By employing hierarchical

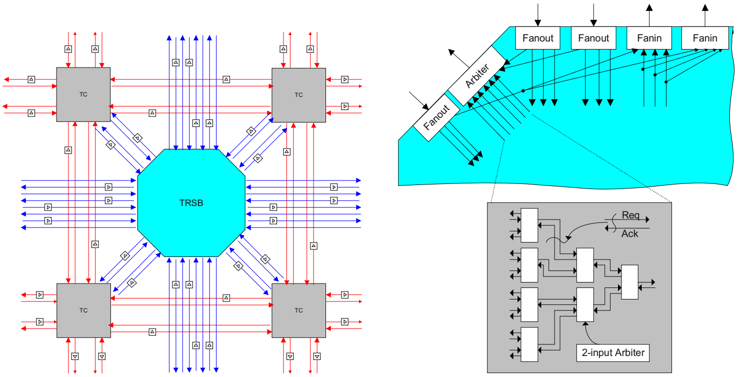


Fig. 1. APL timing layer routing structure and TRSB circuit design

routing structure, routing, hence, routing delay, is much more predictable than using symmetrical structure. The timing layer routing strictly follows the same fashion as the logic layer routing, which is crucial for a better performance in bundled-data protocol. Left part of Fig. 1 shows the APL timing layer routing structure.

A Timing Routing Switch Box (TRSB) as shown in right part of Fig. 1 provides four handshaking pairs in each direction to its four neighbors. Three basic functions need to be realized inside a TRSB: fanin, fanout and arbitration. Fanin and fanout are used to distribute data to and from different locations and arbitration is needed when data compete to the same location.

3 Dynamic Partial Reconfiguration

Dynamic partial reconfiguration is possible in APL architecture through a parallel configuration port. A timing region is treated as a basic reconfiguration unit. Inside each reconfiguration unit, memories are read and written serially. A local memory controller attached with each reconfiguration unit controls the serial read/write of the reconfiguration unit memory. Left part of Fig. 2 shows the memory organization.

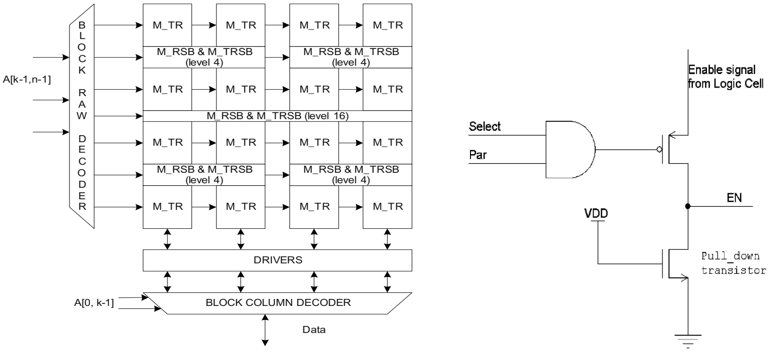


Fig. 2. Dynamic partial reconfiguration of APL

To do dynamic partial reconfiguration, it is essential to guarantee that the reconfiguration of one part of the FPGA will not cause false operations of the other parts. This can be easily realized due to the asynchronous feature of the APL architecture. What we need to do is to simply disable all the input/output ports of the corresponding timing cell when the reconfiguration process begins. A simple enable/disable circuit has been added to the timing cell for this purpose as shown in right part of Fig. 2. Signal Par is the partial reconfiguration mode select signal. It only turns high during the partial reconfiguration period. Signal Select comes from the block raw and column decoders. It is high when the corresponding reconfiguration unit is selected. The n-transistor works as a pull-down resistor. This way, the En signals are driven low when the corresponding partial reconfiguration begins.

4 Self-Reconfiguration

Each TR has a dedicated reconfiguration port that contains a handshaking signal pair and 4-bit reconfiguration data lines. The reconfiguration port connects to the central configuration controller that controls the modification of the configuration memory. Since there is only one central configuration controller that can directly access the configuration memory, reconfiguration requests from different TRs must be processed sequentially. Therefore, an arbitration mechanism among different reconfiguration requests is needed. The left part of Fig. 3 shows the self-reconfiguration structure.

This structure also fits the hierarchical routing structure of APL very well: All the arbitration modules are placed at the same position of the corresponding RSBs, and the reconfiguration data lines are multiplexed with the interconnect wires. Therefore, no extra data lines are needed for the reconfiguration purpose. The circuit that realizes the arbitration function is called cfg-arbiter here. The cfg-arbiter performs differently from the ordinary arbiter. The reconfiguration process takes many handshaking cycles. Once a reconfiguration request wins the arbitration, the cfg-arbiter must hold the path for the request until the reconfiguration process is done. The right part of Fig. 3 shows the circuit implementation of the cfg-arbiter.

In the cfg-arbiter design, an exclusive element is used to grants service to only one request and delays other requests. A rank of D flip-flops is used to catch the result of arbitration and keep the communication channel until the reconfiguration process is done. At that time, a “Done” signal is generated by the central configuration controller to reset the D flip-flops.

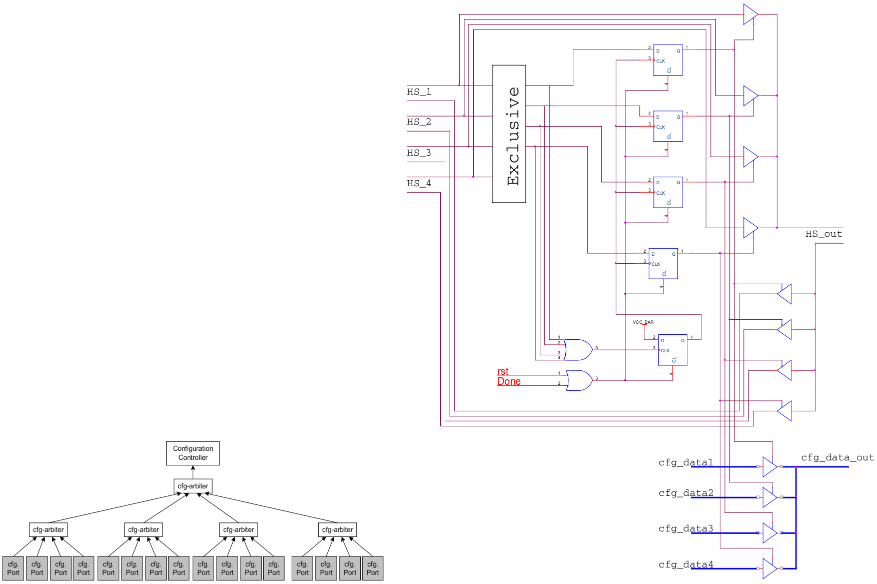


Fig. 3. Self-reconfiguration of APL

Table 1. APL_1 implementation details

Die size	2.5mm × 1.8 mm
4-LUTs	64
Timing regions	4
Timing layer area	0.4 mm^2
Logic layer area	4.2 mm^2
Configuration memory size per TR	1520 bits
t_{cTR}: partial configuration time per TR	1.9 μs
t_{lc}: clock to logic cell output	22.5 ns
t_{dly}: interconnect delay between adjacent timing cells	1.8 ns
t_{hs}: handshaking cycle time between TRs	14.2 ns
t_{cfg}: self-reconfiguration cycle time	15.7 ns

5 Experimental Results

To evaluate our architecture, we implemented a proof-of-concept design APL_1 of the proposed architecture using TSMC0.18μm technology with 6 metal layers. APL_1 contains 4 Timing Regions. Table 1 gives some implementation details.

From the implementation, we can see that the area overhead of adopting the GALS design style is around 10% (the timing layer area). The handshaking communication time is one local clock cycle. But our APL_1 implementation is very small and the timing regions are adjacent to each other, we can expect the handshaking communication time to be longer when the FPGA array size becomes larger and more routing switch boxes are needed.

Our future work will focus on the automatic mapping of designs to APL. The primary goal would be to explore timing region level parallelism while minimizing the communication overhead across timing regions.

References

1. David Bormann, Peter Cheung. Asynchronous Wrapper for Heterogeneous Systems. IEEE International Conference on Computer Design, Oct, 1997
2. W. S. VanScheik, R. F. Tinder, High Speed Externally Asynchronous/Internally Clocked Systems. IEEE Transaction on Computers, July, 1997, vol. 46, No. 7, 824-829
3. Joep Kessels, S. J. Kim, et.al. Clock Synchronization through Handshaking Signalling. 8th International Symposium on Asynchronous Circuits and Systems, April, 2002
4. Andrew Royal, Peter Cheung. Globally Asynchronous Locally Synchronous FPGA Architectures. 13th International Workshop on Field Programmable Logic and Applications, 2003
5. Payne, R. E. Self-Timed FPGA Systems. 5th International Workshop on Field Programmable Logic and Applications, 1995
6. Konishi, R., Hideyuki, I., et al. PCA-1: A Fully Asynchronous, Self-Reconfigurable LSI. In 7th International Symposium on Asynchronous Circuits and Systems, March 2001

7. John Teifel, Rajit Manohar. Highly Pipelined Asynchronous FPGAs. Proceedings of the 12th ACM International Symposium on Field Programmable Gate Arrays, February 2004
8. Xilinx Incorporation, XC6200 Field Programmable Gate Arrays Data Sheet, 1997.
9. Donlin, A., Self-Modifying Circuitry – a Platform for Tractable Virtual Circuitry. 8th International Workshop on Field Programmable Logic and Applications, 1998
10. Sidhu, R., Wadhwa, S., et al. A Self-Reconfigurable Gate Array Architecture, 10th International Workshop on Field Programmable Logic and Applications, August 2000
11. Xilinx Incorporation. Virtex 2.5V FPGA: Complete Data Sheet. Dec. 09, 2002

Hardware Support for Dynamic Reconfiguration in Reconfigurable SoC Architectures*

Björn Griese, Erik Vonnahme, Mario Porrmann, and Ulrich Rückert

Heinz Nixdorf Institute and Department of Electrical Engineering
University of Paderborn, Germany
{bgriese, vonnahme, porrmann, rueckert}@hni.upb.de

Abstract. Dynamic reconfiguration of digital hardware enables systems to adapt to changing demands of the applications and of the environment. This paper concentrates on a core element of a reconfigurable hardware architecture, which supports a Real-time Operating System in reconfiguring the hardware – the Run-time Reconfiguration (RTR) Manager. The RTR-Manager has been developed to control, monitor and execute dynamic reconfiguration. It enables a fast adaptation of the SoC architecture through context switching between different configurations. Configuration code analysis and observation of the reconfiguration process guarantee a correct reconfiguration and increase the safety of the system. The system was successfully implemented and evaluated by means of the Rapid Prototyping System RAPTOR2000.

1 Introduction

A hardware support system for dynamically reconfigurable SoC architectures is introduced, which consists of an embedded processor system and programmable logic integrated in an FPGA. Upon this architecture, a Real-time Operating System (RTOS) is implemented. The RTOS and the FPGA allow for dynamically exchanging software components and reconfiguring hardware modules [1].

The core of our architecture is a Run-time Reconfiguration Manager (RTR-Manager) embedded in hardware, which serves as an interface between the operating system and the reconfigurable logic. It offers services to the RTOS for controlling and monitoring of the hardware configuration, supports the dynamic partial reconfiguration of SoC modules and relieves the RTOS from low-level hardware administration.

By introducing dynamically reconfigurable hardware in SoCs, an additional source of failure emerges. Potential failures are, e.g., replacing the wrong parts of the reconfigurable resources, loading of false or faulty modules, and unforeseen interferences between hardware modules. This could result in not fulfilling real-time criteria, failure of the whole system, or even damage to the hardware. Appropriate mechanisms to prevent such situations have to be implemented, and a basic concept for detecting such errors and coping with them is incorporated in the RTR-Manager.

* This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 842–846, 2004.

2 Architecture of the Reconfigurable System

The proposed architecture consists of an embedded processor running the RTOS. Memory, reconfigurable hardware modules and application specific hardware modules as well as the RTR-Manager are attached to the processor bus. The RTR-Manager uses dedicated interconnections to control the modules and to monitor the states of the modules. In addition to the processor bus, the modules can use a module synchronization bus to communicate with each other without any protocol overhead. As some modules need to communicate with external hardware, the architecture supports controlled access to the user I/Os of the FPGA.

In the current approach, the hardware modules are placed onto the FPGA in particular module slots during run-time. By using partial reconfiguration, only one module slot is changed at a time, while the remaining parts of the system keep working without being interrupted. This type of modules also facilitates the use of a design flow published by Xilinx, which enables the generation of partial reconfigurable hardware modules with a high level description language entry [3]. The implementation of the RTR-Manger can easily be extended to flexible sized modules.

The aim of the introduced architecture is to support methods needed for dynamic reconfiguration of hardware modules. As mentioned in the introduction, different implementations of hardware modules in the FPGA are reconfigured during run-time. A fast transition between the current implementation and a new implementation can be realized if the module, which will be replaced, stays working until the configuration of the new hardware module has finished. The RTR-Manager can switch between the old and the new module in one clock cycle, which we call discrete transition. For the replacement of modules it may also be necessary that the new module obtains status information, e.g., register entries, from the current module. This information can be exchanged via the module synchronization bus.

In some cases exchanging internal states is not sufficient to synchronize two modules. E.g., the transition between different controller modules for automation applications has to be smooth, i.e., it is necessary to synchronize the two controllers. During the synchronization, both modules receive the same information from the processor and the external user I/Os. Once the controllers are synchronized, the processor initiates the discrete transition, and the old module will be removed.

RTR-Manager

The main function of the RTR-Manager is to control and observe the dynamic reconfiguration. This service of the RTR-Manager is provided to the RTOS on the processor system. For the purpose of optimization the software requests the RTR-Manager to reconfigure the FPGA with new modules at run-time. The RTOS also provides the memory address of the FPGA configuration bitstream and it selects the module slot to be reconfigured. Furthermore, the RTOS can select several modes for the dynamic reconfiguration. This includes a direct activation of a module after the reconfiguration and an automated status transfer between two modules.

To initiate the process of dynamic reconfiguration, the processor sends an instruction to the RTR-Manager. First of all, the RTR-Manager deactivates the module that will be replaced. Just before the RTR-Manager starts the reconfiguration, it analyzes the header of the configuration file for security purposes. Afterwards, the RTR-Manager triggers the reconfiguration of the FPGA. When the reconfiguration is completed, the RTR-Manager receives a status message indicating the correctness of the reconfiguration. If no error occurs during reconfiguration and the RTOS has selected the corresponding options, the RTR-Manager starts the module transfer and activates the module. In case of an error, the RTR-Manager immediately stops the dynamic reconfiguration and triggers a processor interrupt. Furthermore, the RTR-Manager supports the synchronization and cross-fading of two modules. With the help of the RTR-Manager, modules can be set into a passive mode. In this mode, the RTR-Manager deactivates the write access to the user I/Os and so the modules are just able to listen to these I/Os. Active modules are able to drive both bus and I/O signals; deactivated modules can only be reactivated by the RTR-Manager.

The analysis of the FPGA configuration code belongs to the security concept of the reconfigurable system and is implemented as a part of the RTR-Manager. The RTR-Manager checks every new configuration bitstream before it is loaded into the FPGA. This is very important as faulty bitstreams can overwrite and destroy the SoC architecture inside the FPGA. The current RTR-Manager checks if the configuration file was generated for the correct FPGA, and if the header has the correct form. Further tests for the partial reconfiguration can be integrated easily, e.g., tests of the size and the position of a partial reconfigured module inside the FPGA have to be conducted.

3 Implementation and Test

The reconfigurable SoC architecture described above is implemented on our RAPTOR2000 Rapid Prototyping System [2], which is able to execute full or partial configuration either at start-up or at run-time. RAPTOR2000 consists of a PCI based motherboard and up to six daughterboards (in the context of RAPTOR2000, the term *module* refers these daughterboards).

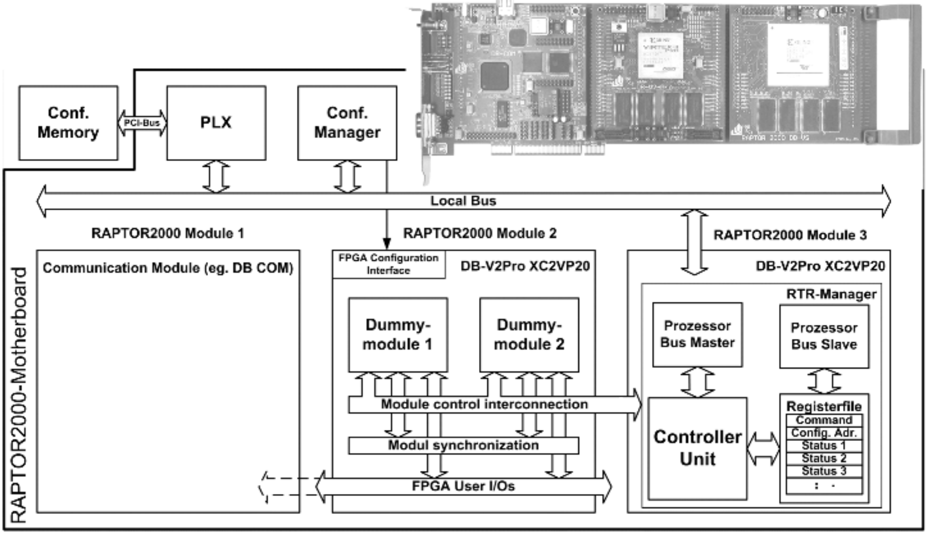


Fig. 1. Prototyping architecture

The DB-V2 Pro FPGA module, used for the reconfigurable SoC, contains a XC2VP20/30 Virtex-II Pro FPGA and up to 256 MB SDRAM. The RTOS is executed by one of the embedded PowerPC processors and the reconfigurable logic is used for the other parts of the SoC architecture. Finally, all building-blocks of our architecture fit into a single FPGA. However, for a first implementation the SoC architecture is partitioned into two designs, located on two DB-V2Pro FPGA modules. As shown in Fig. 1, the RTR-Manager is implemented in the rightmost FPGA, while the reconfigurable SoC modules are mapped onto the center FPGA module, because the latter is our target for partial dynamic reconfiguration. In the current implementation the configuration memory is located in the main memory of the host computer. In order to minimize communication via the PCI Bus, future implementations, will support the storage of the configuration data in the SDRAM on the FPGA module.

As illustrated in Fig. 1, two modules have been designed, which implement a standardized interface between the SOC modules and the RTR-Manager. With these modules, the data communication between the SoC modules was tested, and the possibility to switch over from one SoC module to another has been proved. Additionally, the activation of SoC modules by the RTR-Manager and the write access control of the SoC modules have been verified. An application specific graphical user interface for controlling and monitoring of our SoC architecture has been implemented using the RAPTOR2000 software environment. The software displays all registers of the RTR-Manager. Additionally, the user can access the register entries, e.g., to start the reconfiguration or to activate a module.

4 Results

The VHDL description of the RTR-Manager has been synthesized for a XC2VP20-6 Virtex-II Pro FPGA, resulting in a maximum clock frequency of 53.2 MHz and a size of 861 slices, i.e., 9 % of the FPGA resources. By means of a logic analyzer, we have measured the reconfiguration time, including the time for the configuration code analysis, the module transfer and the module activation. Table 1 depicts the results of the measurements, showing that most of the time is spend on the configuration of the FPGA. With 3 µs, the time consumed by the RTR-Manager is very small compared to the configuration time of the FPGA (0.02%).

Table 1. Measured reconfiguration time on the Virtex-II Pro at 25 MHz.

Operation	Time	Clock Cycles
Initialize RTR-Manager	0.12 µs	3
Configuration code analysis	2.20 µs	55
Start configuration manager	0.28 µs	7
Status request	0.19 µs	5
Synchronization and activation	0.19 µs	5
Total RTR-Manager time	3.00 µs	75
Partial Configuration (57 KB)	14.325 ms	358,125
Total reconfiguration time	14.328 ms	358,200

For the configuration of the FPGA we use the "Select Map" interface of the Virtex-II Pro, which transmits one byte per clock cycle. To compare the measured configuration time with the minimum possible values, we calculate the possible theoretical value for the reconfiguration of the Virtex-II Pro FPGA. With a clock frequency of 50 MHz the transfer rate *r* is 50 MB/s. The length *L* of the tested partial reconfiguration file for the XC2VP20 is 57 KB. The reconfiguration time is T_{conf}= L/r. With 1.14 ms, the minimum reconfiguration time is much less than the 14.33 ms that are given in Table 1. This is, on the one hand, due to the current design frequency of 25 MHz. On the other hand, the data transfer of the bitstream from the external configuration memory (located in the host PC) to the FPGA has not yet been optimized.

To provide fast dynamic reconfiguration, it is important that the RTR-Manager does not significantly increase the reconfiguration time. Therefore, the time that is spent by the RTR-Manager is compared to the minimum time that is required for a complete and for a partial reconfiguration. It can be seen that the time required by the RTR-Manager is even small compared to the minimum reconfiguration time, which is spent for a partial reconfiguration with a bitstream of 57 KB (<0.27%).

5 Summary and Outlook

In this paper we have presented a hardware support system for dynamically reconfigurable SoC architectures. A Run-time Reconfiguration Manager has been developed, which supports the real-time operating system in controlling and reconfiguring the hardware modules inside an FPGA. All functions of the RTR-Manager have been tested successfully on our RAPTOR2000 prototyping system, i.e., the interaction with the software, the configuration code analysis for safety reasons, and the dynamical reconfiguration of an FPGA. The influence of the RTR-Manager on the FPGA reconfiguration time is below 0.27%, even if small partial bitstreams are loaded at the maximum configuration speed of 50 MB/s. Finally, a hardware support architecture is now available to implement a dynamically reconfigurable system.

Our future work will focus on the implementation of hardware IP modules, i.e., an operating system scheduler, and real-time communication modules. These modules will allow for generating partial bitstreams with reasonable functionality. Further work will be spent on using the internal configuration access point (ICAP) of the Virtex-II Pro for FPGA reconfiguration.

References

1. K. Compton, S. Hauck. Configurable Computing: A Survey of Systems and Software. ACM Computing Surveys, Vol. 34, No. 2. pp. 171-210. June 2002
2. H. Kalte, M. Porrmann and U. Rückert. A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In: Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC*)*, Hamburg, Germany, 2002.
3. D. Lim and M. Peattie. Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations. Xilinx, Mai 2002.

Optimal Routing-Conscious Dynamic Placement for Reconfigurable Devices

Ali Ahmadinia[1], Christophe Bobda[1], Sándor P. Fekete[2],
Jürgen Teich[1], and Jan C. van der Veen[2]

[1] Department of Computer Science 12, University of Erlangen-Nuremberg, Germany
{ahmadinia,bobda,teich}@cs.fau.de
[2] Department of Mathematical Optimization, Braunschweig University of Technology, Germany, {s.fekete,j.van-der-veen}@tu-bs.de.

Abstract. We describe algorithmic results for two crucial aspects of allocating resources on computational hardware devices with partial reconfigurability. By using methods from the field of computational geometry, we derive a method that allows correct maintainance of free and occupied space of a set of n rectangular modules in optimal time $\Theta(n \log n)$; previous approaches needed a time of $O(n^2)$ for correct results and $O(n)$ for heuristic results. We also show that finding an optimal feasible communication-conscious placement (which minimizes the total weighted Manhattan distance between the new module and existing demand points) can be computed in $\Theta(n \log n)$. Both resulting algorithms are practically easy to implement and show convincing experimental behavior.

Keywords: Reconfigurable computing, field-programable gate array (FPGA), module placement, occupied space manager (OSM), routing-conscious placement, Manhattan metric, line sweep technique, optimal running time, lower bounds.

1 Introduction

One of the cutting-edge aspects of reconfigurable computing is the possibility of *partial* reconfiguration of a device. In this paper we resolve two crucial issues for this task:

1. Given a set of n rectangular modules that have been placed on a chip, identify all feasible positions for a new module.
2. Given a set of n rectangular modules that have been placed on a chip, a new module, and demands for connecting it to existing sites, find a feasible position for the module that minimizes the total weighted distance to the given sites.

Related Work. The first of the above issues is the task of maintaining free space. Bazargan et al. [3] describe how to achieve this by maintaining the set of all maximal free rectangles; as this set can have size $\Omega(n^2)$, the complexity is quadratic. Alternatively, they propose partitioning free space into only $O(n)$ free rectangles; the price for this improved complexity is the fact that no feasible placement may be found, even though one exists. Walder et al. [7] have suggested ways to reduce this deficiency and did report on experimental improvement, but their $O(n)$ procedure is still a heuristic approach that may fail in some scenarios. Thus, there remains a gap between $O(n^2)$ methods that report

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 847–851, 2004.

an accurate answer, and $O(n)$ heuristics that may fail in some scenarios. Ahmadinia et al. [1] suggested maintaining occupied space instead of free space, but (depending on the computational model) their approach is still quadratic.

The more difficult task of routing-conscious placement has received less attention: Clearly, *optimal* placement of a new module has to go beyond feasible placement. For configurable computing, this second aspect has only been treated very recently, in work by Ahmadinia et al. [1], who suggest a heuristic to find a feasible placement for a new module with small total weighted Euclidean distance to a set of demand points. However, according to [5], using Manhattan distances is more appropriate.

Our Results. We resolve both of the above issues:

- We give a $O(n \log n)$ method to provide an occupied space manager (OSM). This approach uses a plane-sweep approach from computational geometry.
- We give a matching lower bound of $\Omega(n \log n)$ for locating a maximal free rectangle between a set of n modules, showing that our method has optimal complexity.
- We show that our OSM can be extended to find a feasible position that minimizes total weighted Manhattan distance to existing sites. The resulting algorithm still has an optimal running time of $\Theta(n \log n)$.
- We describe implementation details to illustrate that our method is fast and easy.
- We provide experimental data to demonstrate the practical usefulness of our results.

In Section 2, we present our optimal OSM. Section 3 describes optimal routing-conscious placement, followed by implementation details and experimental data in Section 4. See [2] for a full version of our paper.

2 The Occupied-Space Manager

Our occupied-space manager is based on a modification of the well-known algorithm ContourOfUnionOfRectangles (CUR) [6] that finds the contour of a union of axis-parallel rectangles. As the number of contour segments is linear in n, we achieve a running time of $O(n \log n)$. Note that we do *not* require the contour to be connected, i.e., our approach works even if there are holes in the arrangement.

As shown in Figure 1, we shrink the area of the chip and simultaneously blow up the existing modules by half the width and half the height of the new module. Then placing the new module m reduces to finding free space for a point.

In general finding the contour of a set of axis-aligned rectangles can be done by using the CUR algorithm as described in [6]. Our algorithm is a modification of CUR and returns a linear number of vertical and horizontal line segments. The building blocks of CUR are an algorithmic technique from computational geometry called *plane sweep* and a data structure called *segment tree*. See [4] for in-depth introductions.

The crucial part of our algorithm are two plane sweeps: one horizontal sweep that discovers all the vertical contour segments and one vertical sweep that finds all horizontal segments.

For the horizontal sweep we add for each of the modules in M' $(x'_i, Open, y'_i, y'_i{+}h'_i)$ and $(x'_i + w'_i, Close, y'_i, y'_i + h'_i)$ to a list L. After sorting this list lexicographically all elements are processed. In case of an $Open$ event the corresponding contour points are

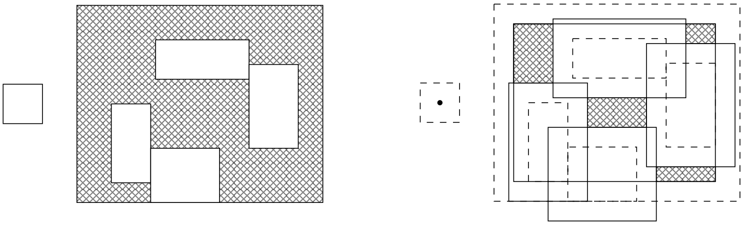


Fig. 1. (Left) A set of existing modules, and an additional module. (Right) Expanding existing modules and shrinking chip area and the new module reduces free-space management to placing a single point.

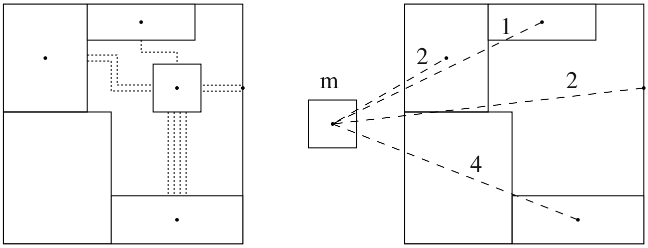


Fig. 2. (Left) Physical chip (Right) Communication model with $k = 4$: The numbers on the connections are the b_{im}.

retrieved from the segment tree and the segment $[y'_j, y'_j + h'_j]$ is added to the tree. For a $Close$ event the segment $[y'_i, y'_i + h'_i]$ is removed from the tree and the corresponding contour points are retrieved.

In the CUR algorithm we would construct the horizontal contour segments from the vertical segments. In our setting we would not find free space of height 0. So we need to do another vertical plane sweep to discover all horizontal segments.

In the algebraic tree model of computation, there is a lower bound of $\Omega(n \log n)$ on the complexity of deciding the maximum size of a free rectangle between n existing rectangles. In summary, we get the following result:

Theorem 1. *The complexity of* FINDCONTOURSEGMENTS *is* $\Theta(n \log n)$.

3 Routing-Conscious Placement

An appropriate measure for the cost of communication between modules is their Manhattan distance, weighted by the relative amount of communication. See Figure 2. Finding an optimal feasible placement under this measure can still be achieved in time $\Theta(n \log n)$, making use of local optimality properties, our OSM, and another application of plane sweep techniques.

When placing an additional module m at position (x, y), with existing modules placed at (x'_i, y'_i) and buswidth b_{im} for the communication path needed to create a routing unit between modules i and m, we consider the objective function

$$\min\{\sum_{i=1}^{k} b_{im}\|(x'_i, y'_i) - (x, y)\|_1 : (x'_m, y'_m) \in F' \setminus \bigcup_{m'_i \in M'} m'_i\}.$$

In the Manhattan metric, this can be reformulated to

$$c(x', y') = \sum_{i=1}^{k} b_{im}|x_i - x'| + \sum_{i=1}^{k} b_{im}|y_i - y'|.$$

This means we may consider two separate minimization problems, one for each coordinate. If we ignore feasibility, both minima are attained in the respective weighted medians. As we already sort the coordinates for performing plane sweeps, the running time for this step is not critical. If the median is in the occupied space there are only two other types of points where the global optimum could be located.

One type of point can be found by intersecting the contour of the occupied space with the median axes $l_x = \{(x_{med}, y) : y \in [0, H]\}$ and $l_y = \{(x, y_{med}) : x \in [0, W]\}$. In these points the x- or y-coordinate of the gradient vanishes. We cannot move in the direction of a better solution because that way is blocked by either a vertical or a horizontal segment of the contour.

The other type of points are some of the vertices of the contour. These points are the intersections of horizontal and vertical segments forming an interior angle of $\frac{\pi}{2}$ pointing in the direction of the median. In these points neither of the gradients vanishes, but local improvement is blocked by contour segments.

By inspecting all $O(n)$ local optima one finds the global optimum. Using incremental plane sweep techniques, evaluation of all local optima can be achieved in time $O(n \log n)$. Again, see [2] for details.

As the lower bound on feasible placement still applies, we get the following:

Theorem 2. *A feasible position with minimum communication cost can be computed in time $\Theta(n \log n)$.*

4 Experimental Results

The running time of our algorithm is not only good in theory, but also quite practical (as constants are small) and easy to implement. Here we show some results of our implementation. See Table 1 for an overview and [2] for details.

We have randomly generated different kinds of benchmark instances with 100 modules. The instances differ in module size and distribution of the sizes. We benchmarked a g++ 3.2 compiled c++ implementation of our algorithm against the algorithms described in [1] and [3]. Shown in the first set of columns in the table is a comparison of overall running times for 100 modules for each instance in milliseconds on a 2.53GHz Intel Pentium 4. Remarkably, our algorithm has clearly the fastest running times, even though it computes a much better solution. This illustrates the superiority of a plane-sweep apporach. Clearly, the difference in running times will increase for even larger instances. The second set of columns compares the average routing cost per module. Note that in

Table 1. Experimental results for the different benchmark instances. Overall running time, average routing cost for each module, and rejection rate are shown for the different algorithms. RCP denotes the algorithm described in this paper, NAOP refers to the algorithm as described in [1] and KFF is the algorithm KAMER combined with First Fit as presented in [3].

		Running Time (ms)			Routing Cost			Rejection Rate		
		RCP	NAOP	KFF	RCP	NAOP	KFF	RCP	NAOP	KFF
Uniform	5-10%	173	197	204	1403	3641	9522	0%	0%	0%
Uniform	10-15%	162	208	194	1747	5490	14311	0%	1%	0%
Uniform	15-20%	160	172	158	2044	7250	19791	2%	5%	1%
Uniform	20-25%	156	181	161	1987	7061	20159	10%	12%	9%
Uniform	5-25%	168	224	215	1721	6741	21347	5%	8%	5%
Increasing	5-25%	196	252	243	1931	6914	21910	8%	14%	6%
Decreasing	25-5%	175	232	228	611	2311	11712	0%	3%	4%

[1], placement is done according to a weighted Euclidean distance, and optimization is only done heuristically. As a consequence, the objective values are markedly higher. [3] does not take routing cost into account and places by some bin-packing like heuristic that tries to minimize rejection rate. As a result, communication cost is one order of magnitude higher than for our method. The third set of columns compares the average number of modules that had to be rejected due to lack of space on the chip, which is one of the objectives in [3]. Even though this figure is not considered by our algorithm, the total number of rejected modules for our algorithm is precisely the same as for [3]. Again, our results dominate the ones for [1] by a clear margin.

In summary, our algorithm is faster, better and more robust against rejection than the method described in [1]. It is also faster, much better and as robust against rejection as the approach described in [3].

References

1. A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich. A new approach for on-line placement on reconfigurable devices. In *Proc. IPDPS-2004, RAW-2004, IEEE-CS Press*, 2004.
2. A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. van der Veen. Optimal routing-conscious placement for reconfigurable computing. Manuscript (submitted), 2004.
3. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *In IEEE Design and Test - Special Issue on Reconfigurable Computing*, January-March:68–83, 2000.
4. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
5. J. Mache and V. Lo. The effects of dispersal on message-passing contention in processor allocation strategies. In *Proc. Third Joint Conference on Information Sciences, Sessions on Parallel and Distributed Processing*, volume 3, pages 223–226, 1997.
6. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.
7. H. Walder, C. Steiger, and M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Proc. IPDPS-2003, RAW-2003, IEEE-CS Press*, page 178.

Optimizing the Performance of the Simulated Annealing Based Placement Algorithms for Island-Style FPGAs

Alexander Danilin and Sergei Sawitzki

Philips Research Laboratories
Prof. Holstlaan 4
5656AA Eindhoven, The Netherlands
{Alexander.Danilin,Sergei.Sawitzki}@philips.com

Abstract. This work introduces three improvements to the traditional simulated annealing algorithm, which is widely used in industrial and academic tools for FPGA placement. The improved algorithm has been tested with the 20 largest benchmarks from the MCNC set and the results were compared with the VPR placer. The outcome is nearly 3% better timing and about 6% less swap moves in the kernel of the simulated annealing algorithm. The main positive result is the reduction of the run time of the simulated annealing algorithm without sacrificing the placement quality for combined routability and timing driven case. For most benchmarks, the quality of the final placement could even be improved despite using less swap moves.

1 Introduction and Previous Work

Simulated annealing has been introduced in the early sixties and since then became a very popular approach to solution of many optimization problems. In particular, placing algorithms for FPGA and ASIC design are often using it to minimize the timing or routing cost of the netlists. The basic idea is to minimize the cost function (defined in terms of timing and routability) of the placement by applying random swap moves to pairs of blocks in the netlist. Since the algorithmic kernel of the simulated annealing works with different cost parameters building a cost function and a schedule, much research effort was spent on their optimization. Many papers explore the dependence between run-time and placement quality. One of the latest implementations, considered state-of-the-art by the research community, was introduced with the VPR tool [1]. The novel approach, introduced in this paper does not try to trade quality for run-time, but rather to optimize some internal settings of the simulated annealing.

The rest of this paper is organized as follows. Section 2 describes the traditional simulated annealing algorithm. In section 3 the proposed improvements are described in detail. Section 4 introduces the experimental set-up and discusses the results of the benchmarking. Section 5, finally, summarizes the highlighs and draws some conclusions.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 852–856, 2004.

2 Simulated Annealing in a Nutshell

To explain the proposed improvements of the simulated annealing, a sketch of the algorithm will be provided first, based on [1]. The annealing schedule determines the speed of temperature decrease depending on the fraction of accepted moves. Cost function *Cost* is checked after every move to determine its acceptance. The placement is started with a random seed and the temperature T is set to an initially high value. In each step, two blocks or I/O pads are swapped pair wise or moved to an empty space. If the swap decreases the cost function, it is always accepted, otherwise the expression $F = e^{-\Delta Cost/T}$ is evaluated and compared to some random number μ in the range from 0 to 1 (called probability of acceptance of bad move). If F is greater than μ, the move is accepted, otherwise it is rejected (μ is regenerated for every new swap). The cost function for routability driven placement is called linear congestion and is computed as follows:

$$Wiring\_Cost = \sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)} + \frac{bb_y(i)}{C_{av,y}(i)} \right], \qquad (1)$$

where N_{nets} is the number of nets, bb_x and bb_y are the horizontal and vertical dimensions of the bounding box of the corresponding net respectively. $C_{av,x}$ and $C_{av,y}$ denote the average horizontal and vertical routing channel capacities in tracks and q is a compensating coefficient for too optimistic wire lengths of the bounding box model for nets with more than 3 terminals [2]. For timing driven placements an additional cost function is defined as $Timing\_Cost = \sum_{\forall i,j \subset circuit} Delay(i,j) \cdot Criticality(i,j)^{\eta}$, where $Delay(i,j)$ is a delay between nodes i and j, η is an empirical measure to weight critical connections more heavily and $Criticality(i,j)$ is defined as $Criticality(i,j) = 1 - \frac{Slack(i,j)}{D_{max}}$, whereby $Slack(i,j)$ denotes the time slack between the nodes i and j and D_{max} stands for the maximum delay in the netlist (= critical path). Taking both routability and timing into account, the change of the cost function for every move can then be defined as

$$\Delta Cost = (1-\varphi) \cdot \frac{\Delta Timing\_Cost}{Previous\_Timing\_Cost} + \varphi \cdot \frac{\Delta Wiring\_Cost}{Previous\_Wiring\_Cost}. \qquad (2)$$

In the classic approach φ is set to 0.5. The new temperature is computed according to the equation $T_{new} = \gamma \cdot T_{old}$. The value of γ is determined according to the schedule published in [1]. A fraction of accepted moves α is used to control the R_{limit} value, limiting the maximal distance between the blocks, considered for the swap. Initially, R_{limit} is set to the entire array, but with every temperature decrease it is recomputed according to the equation $R_{limit}^{new} = R_{limit}^{old} \cdot (1 - 0.44 + \alpha)$ and normalized to fit the range $1 \dots Array\_size$. The algorithm is terminated when

$$T < \varepsilon \cdot \frac{Cost}{N_{nets}}, \qquad (3)$$

whereby ε should be chosen reasonably small (0.005 by default). The number of swap attempts per temperature can be chosen flexibly, $10 \cdot (N_{blocks}^{1.33})$ was shown to be good enough for high quality placements [1].

3 Improving Simulated Annealing

Observation of the behavior of the algorithm for the MCNC benchmark set in VPR shows, that some slight changes to these settings may improve the performance. The first point to address is the wiring cost function. After introducing the parameters $S_x = C_{av,x} \cdot Array\_size_x$, and $S_y = C_{av,y} \cdot Array\_size_y$, where $Array\_size_x$ and $Array\_size_y$ are respectively the horizontal and vertical dimensions of the array, the equation (1) is redefined as

$$Wiring\_Cost = \sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{S_x(i)} + \frac{bb_y(i)}{S_y(i)} \right], \tag{4}$$

With this new wiring cost function $\Delta Cost$ and, thus, also the corresponding temperature values are changed. Implementation of this adopted cost function leads directly to the second improvement of the algorithm.

As described above, the acceptance probability for the bad move μ is generated randomly for every swap in the range from 0 to 1. However, using the wiring cost function defined above has a consequence, that most of $\Delta Cost$ values are falling into the range of 0.9 to 1. Setting μ to this range reduces the general acceptability of extremely bad moves, but as experiments have shown, does not influence the quality of the final placement. By applying this setting together with the redefined wiring cost function, the anneal termination condition (3) is reached faster than in [4]. Basically, the number of temperature iterations and thus the total number of moves is reduced.

Finally, the most powerful improvement is the weighting of the cost function by adjustment of the φ parameter. φ is defined as

$$\varphi = \frac{2 \cdot R_{limit}}{Array\_size_x + Array\_size_y}. \tag{5}$$

The φ value is forced to be 0.3 if it is less than 0.3 and 0.45 if it is greater than 0.45 (in general it decreases with falling temperature). At higher temperatures, the wiring part of the cost function dominates, with decreasing temperatures the timing part becomes more important. The reasoning behind this approach is based on the observation, that routability is most significantly improved, when comparatively long-distance swaps are still possible (at high temperatures). Short distance moves which dominate at lower temperatures, appear more critical for the timing. Consequently, at higher temperatures the fluctuation of the timing cost function from very good to very bad values is quite high and the timing cost is less feasible as measure for the overal quality. With lowering temperatures, *Timing_Cost* becomes more stable and should get more weight. In counterpart to the pure routability-driven placement or equal weighting of wiring and timing components, the schedule described above shows better performance. This also corresponds to the fact, that the most effective phase of the simulated annealing takes place, when α is close to 0.44 as was experimentally shown in [3].

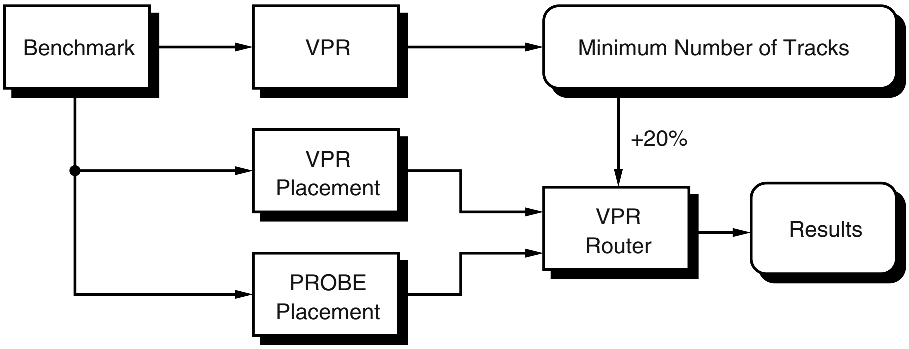


Fig. 1. Experimental set-up and design flow

4 Experimental Set-up and Benchmarking

To prove the effectiveness of the proposed improvements, the modified simulated annealing algorithm has been implemented within PROBE (placement and routing object-oriented editor) tool. The general experiment set-up is depicted in Fig. 1. The benchmarks are 20 largest circuits from the MCNC set. As proposed in [4], the relaxed case is used for the benchmarking. The target architecture is using logic blocks (LBs) consisting of one 4-input LUT and one register.

Table 1 summarizes the final results. Concerning the number of moves in the simulated annealing inner loop, 15 out of 20 benchmarks perform better. For the `des` netlist, the number of moves could be reduced by more than $\frac{1}{3}$. However, on average the improvement is only 6.2% due to dramatic increase (39.4%) of moves for `s298` benchmark and four other circuits. The timing for the relaxed case is improved by 2.9% on average. For 12 benchmarks, the critical path was decreased by 1.2-33.6%. Other 8 benchmarks showed an increase of the critical path by 0.6-11.4%. Concerning the routability, the optimized algorithm uses the same amount of tracks per routing channel with the exception of `dsip` benchmark, for which PROBE placement does not fit within relaxed case. On average, in terms of routability PROBE is only neglectingly worse even if the minimal number of tracks instead of the relaxed case is considered.

5 Conclusions

This paper introduced three improvements to the simulated annealing algorithm concerning the probability of the acceptance of the bad move, calculation of the wiring cost function, and weighting the overall cost function depending on the temperature. Tests with 20 circuits from the MCNC set shows, that the number of moves in the simulated annealing inner loop could be reduced by more than 6% (geometric average). The critical path could be reduced by nearly 3%. Even if it seems to be a rather marginal improvement, for the case of FPGA circuit design with tight timing constraints saving 3% may have significant influence,

Table 1. Results summary. All numbers are given for the relaxed case, "less is better"

Circuit (# of LBs)	total # of moves $\cdot 10^6$			critical path (ns)			# of routing tracks		
	VPR	PROBE	Δ in %	VPR	PROBE	Δ in %	VPR	PROBE	Δ
ex5p(1064)	1.23	1.14	−7.7	68.7	67.0	−2.5	19	19	0
apex4(1262)	1.49	1.28	−14.0	72.7	71.9	−1.2	18	18	0
dsip(1370)	2.75	1.97	−28.2	74.5	49.5	−33.6	8	9	1
misex3(1397)	1.62	1.55	−4.5	69.5	64.7	−7.0	16	16	0
tseng(1407)	1.32	1.28	−2.7	55.8	61.1	9.5	10	10	0
diffeq(1497)	1.81	1.80	−0.9	60.9	65.3	7.3	12	12	0
alu4(1522)	1.87	1.82	−2.6	81.6	82.9	1.6	14	14	0
des(1591)	3.02	1.98	−34.6	73.7	77.4	5.1	11	11	0
bigkey(1707)	3.27	2.63	−19.6	55.5	46.5	−16.2	12	12	0
seq(1750)	2.32	2.27	−2.2	74.2	72.0	−2.9	17	17	0
apex2(1878)	2.48	2.54	2.5	87.9	88.4	0.6	16	16	0
s298(1931)	2.48	3.46	39.4	132.4	130.3	−1.5	13	13	0
frisc(3556)	6.10	6.98	14.4	128.4	129.5	1.0	20	20	0
elliptic(3604)	6.75	5.85	−13.4	108.7	102.5	−5.7	17	17	0
spla(3690)	6.24	6.03	−3.4	106.3	118.4	11.4	19	19	0
pdc(4575)	8.41	7.57	−10.0	147.7	127.2	−13.9	25	25	0
ex1010(4598)	8.23	7.33	−10.9	155.1	151.4	−2.4	16	16	0
s38417(6406)	12.22	13.14	7.5	88.6	86.5	−2.4	11	11	0
s38584.1(6447)	13.49	13.97	3.6	83.7	90.9	8.5	12	12	0
clma(8383)	19.33	16.88	−12.6	167.7	164.9	−1.7	17	17	0
Geom.Av.	3.74	3.51	−6.2	89.4	86.9	−2.9	14.6	14.7	0.1
Total (59635)	106.50	101.47	−5.0	1893.6	1848.3	−2.4	303	304	1

so the approach introduced in this paper appears feasible after all. The main positive outcome of this work is the fact, that the run time of the simulated annealing algorithm was reduced without sacrificing the placement quality for combined routability and timing driven case. For most benchmarks, the quality of the final placement could even be improved despite using less swap moves.

References

1. Betz, V., Rose, J.: VPR: A New Packing, Placement and Routing Tool for FPGA Research. In: Field-Programmable Logic and Applications/7th International Workshop (FPL'1997), London, UK, Proceedings, LNCS Vol. 1304, Springer (1997), pp. 213-222
2. Cheng, C.E.: RISA: Accurate and Efficient Placement Routability Modeling. In: Proceedings of the 31st Design Automation Conference, (1994), pp. 690-695
3. Lam, J., Delosme, J.: Performance of the New Annealing Schedule. In: Proceedings of the 35th Design Automation Conference, (1998), pp. 306–311
4. Marquardt, A., Betz, V., Rose, J.: Timing-Driven Placement for FPGAs. In Proceedings of the 8th ACM International Conference on Field-Programmable Gate-Arrays (FPGA'2000), Monterey, California (2000)

Automating the Layout of Reconfigurable Subsystems via Template Reduction

Shawn Phillips, Akshay Sharma, and Scott Hauck

Department of Electrical Engineering
University of Washington, Seattle, WA
{phillips, akshay, hauck}@ee.washington.edu

Abstract. When designing SoCs, a unique opportunity exists to generate custom FPGA architectures that are specific to the application domain in which the device will be used. The inclusion of such devices provides an efficient compromise between the flexibility of software and the performance of hardware, while at the same time allowing for post-fabrication modification of circuits. To automate the layout of reconfigurable subsystems for systems-on-a-chip, we present template reduction. Template reduction enables a designer to eliminate resources from a template that are unnecessary to support the specified application domain. To facilitate this, we have created a feature rich template, from which we automatically generate application specific reconfigurable circuits. Compared to the full template, we achieve designs that are 53.4% smaller and 13.9% faster, while continuing to support the algorithms in a particular application domain.

1 Introduction

In the traditional FPGA design space there is a limit to the number and variety of FPGAs that can be supported – large NREs due to custom fabrication costs and design complexity means that only the most widely applicable devices are commercially viable. However, a unique opportunity exists in the system-on-a-chip (SoC) design space. FPGAs have a role in this design space as well, providing a region of programmability in the SoC that can be used for run-time reconfigurability, functionality improvements, multi-function SoCs, and other situations that require post-fabrication customization of a hardware subsystem. This gives rise to an interesting opportunity: since the reconfigurable logic will need to be custom fabricated along with the overall SoC, that reconfigurable logic can be optimized to the specific demands of the design.

The goal of the Totem project [1, 2, 3, 4] is to reduce the design time and effort in the creation of a custom reconfigurable architecture. The architectures that are created by Totem are based upon the applications and constraints specified by the designer. Since the custom architecture is optimized for a particular set of applications and constraints, the designs are smaller in area and perform better than a standard FPGA while retaining enough flexibility to support the specified application set, with the possibility to support applications not foreseen by the designer.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 857–861, 2004.

2 Background

2.1 RaPiD

The reconfigurable-pipelined datapath (RaPiD) [5] has been chosen as a starting point for the architectures that we will be generating. The goal of the RaPiD-I architecture is to provide performance at or above the level of a dedicated ASIC, while also retaining the flexibility that reconfigurability provides. RaPiD-I is able to achieve these goals through the use of course-grain components, such as memories, ALUs, multipliers, and pipelined data registers.

Along with coarse-grain components, the initial RaPiD-I architecture consists of a one-dimensional routing structure, instead of a standard FPGA's two-dimensional interconnect. The RaPiD-I architecture is able to take advantage of the reduction in complexity that a one-dimensional routing structure provides because all of its computational units are word-width devices. This structure has proven effective in supporting high-performance signal processing applications [5].

2.1 Totem

The goal of the Totem project is to create tools to generate domain-specific reconfigurable architectures based on designers' needs. One way the Totem project can achieve its goal is to remove as much flexibility as possible from a reconfigurable device, while still supporting the particular algorithms or domain that concerns a designer. While the gains of removing unneeded overhead are apparent, creating a custom reconfigurable architecture is a time consuming and costly endeavor; thus, another goal of the Totem project is to automate the creation of these custom architectures. The overall Totem design flow can be broken into three parts: high-level architecture generation, VLSI layout generation, and place-and-route tool generation.

The focus of this work is the automatic generation of mask layouts, which is performed by the VLSI layout generator. The layout generator will receive, as input from the high-level architecture generator, the Verilog representation of the custom circuit. We are currently investigating three possible methods of automating the layout process: standard-cell generation [1], circuit generators, and template reduction. Here we present the template reduction method.

3 Template Reduction Method

The idea behind template reduction is to start with a full-custom layout that provides a superset of the required resources, and remove those resources that are not needed by a given domain. One example of a similar approach to template reduction in industry is eASIC's FlexASIC™ [6]. Their approach enables designers to remove unneeded routing resources by the elimination of vias, creating a reconfigurable device that is similar to an anti-fuse based design. The goal of the Template Reduction Method is to not only remove unneeded routing resource, but to also remove unneeded functional units.

During template reduction, the removal of resources is done by automatically editing the layout to eliminate the transistors and wires that form the unused resources, as well as automatically replacing programmable connections with fixed connections or breaks for flexibility that is not needed. In this way we can get most of the advantage of a full custom layout, while still optimizing towards the actual intended usage of the array.

Template reduction has been broken into three tasks. The first is the creation of a feature rich macro cell, which is used as an initial template that will be reduced and compacted to form the final circuit. The second is the creation of the reduction list that identifies the resources that should be removed. The final task is the implementation of the reductions on the template, followed by the compaction of the resultant circuit.

3.1 Template Cell

Extensive profiling has been performed to create the feature rich template. This led us to the cell, called RaPiD-II, which is a more feature rich version than the original RaPiD-I cell. The increase in resources is required, since we have found that the original full-custom RaPiD-I cell does not have enough interconnect resources to handle some of the benchmarks intended for the architecture [4]. RaPiD-II addresses this issue because it has 24-buses and three bus-connectors per functional unit, compared to RaPiD-I, which has 14-buses and one bus-connector per functional unit. In addition to the increase in routing resources, the RaPiD-II cell that was chosen had to have a rich enough resource mix of functional units to support a large set of applications. Note that RaPiD-II isn't an architecture chosen just for template reduction, but instead is the RaPiD tile we believe is the best for implementation in any methodology, including full custom tiles. Template reduction will work on RaPiD-II, the original RaPiD-I, or any other premade tile that has at least enough resources to support the desired circuits.

3.2 Reduction List Generation

The next task in template reduction is the creation of the reduction list. Towards this end we have implemented a subtractive scheme that eliminates as many functional units and routing resources, collectively called "resources", as possible while placing and routing a set of netlists onto the template architecture. Individual netlists are placed using a simulated annealing approach [2], and routed using the Pathfinder algorithm [7]. Initially, all netlists in the set are individually placed and routed on the template architecture. At the end of this first run, the fraction of netlists that used each resource in the template is recorded, and a cost (referred to as *usage_cost*) is assigned to each resource based on the fraction of netlists that used the resource during the previous run.

After completion of the first run on all netlists, a second run is commenced during which the netlists in the set are individually placed and routed again on the template architecture. However, for any given netlist, the cost of using a resource during the second run is influenced by the *usage_cost* of that resource. At the end of the second run, the *usage_cost* of each resource is again adjusted in a manner identical to that at the end of the first run, and a third run is begun. Once the three runs are completed, we have a list of the resources that can be eliminated from the template architecture.

3.3 Reduction and Compaction

Once the reduction list is generated, the final task is to automatically edit the template, followed by a compaction step to reduce the template size. To reduce the template, the layouts were automatically edited within the Cadence CAD tools. To achieve the required automation Cadence SKILL [8] routines were written for each reduction that the subtractive method performs.

First among the reductions is the elimination of any unused cells (that is, complete RaPiD-II tiles). The next reduction is the elimination of any functional units in any cell that are not needed. Next, we remove any of the bidirectional bus-connectors that are not needed in the interconnect. The final reduction is the removal of any unused wires. When an unused wire is removed, the corresponding transistors and programming bits in any muxes and drivers on the wire are also removed. Once all of the reductions have been preformed the final design is compacted.

4 Results on Benchmarks

We are using five sets of netlist to evaluate the template reduction method. All of the netlist sets have been compiled using the RaPiD compiler [9]. The five benchmark sets are:

- Radar – used to observe the atmosphere using FM signals
- Image Processing – a minimal image processing library
- FIR –six different FIR filters, two of which time-multiplex use of multipliers
- Matrix Multiply – five different matrix multipliers
- Sorters – two 1D sorting netlists and two 2D sorting netlists

The template reduction method is able to reduce the number of functional units by an average of 45%, and the routing resources by an average of 75%. Through these reductions, we have found that the template reduction method produces circuits that are on average 53.4% smaller and 13.9% faster than the unreduced template.

5 Conclusions

With the advent of SoCs, it is now possible to reduce the NRE cost of creating custom reconfigurable devices. This presents some interesting possibilities for high performance reconfigurable circuits that are targeted at specific application domains, instead of random logic. Automation of the design flow is required, if these new custom architectures are to be designed in a timely fashion.

The template reduction method is able to leverage full custom designs, while still removing unneeded resources. This enables it to create circuits that perform at or better than that of the initial full custom template. In this work we have shown that the automation of the layout portion of the design flow is possible using a template reduction methodology. We have created the RaPiD-II cell, since the RaPiD-I cell was not able to implement all of the circuits from the benchmark suite, circuits that it was targeted to support. We have found that the template reduction method produces circuits that are 53.4% smaller and 13.9% faster than RaPiD-II.

Acknowledgements. The authors would like to thank the RaPiD group, especially Carl Ebeling and Chris Fisher, for the RaPiD-I layout used in this research. We also are indebted to Larry McMurchie for support on the Cadence tool-suite. This work was funded in part by grants from NSF and NASA. Scott Hauck was supported in part by an NSF CAREER award and an Alfred P. Sloan Research Fellowship.

References

1. S. Phillips, Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip, M.S. Thesis, Northwestern University, Dept. of ECE, July 2001.
2. A. Sharma, Development of a Place and Route Tool for the RaPiD Architecture, M.S. Thesis, University of Washington, 2001.
3. K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", IEEE Symposium on FPGAs for Custom Computing Machines Conference, 2001.
4. K. Compton, A. Sharma, S. Phillips, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", International Conference on Field Programmable Logic and Applications, pp. 59-68, 2002.
5. C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath", 6th Annual Workshop on Field Programmable Logic and Applications, 1996.
6. Application Specific Programmable Platform using eASICore® Whitepaper Version 1.0, http://www.easic.com/technolgy/whitepapers.html, March 2004.
7. L. E. McMurchie and C. Ebeling "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", Symposium on Field-Programmable Gate Arrays, pp.111-117, 1995.
8. Cadence Design Systems, Inc., "Openbook", version 4.1, release IC 4.4.5, 1999.
9. D. C. Cronquist, P. Franklin, S.G. Berg, C. Ebeling, "Specifying and Compiling Applications for RaPiD", *IEEE Symposium on FPGAs for Custom Computing Machines* 1998.

FPGA Acceleration of Rigid Molecule Interactions[*]

Tom Van Court, Yongfeng Gu, and Martin Herbordt

Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215
{herbordt|maplegu|tvancour}@bu.edu

Abstract. Modeling of molecule interactions often uses rigid models and correlation techniques, either in early screening passes or as steps within more complex models. Even rigid models are time-consuming when applied to large models at $10^3 - 10^5$ different three-axis rotations. This paper presents an FPGA structure for performing the correlations efficiently using a systolic array for 3-D correlation and an addressing technique for low-overhead rotation of a 3-D voxel models around three axes. We find a 200× speedup in our FPGA implementation compared to the standard transform-based method.

1 Introduction

Molecule interaction modeling is simulation of chemical systems, to understand how two compounds will bind. This is important in many biological interactions including screening of drugs, fitting a relatively small drug molecule (or *ligand*) to a large biomolecule (*substrate*). The ligand is tried in all three-dimensional offsets and in all three-axis rotations, a six-dimensional search of all possible *poses* of the two molecules. The best chemical and mechanical fit between the ligand and substrate is the *docked* configuration.

One simplifying assumption [1,2] is that the two molecules are rigid and that electrostatic, solvation, and other forces [3] have scalar values at fixed distances from each molecule. Using these assumptions, the molecules and forces are digitized onto 3D voxel grids. The best fit is found using 3D correlation, repeated at many different rotations. Standard PC implementations use 3D Fourier transforms to perform the correlation efficiently. We report an FPGA-based hardware accelerator that uses direct correlation to give a 200× speedup.

2 Application Detail and Serial Reference Code

A simple model describes each voxel as interior to a molecule, in the molecule's surface region, or exterior to the molecule. When the two grids are aligned,

[*] This work was supported in part by the National Science Foundation through award 9702483 and the NIH through award RR020209-01.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 862–867, 2004.

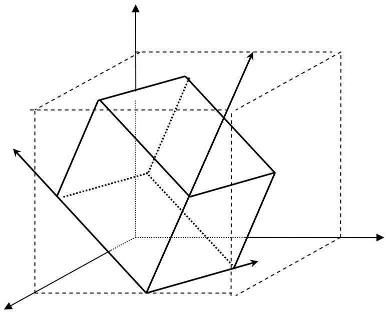


Fig. 1. Bounding box of rotated model

interior-interior overlaps represent collisions between the molecule bodies. Surface-surface overlaps represent regions where binding can occur. A few collisions need not disqualify a relative placement because real molecules flex and digitization necessarily introduces inaccuracies. Standard encoding (e.g. [3,4]) treats interior voxels as imaginary values so their products are negative. Real voxel values represent surface regions, so their products are positive. The total score $\mathcal{S}$ of one relative offset (x, y, z) between molecules A and B can be written as

$$\mathcal{S}_{x,y,z} = \textstyle\sum_i \sum_j \sum_k a_{i-x,j-y,k-z} \cdot b_{i,j,k}$$

where elements with out-of-bounds subscripts have 0 value. This is easily recognized as a 3D correlation. A change of subscript variables rewrites this as $a'_{x-i,y-j,z-k} \ = a_{i-x,j-y,k-z}$, so the correlation becomes a 3D convolution $\mathcal{S} \ = A' \star \ B$. Direct convolution takes $O(N^6)$ steps for grids of size N, which can be reduced to $O(N^3 \ \log \ N)$ using Fourier transforms:

$$\mathcal{S}_{x,y,z} = A' \star B = \mathcal{F}^{-1}(\mathcal{F}(A') \times \mathcal{F}(B))$$

One grid, assume A', is transformed and reused in all poses, so only B needs to be transformed at each rotation. The computation sequence for each rotation then consists of 1) a three-axis 3D rotation of grid B, giving grid B', 2) a 3D Fourier transform $\mathcal{F}(B')$, 3) the 3D multiplication $\mathcal{F}(A') \times \mathcal{F}(B')$,and 4) the inverse Fourier transform $\mathcal{F}^{-1}$ on the product.

As shown in Figure 1, the bounding box of B' can be up to $\sqrt{3} \approx \ 1.73$ the size of grid B, or $\sqrt{3}^3 \approx \ 5.2$ the number of voxels, though the average bounding box has only about 40% as many voxels as the worst case. The molecule is not a repeating structure, so transforms require grids A' and B' to be padded until both molecules fit without overlap. Padding and expansion in rotation do not change the polynomial complexity of the calculation, but can increase the number of voxels by a factor of 40, relative to the un-padded, un-rotated grids.

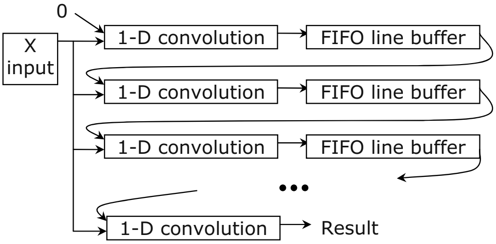


Fig. 2. Modified McWhirther/McCanny 2D convolution array

3 FPGA Algorithm, Implementation, and Results

The purpose of the Fourier transform and inverse is to avoid direct convolution. Convolution is a staple of signal processing, however, so it has been well studied and efficient FPGA structures are available for performing it. The rest of this section shows an FPGA accelerator for rigid molecule docking by direct 3D convolution. First, we present a modified McWhirther-McCanny (MM) [5] convolver that generates one $\mathcal{S}_{x,y,z}$ score value per clock cycle after startup. Next, we describe a technique for addressing the voxel model so that 3D rotation is eliminated as a separate step. Finally we demonstrate that, for realistic model sizes, direct convolution can run faster than the transform-based algorithm since it benefits from zero loop and load/store overhead, massive parallelism, and other optimizations. The net effect is an FPGA algorithm roughly 200× as fast as PC-based serial code.

Systolic array for three-dimensional convolution. Our 3D convolution is similar to the non-recursive form of the MM systolic array. The biggest change is extension of the array from its original 2D form to 3D form, shown in Figure 2. Each row in the MM array includes a 1D convolver, which holds coefficients for one row of the smaller molecule A. The row also contains a synchronous FIFO, which holds the rest of the 1D convolution result of length $x_A + x_B - 1$. Each rotation of B is broadcast to the array one voxel at a time. After a startup delay, one convolution result is output per cycle.

FIFOs (row buffers) extend the 1D convolver to result length $x_A + x_B - 1$, before sending results to the next row in that 2D plane. Likewise, the 2D plane consists of a column of y_A 1D row convolutions plus a 2D buffer connecting to the next plane in the 3D stack. The full size of the 2D result is $x_A + x_B - 1$ by $y_A + y_B - 1$, so an additional FIFO of length $(y_B - 1)\ \times (x_A + x_B - 1)$ is added to each plane. These plane buffers are also RAM FIFOs that can be programmed to handle different values of $(y_B - 1)\ \times (x_A + x_B - 1)$. The 3D array is a stack of z_A 2D planes, as shown in Figure 3.

The sizes of A and B can both vary according to the molecules modeled and the rotation of B, so the FIFO has adjustable length. Different A lengths are accommodated by using parts of the 1D convolver as FIFO elements. This allows relatively gate-intensive computation cells to hold the A values and the RAM

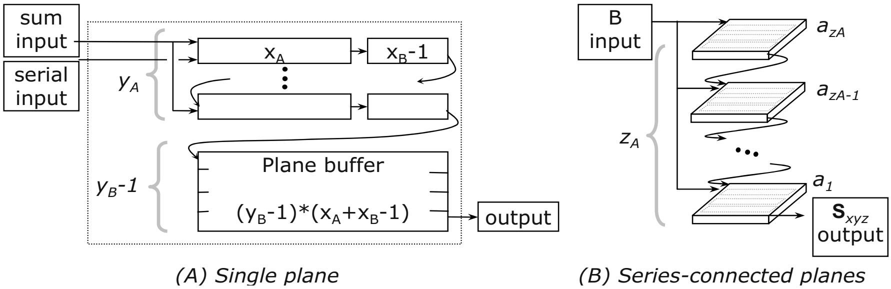


Fig. 3. Two-dimensional convolver extended to 3D

FIFO to hold the bulk result data. Since the convolution cells store the voxel values for the A grid, the number of gates available for convolution cells sets the upper bound on the size of the smaller molecule. RAM resources for the FIFOs and for the unrotated B image limit the size of the B molecule.

Each cell of the convolution array is one multiply-accumulate step of the convolution. For one-bit collision and surface information, multiplication reduces to AND operations and accumulation to conditional incrementing. We use saturating arithmetic to reduce the number of bits per accumulator without wraparound.

Fourier transform implementations must pad molecule grids because molecules are not repeating structures and because common FFT codes require 2^N grid sizes. Direct convolution does not pad for either reason, so uses far fewer voxels for each correlation.

Address generation for 3-axis rotation. Docking requires a three-axis rotation of one molecule relative to the other before each convolution. Our approach performs the inverse linear transformation, from (i, j, k) traversal coordinates to (x, y, z) unrotated coordinates, using fixed-point arithmetic precise to ± 0.5 units over the entire grid. The (x, y, z) are then converted to linear memory addresses. Because of the regular traversal order, this can be optimized to one addition and range test per axis, performed in parallel across all three axes. The regular access pattern allows even these operations to be pipelined. The result is a 3D rotation engine with:

- No separate rotation phase or buffer for rotated image,
- Precise sizing of bounding box to rotated size, averaging 40% the volume of the worst-case rotation's bounding box,
- Setup requiring only 18 values per rotation, and
- Implementation using modest amounts of logic, 268 logic slices in a Xilinx Virtex-II Pro for a 128^3 grid.

FPGA implementation and performance results. We require a computation grid that accommodates typical drug candidates: a $25 \times 25 \times 25$Å model will handle many pharmacophores of interest. The computation must also handle common substrate molecules, or at least their active regions: $200 \times 200 \times 200$Å is acceptable for our initial implementation. Resolution around 2Å is common in models of the substrate proteins [6], so is used as the size of each grid cell. The Xilinx Virtex-II Pro XC2VP100 FPGA has enough capacity for a cubical convolution array 14 cells on a side, and memory adequate for 100^3 grid cells in the substrate model and FIFOs. Thus, it meets these molecule size requirements.

We synthesized the accelerator for a Xilinx XC2VP100 FPGA using two-bit voxel values, 14^3 cells in the convolution array, buffer capacity for substrates to 100^3, and rotated addressing for access to the substrate model.

For comparison, we used only the time of a 3D Fourier transform and inverse running on a 3GHz Intel Xeon processor, and skipped the rotation and grid multiplication steps. The C-language 3D transform and inverse were taken from [7]. The C code operates only on cubes of size 2^L for integer L, so times are reported for the lowest power-of-two size that holds the result. Direct convolution in C took several times as long as the transforms, and is not reported. FPGA timings are based on a clock estimate of 24.9ns. We assume 5376 rotations, corresponding to angles sampled at 12.7° intervals.

Table 1. Performance results, FPGA vs. serial C code

Convolution	*FPGA*		*3GHz Xeon*		FPGA
size	once (ms)	5376×	once (ms)	5376×	Speedup
$100 \star 14$	35.8	3:13	-	-	-
$66 \star 14$	12.2	1:06	4,250	6:20:48	347×
rotated $66 \star 14$ *(worst case)*	52.1	4:40	4,250	6:20:48	82×
rotated $66 \star 14$ *(avg case)*	20.3	1:49	4,250	6:20:48	209×
result size 256	-	-	36,840	55:00:52	-
rotated $100 \star 14$ *(worst case)*	159.8	14:19	36,840	55:00:52	230×
rotated $100 \star 14$ *(avg case)*	73.5	6:35	36,840	55:00:52	501×

4 Extensions

We have assumed that the ligand is small relative to the substrate. In order to analyze interactions between large molecules, we take advantage of the fact that convolution is a linear operation. The resulting algorithm is analogous to the one presented here, but uses multiple passes through the convolution array. This is currently under development, as are additional scoring functions based on electostatics and other phenomena.

References

1. Chen, R., Weng, Z.: A novel shape complementarity scoring function for protein-protein docking. Proteins **51** (2003) 397–408
2. Wriggers, W., Milligan, R., McCammon, J.: Situs: A package for docking crystal structures into low resolution maps. J. Struct. Biol. **125** (1999) 185–195
3. Chen, R., Weng, Z.: Docking unbound proteins using shape complementarity, desolvation, and electrostatics. Proteins **47** (2002) 281–294
4. Katchalski-Katzir, E., Shariv, I., Eisenstein, M., Freeman, A., Aflalo, C., Vasker, I.: Molecular surface recognition: Determination of geometric fit between prtoeins and their ligands by correlation techniques. PNAS **89** (1992) 2195–2199
5. Swartzlander, E.: Systolic Signal Processing Systems. Marcel Drekker, Inc. (1987)
6. Berman, H., et al.: The protein data bank. Nucl. Acids Res. **28** (2000) 235–242
7. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press (1992)

Mapping DSP Applications to a High-Performance Reconfigurable Coarse-Grain Data-Path*

M.D. Galanis[1], G. Theodoridis[2], S. Tragoudas[3], D. Soudris[4], and C.E. Goutis[1]

[1]University of Patras,
mgalanis@vlsi.ee.upatras.gr
[2]Aristotle University,
[3]Southern Illinois University, USA
[4]Democritus University, Greece

Abstract. A high-performance reconfigurable coarse-grain data-path, part of a hybrid reconfigurable platform, is introduced. The data-path consists of coarse grain components that their flexibility and universality is shown to increase the system's performance due to significant reductions in latency. An automated methodology for mapping applications on the proposed data-path is also presented. Results on DSP benchmarks show important performance improvements, up to 44%, over existing high-performance data-paths.

1 Introduction

Coarse-grain units in hybrid reconfigurable systems improve the performance and reduce the energy consumption of applications, when these are mapped to this type of reconfigurable hardware [1]. Research in Architectural Synthesis [2] and in automatic instruction generation in Application Specific Instruction Processors (ASIPs) [3, 4], proved that complex data-path resources, called *templates*, accelerate computational intensive applications, like DSP ones. A template may be a specialized hardware unit or a group of optimally-designed chained units. Chaining [5] is the removal of the intermediate registers between the primitive units (like single ALUs) for improving the total delay of the units combined. A high performance reconfigurable coarse-grain data-path that can efficiently implement DSP applications and it is a part of a generic hybrid reconfigurable platform (Figure 1), is presented in this work. Each computational component of this data-path can easily realize any template due to its universal and flexible structure. There is a full utilization of chaining of operations through direct inter-component connections resulting in performance improvements compared with existing template-based methods [2, 3, 4].

2 Data-Path Description

The proposed data-path is composed by: (a) the Coarse-Grain Components (CGCs), (b) a register bank, and (c) a reconfigurable interconnection network, which enables

* This work was partially supported by the project IST-34793-AMDREL funded by the E.C. Also, it was partially funded by the Alexander S. Onassis Public Benefit foundation

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 868–873, 2004.

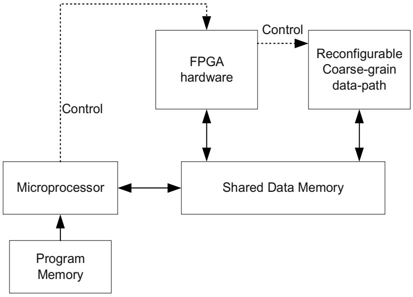


Fig. 1. Hybrid reconfigurable platform architecture

the inter-CGC connections and the connections between the CGCs and the register bank. The structure of the CGC is an *n*-by-*m* (*n*x*m*) array of nodes, where *n* and *m* are the number of nodes per row and column, respectively. In Figure 2a shows a 2x2 CGC. Each CGC node (Figure 2b) consists of two functional units that are a multiplier and an ALU, where one of them is enabled in a specific clock cycle. The data-bus of the CGC is 16-bit, since such bit-width is sufficient for the majority of the DSP applications. The fine-grain part (FPGA) of the hybrid granularity platform hosts the control-unit of the CGC data-path. A crossbar interconnection network provides full connectivity between the CGCs and among the CGCs and the register bank. A hierarchical network, like the *fat-tree* [6], can be used when a large amount of CGCs compose the data-path. The fat-tree network allows scaling of the CGC data-path, something that it is not achieved by a crossbar, since the fat-tree's interconnect delay increases logarithmically with the number *N* of CGCs, and not in quadratic manner as in a crossbar network (where $O(N^2)$ switches are required).

CGCs with a value of $2 \leq n \leq 3$ and $2 \leq m \leq 3$ are adequate to be used for improving performance of DSP applications. This is justified by the fact that in existing template-based methods [2, 3, 4], templates with *depth*=2 and $1 \leq width \leq 2$ (i.e n=2, $1 \leq m \leq 2$ for the CGC) were mainly used due to two reasons: (a) larger templates introduce larger area and control overhead relative to a primitive resource data-path [2], and (b) templates consisting of two operations in sequence contribute the most to the performance improvements [3, 4].

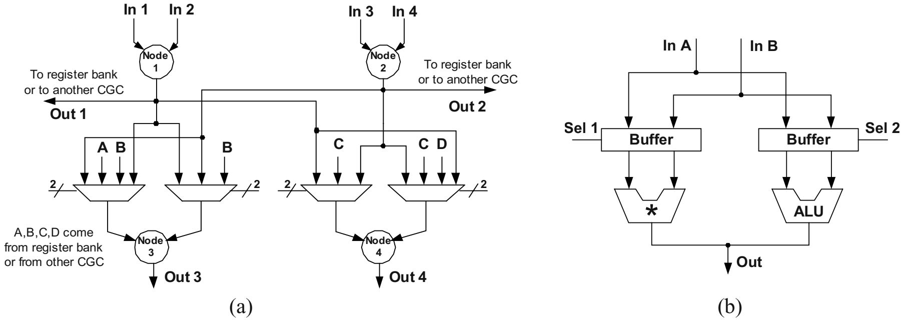

In 1
In 2
Node 1
To register bank or to another CGC
Out 1
A B
B
2
2
A,B,C,D come from register bank or from other CGC
Node 3
Out 3
In 3
In 4
Node 2
To register bank or to another CGC
Out 2
C
C D
2
2
Node 4
Out 4
(a)
In A
In B
Sel 1
Buffer
Buffer
Sel 2
*
ALU
Out
(b)

Fig. 2. Detailed architecture of a 2x2 CGC (a) and the CGC node architecture (b)

Compared with an equivalent CGC functionality realized by templates, like the ones of [2, 3] the CGC's critical path increases due to the summations of the delays of levels of tri-state buffers and multiplexers. The delay of a 2x2 (2x3) CGC compared with a template composed by two multipliers in a sequence, is increased by 4.2% (4.7%), when both are synthesized at a 0.13μm ASIC CMOS process. These delay overheads are negligible and they do not negate the performance gains (through clock cycle reduction), as these are shown in Table 1. Although extra control signals are required to configure a CGC compared to a primitive or a template resource, the control-unit can be designed in such a way that does not incur an increase to the delay of the whole data-path. This can be achieved when control signals are grouped together to define a subset of the state of the control-unit in a control-step (c-step). This way of synthesizing the control-unit is supported by current CAD synthesis tools [7], where the control-unit can be automatically synthesized under a given delay constraint. Nevertheless, the area of the control-unit increases. However, since our priority is high performance and not area consumption, this area increase is not a major consideration in this work. The area redundancy of the CGC data-path - also due to the 2 functional units in the CGC node - can be consider as a trade-off for achieving high-performance, as this is also the case in existing template-based data-paths. In [2], an average increase of 66% in area consumption was reported.

3 Mapping Methodology Description

Due to the universal and reconfigurable structure of the CGC, a full Data Flow Graph (DFG) covering using only CGCs is easily obtained, while the application mapping is simplified. Also, a full exploitation of chaining of operations both inside and among CGCs - resulting in performance improvement over primitive resource and template-based data-paths - is achieved. The existence of a library consisting of only one type of resource (i.e. the CGC) further simplifies binding with the CGCs. On the other hand, to cover a DFG by a template-based data-path, the data flow structure and the type of the operations of the DFG portion have to be matched with a template available in the library, which results in a difficult matching problem. In previous methods [2, 3, 4], due to the inflexible structure of their templates, a large number of templates is usually instantiated, as it is proven in this paper's experimental results. This prevents the design of an efficient inter-template network. Also, when partial matching [2] is not supported by the available templates as in [3], the uncovered DFG operations have to be realized by primitive resources. This may result in an increase of delay, area, and power. For example, if the primitive operations are implemented in FPGA hardware [3], the performance is degraded and this is justified by the results of Table 1.

The steps of the mapping process are: (a) scheduling of DFG operations, and (b) binding with the CGCs. The design choice was to use a list scheduler [5]. In the CGC-based data-path, the list scheduler is simplified, since it handles one resource type, which is the CGC node. Due to the features of the introduced CGC data-path a simple, though effective, algorithm is used to perform binding. The pseudo-code of

```
do {
 for the number of CGCs
  for (CGC_index=0; CGC_index <n; CGC_index ++)
   while (col_idx <  number of ops in a row && col_idx < number of uncovered DFG
nodes)
              map_to_CGC (DFG_node, CGC_index, col_idx)
   end while;
  end for;
 end for;
} while (the DFG is covered)
```

Fig. 3. The CGC binding algorithm

the binding algorithm is illustrated in Figure 3. The input is the scheduled DFG, where each c-step has T_{prim} duration. After binding, the overall latency of the DFG is measured in new clock cycles having period T_{CGC} that is set for having unit execution delay for the CGCs. We define a term called *CGC_index* that it is related to the new period T_{CGC} and it represents the current level of CGC's operations that bind the DFG nodes.

4 Experimental Results

A tool has been developed in C++ for mapping DFGs to the CGC data-path. The DFGs used in the experiments were obtained from representative benchmarks described in VHDL and C. For the extraction of the DFGs from the kernel's code, tools like the SUIF2 compiler, were used. An *experiment* was performed that showed that a data-path with two 2x2 CGCs achieves an average decrease in clock cycles (latency) of 58.1%, when compared with a data-path composed by primitive resources and its clock cycle is set to the ALU delay. The performance of the proposed data-path is higher if $T_{CGC}<2.4 \cdot T_{prim}$ (1), where T_{CGC} and T_{prim} is the clock period for the CGC and the primitive resource based data-path, respectively. Equation (1) has been experimentally satisfied after comparing the 2x2 CGC delay with a 16-bit ALU delay, both implemented in structural VHDL and synthesized with a 0.13μm CMOS process. The experiment has showed that $T_{CGC}=2.14 \cdot T_{prim}$.

A *second experiment* compares the performance of the CGC data-path with a template-based one. The template library consists of *multiply-multiply, multiply-alu, alu-alu,* and *alu-multiply* templates that enable partial matching [2]. These templates are chosen because they are proposed by the majority of the existing methods [2, 3, 4] to be used to derive high-performance data-paths for DSP applications. The clock period for the template-based and the CGC data-path is set to the combinational delay of two multiplications in sequence, so as the clock cycle (performance) comparison is straightforward. The binding with the templates is performed so as the available primitive computational resources (multipliers and/or ALUs) in each c-step is equal with the ones in the CGC data-path. The CGC data-path achieves better performance than the template-based one, due to the *average reduction* in clock cycles by 20.4% for the case of two 2x2 CGCs comprising the data-path (i.e. 4 operations can be executed in parallel), while for the two 2x3 CGCs the reduction is 20.7%. In this experiment, the number of the template instances for the template-based data-path has been also enumerated. It has been found that, in average approximately 7 (10) template instances are required when 4 (6) operations can be executed concurrently. On the other hand, 2 CGCs instances are required in both cases. So, due the absence of flexible templates (like the CGCs) the generated template-based data-paths are realized by a large number of template instances preventing the adoption of direct inter-template connections and thus the inter-template chaining exploitation.

In a *third experiment* (its results are shown in Table 1) the performance of the CGC data-path with a template-based one is compared, when the template partial matching is *not* enabled. The uncovered DFG nodes are implemented by primitive resources implemented in FPGA technology, like in [3]. The template library of the previous experiment is used. The clock period of the ASIC components (i.e. the templates) T_{ASIC} is set to the delay of the *multiply-multiply* template. In the case of the FPGA hardware, the clock period T_{FPGA} is set to the delay of the *multiplier* unit. In this experiment we assume that $T_{FPGA}=2 \cdot T_{ASIC}$, which is a reasonable assumption for the performance gain of an ASIC technology compared to an FPGA one. To simplify the synchronization problems between the FPGA and the ASIC hardware (due to the presence of two clocks), it is assumed that: (a) a closely coupled template data-path and FPGA hardware, and (b) a clock period set to T_{ASIC}. So, the DFG operations mapped to the FPGA hardware have an execution delay of 2 clock cycles, and the ones mapped to the template data-path have unit-execution delay. As deduced from the results of Table 1, the latency reduction (thus performance improvement) is even greater (approximately 43%) when the CGC data-path is compared with a template-based one that does not support partial matching.

Table 1. Latency (clock cycles) results when the CGC data-path is compared with a template-based one (template full matching + FPGA primitive resources are enabled)

DFG	**Template-based**		**CGC-based**		**% latency decrease**	
	4 res.	6 res.	two 2x2	two 2x3	4 res.	6 res.
ellip	12	12	6	6	50.0	50.0
fir11	11	9	6	6	45.4	33.3
nc	17	12	10	7	41.2	41.7
volterra	10	10	6	6	40.0	40.0
wavelet	19	14	9	7	52.6	50.0
wdf7	13	13	7	7	46.2	46.2
jpeg	134	91	81	54	39.5	40.6
ofdm	55	37	34	23	38.2	37.8
gsm_enc	196	103	103	69	47.5	33.0
gsm_dec	155	130	109	73	29.8	43.8
mpeg2	20	14	9	6	55.0	57.1
rasta	33	21	19	13	42.4	38.1
			Average values		**44.0**	**42.6**

5 Conclusions

A high-performance reconfigurable coarse-grain data-path, part of a hybrid reconfigurable platform, has been presented in this paper. An automated methodology for mapping applications to this data-path was also developed. Important performance gains have been achieved compared with primitive and template-based data-paths.

References

1. Hartenstein, R.: A Decade of Reconfigurable Computing: A Visionary Retrospective. Proc. of Design and Test in Europe (DATE) (2001) 642-649
2. Corazao, M. R., et al.: Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis. IEEE Trans. on Computer Aided Design, vol.15, no. 2 (1996) 877-888
3. Kastner, R., et al.: Instruction Generation for Hybrid Reconfigurable Systems. ACM Trans. on Design Automation of Embedded Systems, vol. 7, no.4 (2002) 605-627
4. Cong, J., et al.: Application-Specific Instruction Generation for Configurable Processor Architectures. Proc. of the ACM Int. Symposium on FPGA (2004) 183-189
5. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill (1994)
6. Leiserson, C.E.: Fat-Trees: Universal Networks for Hardware Efficient Supercomputing. IEEE Transactions on Computers, vol. 43, no. 10 (1985) 892-901
7. Synopsys Design Compiler©: www.synopsys.com (2004)

Exploring Potential Benefits of 3D FPGA Integration

Cristinel Ababei, Pongstorn Maidee, and Kia Bazargan

200 Union St. SE, ECE Department, University of Minnesota, Minneapolis MN 55455
{ababei, pongstor, kia}@ece.umn.edu

Abstract. A new timing-driven partitioning-based placement tool for 3D FPGA integration is presented. The circuit is first divided into layers with limited number of inter-layer vias, and then placement is performed on individual layers, while minimizing the delay of critical paths. We use our tool as a platform for exploring potential benefits in terms of delay and wire-length that 3D technologies can offer for FPGA fabrics. We show that 3D integration results in wire-length reduction for FPGA designs. Our empirical analysis shows that wire-length can be reduced by up to 50% using ten layers. Delay reductions are estimated to be more than 30% if multi-segment lengths are employed between layers.

1 Introduction

In response to mounting problems of the integrated circuit technology, various research groups have shown renewed interest in 3D IC integration, and a number of successful projects have shown the viability of the technology [1], [2]. 3D integration can significantly reduce wire-lengths (hence circuit delay), boost yield, and can particularly be useful for FPGA fabrics. It can address problems pertaining to routing congestion, limited I/O connections, low resource utilization and long wire delays. Even though the idea of 3D integrated circuits is not new, recent technological advances have made it a viable alternative. However, there is a lack of efficient 3D CAD tools that can exploit the potential gains that 3D integration can offer.

There has been previous work on CAD tools for 3D FPGA integration. Alexander et al. proposed 3D placement and routing algorithms [3] for their architecture in [4]. An improved version of the placement algorithm appears as Spiffy, which performs placement and global routing simultaneously [5]. In the experimental methodology presented in [6], placement was performed with VPR [7] and routing was performed with a custom routing tool [8].

In this paper we present a fast placement tool for 3D FPGAs called TPR (Three dimensional Place and Route). Unlike previous works on 3D FPGA architecture and CAD tools, we investigate the effect of 3D integration on delay, in addition to wire-length. We show that wire-length alone cannot be relied on as a metric for 3D integration benefits. The main contribution of our work is as follows.

- We analyze the potential benefits, which can be obtained by 3D integration for FPGAs. More specifically, we place circuits onto 3D FPGA architectures and study the variation in circuit delay and total wire-length compared to their 2D counterparts, under different 3D architectural assumptions. The results of this

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 874–880, 2004.

- study and similar studies in future could guide researchers in designing high performance 3D FPGA fabric architectures.
- We developed a tool, which is available as source and executable on the web. Its purpose is to serve the research community in predicting and exploring potential gains that the 3D technologies for FPGAs can offer. It shall be used as a platform, which can be used for further development and implementation of new ideas in placement and routing for 3D FPGAs.

2 Placement Algorithm

The philosophy of our tool closely follows that of its 2D counterpart, VPR [7]. The flow of the TPR placement and routing CAD tool is shown in Fig. 1. The design flow starts with a technology-mapped netlist in .blif format. Then, the .blif netlist is converted into a netlist composed of more complex logic blocks with T-VPack [9]. The .net netlist as well as the architecture description file are the inputs to the placement algorithm.

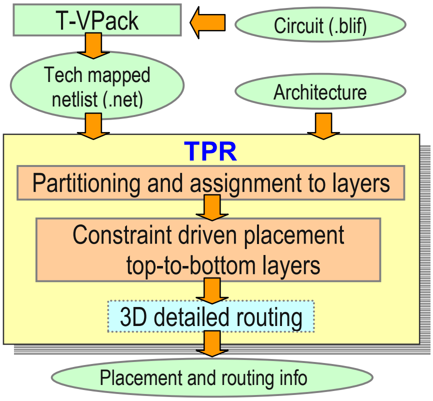

T-VPack
Circuit (.blif)
Tech mapped netlist (.net)
Architecture
TPR
Partitioning and assignment to layers
Constraint driven placement top-to-bottom layers
3D detailed routing
Placement and routing info

Fig. 1. Flow diagram of the 3D placement and routing tool

Our placement algorithm (see Fig. 2) is partitioning-based, and hence scalable in the face of explosive growth of design sizes. The initial “partitioning into layers” step is performed using the min-cut hMetis partitioning algorithm [10]. This is motivated by the limitations imposed by current technologies, which require us to minimize the usage of vertical connections (it was also observed in [11] that optimizing inter-layer interconnect is of key importance for 3D integration technologies).

After the initial partitioning into layers we assign partitions to layers using a linear placement approach. The goal of this step is not only to minimize the total vertical wire-length but also the maximum cut between any two consecutive layers. The actual placement is performed on each layer individually starting with the top layer (layer 0) and continuing downwards till the last layer (layer *L*-1). The placement of every layer is based on edge-weighted quad-partitioning using hMetis partitioning algo-

```
Input:
      Tech mapped netlist .net G(V,E)
      Architecture description file
Algorithm:
1. Initial min-cut partitioning into layers for via minimization
2. For all layers i=0 to L-1 from top to bottom
3.     Do partitioning based placement of layer i
4.         Update timing slacks
5.         Re-enumerate critical paths
6.     Greedy overlap removal
7.     Constraint generation for layers below (only for critical nets)
8. Write .p placement output file
```

Fig. 2. Pseudo-code of TPR placement algorithm

rithm. Edge weights are computed inversely proportional to the slack of nets. However, we also selectively bias weights of the most critical nets. The set of critical nets contains edges on the current k-most critical paths. The delay of the circuit (therefore slacks) and the set of the most critical paths are periodically updated based on the delay assigned to all current cut nets by the partitioning engine. This ensures accurate estimation of the circuit delay as the placement algorithm progresses. The rate of delay update and critical paths re-enumeration is dictated by runtime / estimation accuracy trade-off. The recursive partitioning of a given layer stops when each placement region has less than four blocks. Complete overlap removal is done using a greedy heuristic which moves non-critical blocks (i.e., not on any critical paths) to the closest available empty location.

When the placement of a given layer is finished, we forward propagate placement constraints for the most critical nets. In layers that have net bounding box constraints, terminals that have placement restrictions are fixed in appropriate partitions before a call to the hMetis partitioning engine. This technique explicitly minimizes the 3D bounding-boxes of critical nets, which leads to minimization of the total wire-length and circuit delay. Steps 3 to 7 of the algorithm shown in Fig. 2 are performed for all layers, and when the last layer is finished the circuit is completely placed.

3 Analysis of Placement Delay Estimation

We analyze the relation between circuit delay, which we estimate after placement and circuit delay computed after detailed routing in the 2D case. The goal of this 2D analysis is to study the reliability of the delay estimation during placement (which in turn affects slack calculations and the decision on which nets to tag as critical). To this end, we placed a set of circuits – shown in Table 1, which come with the VPR package – using our TPR placement algorithm on a single layer and recorded the total wire-length and the circuit delay using our estimation[1]. Then, all placements are routed using the timing-driven VPR routing algorithm [7], after which again circuit delay and total wire-lengths are recorded.

[1] Our optimistic delay estimation during and after placement is based on a matrix lookup table of best delays – as reported by VPR routing algorithm – which can be achieved for a route between two generic points say (x_1,y_1) and (x_2,y_2). Then, the entry in the matrix for the route delay is $(i,j) = (|x_1-x_2|,|y_1-y_2|)$.

Table 1. Statistics of simulated circuits

Circuit:	Ex5p	Apex4	Misex3	Alu4	Des	Seq	Apex2	Spla	Pdc	Ex1010
No. CLBs	1064	1262	1397	1522	1591	1750	1878	3690	4570	4598
No. IOs	71	28	28	22	501	76	42	62	56	20

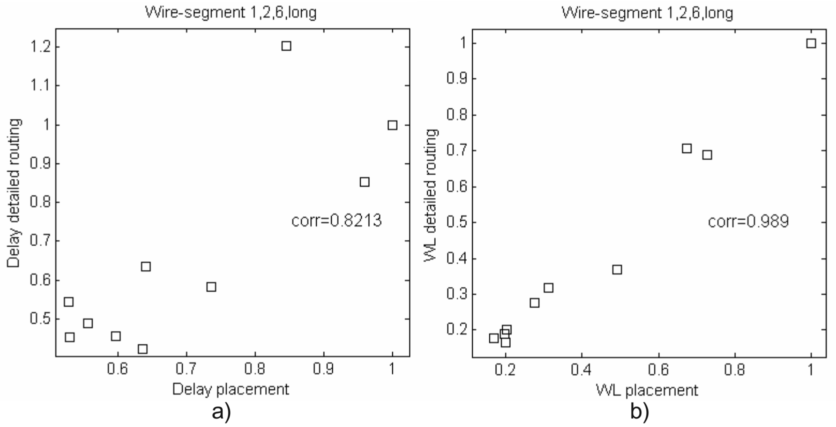


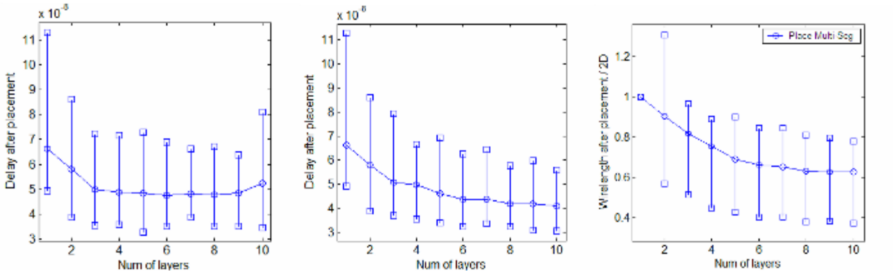
Fig. 3. Correlation between normalized delay (a) and normalized wire-length (b) after placement and after routing

Correlations between the delay and wire-length estimation after placement and the delay and wire-length after detailed routing are shown in Fig. 3.

The plot in Fig. 3.a corresponds to delay and plot in Fig.3.b shows wire-length correlations. Each point in these plots corresponds to one circuit from Table 1. A point with *(x,y)* coordinates indicates that for that circuit, delay (wire-length) estimation at placement was *x*, while the delay (wire-length) after detailed routing turned out to be *y*. The closer *x* and *y*, the more reliable the delay (wire-length) estimation is at the placement level. The plot in Fig. 3.a describes the relation between the normalized delay estimated after placement and normalized delay computed after detailed routing. The routing architecture has wire segments of length one, two and six as well as long wires. The plot in Fig. 3.b shows the correlation between the normalized total wire-length after placement (computed based on half-perimeter of bounding-boxes) and the normalized total wire-length after detailed routing. We note that there is a good correlation in all cases, which tells us that comparing placed circuits based on our estimations should be representative of the results after routing. Our 3D routing tool is yet under development. Preliminary simulations with a first version of it demonstrate plots similar to those in Fig. 3, for a number of layers greater than two.

4 Simulation Results

The goal of our simulations in this Section is to study the variation of the circuit delay and total wire-length with the number of layers when the delay of an inter-layer wire (i.e., vertical via) has different values. In the first set of simulations we assume that inter-layer vias are as long as single-length wire segments (i.e., the distance between two CLBs). This is a reasonable assumption, because 3D fabrication methods such as [1] can create inter-layer vias that are a mere 5-10μm long. The delay of the inter-layer via is assumed to be equal to the delay of a segment of length one as well. We placed circuits of Table 1 on different number of layers and recorded the average circuit delay and total wire-length of four placement runs. The simulation results are shown in Fig. 4.a and Fig. 4.c, where circle points represent the average of the ten circuits and square points represent the minimum and the maximum among the ten circuits of the delay and wire-length, respectively. It can be seen that wire-length decrease is significant (and are similar to the ASIC results from other researchers). Circuit delay decreases as well but starts increasing when the number of layers is larger than nine.



a) b) c)

Fig. 4. a) Circuit delay as a function of the number of layers: the delay of an inter-layer connection that connects terminals that are *k* layers apart is *k* times the delay of a segment of length one b) Circuit delay as a function of the number of layers when the inter-layer delay is independent of distance between layers (equal to the delay of a unit segment) c) Total wire-length – normalized to the 2D case – as a function of the number of layers when the length of an inter-layer via is one

The main reason for the lesser decrease in circuit delay (compared to the potential significant decreases one can achieve with 3D integration of standard cell integrated circuits [11], [12]) is that in FPGAs the net delay is proportional to the number of switches on the net, rather than the Manhattan distance. Therefore, the "segmentation" due to switches necessary for connecting terminals of nets spanning more layers will impair the benefits of 3D integration for FPGAs with a simplistic vertical routing architecture. We also note that the delay increase of some circuits is due to their high internal connectivity, which in turn requires a significant fraction of nets to span at least two layers.

In the second set of simulations we placed all circuits and recorded the average circuit delay for the ideal situation where the delay of an inter-layer wire is assumed to be equal to the delay of a segment of length one, regardless of how many layers separate the net terminals. This setup serves the purpose of analyzing the upper bound of maximum potential delay improvements using our method, which can be achieved by the 3D integration[2]. The simulation results are shown in Fig. 4.b. It can be seen that delay improvements of more than 30% can be achieved using nine or ten layers.

We note that the total wire-length decreases as more layers are used (see Fig. 4.c). This decrease can be up to 50% for some circuits, mainly depending on the internal connectivity of the circuits. If the length of the inter-layer via increases, then the total wire-length decrease will be less. That is mainly because the fraction of the vertical wire-length relative to the total wire-length will become significant and also the average net delay will increase due to bending (i.e., switches) of nets spanning more layers. It has to be noted that the decrease in total wire-length can have favorable impact on the routing congestion (hence channel width), as well as power dissipation (especially due to the fact that most of the power dissipated in FPGAs is due to interconnects, which account for more than 80% of the total area) as predicted by Rahman et al. in [13]. On the other hand, the potential gain in terms of circuit delay is smaller (see Fig. 4.a) unless the 3D technology offers a rich vertical routing architecture that has vias that span multiple layers without using a switch (similar to segments of length, e.g., two, six, and long in the 2D architectures). Technologically, this is doable. However, further research needs to be done to find the exact patterns and lengths of such vertical routing segments. The 2D routing architecture cannot be simply extended to the 3D due to the more restrictive limitations on the number of vertical segments.

5 Conclusion

Benefits which 3D FPGA integration can offer were analyzed using a new placement algorithm[3]. The placement algorithm is partitioning-based and has integrated techniques for minimization of the 3D bounding-boxes of nets and of the delays of the critical paths. Simulation experiments showed potential total wire-length decrease up to 50% for some circuits and 30% decrease in delay – for rich 3D vertical routing architectures which have vias that span multiple layers without using switches – compared to the 2D case.

References

1. Reif, R., Fan, A., Chen, K.-N., Das, S.: Fabrication Technologies for Three-Dimensional Integrated Circuits. Proc. International Symposium on Quality Electronic Design (2002) 33-37
2. Guarini, K.W., et al.: Electrical integrity of state-of-the-art 0.13um SOI CMOS devices and circuits transferred for three-dimensional (3D) integrated circuit (IC) fabrication. Technical Digest of the International Electron Devices Meeting (2002) 943-945

[2] We should emphasize, however, that this bound is not going to be far off from a post-routing analysis. In the 2D FPGA architectures of today, the delay of longer segments is comparable to the delay of unit segments. For example, in the Xilinx Virtex architecture, a hex segment of length six has a delay of 1.1 times the delay of the unit segment.

[3] Implementation available for download at: http://www.ece.umn.edu/users/ababei/

3. Alexander, A.J., Cohoon, J.P., Colflesh, J.L., Karro, J., Peters, J.L., Robins, G.: Placement and Routing for Three-Dimensional FPGAs. Canadian Workshop on Field-Programmable Devices (1996) 11-18
4. Alexander, A.J., Cohoon, J.P., Colflesh, J.L., Karro, J., Robins, G.: Three-Dimensional Field-Programmable Gate Arrays. Proc. International ASIC Conf. (1995) 253-256
5. Karro, J., Cohoon, J.P.: A spiffy tool for the simultaneous placement and global routing for three-dimensional field-programmable gate arrays. Great Lakes Symposium on VLSI (1999) 226-227
6. Chiricescu, S., Leeser, M., Vai, M.M.: Design and Analysis of a Dynamically Reconfigurable Three-Dimensional FPGA. IEEE Trans. VLSI Systems, Vol. 9, No. 1, (2001) 186-196
7. Betz, V., Rose, J.: VPR: A New Packing Placement and Routing Tool for FPGA Research. Field-Programmable Logic App. (1997) 213-222
8. Chiricescu, S.: Parametric Analysis of a Dynamically Reconfigurable Three-Dimensional FPGA. Ph.D. Dissertation, Northeastern University (1999)
9. Marquardt, A., Betz, V., Rose, J.: Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. Proc. International FPGA Conf. (1999) 37-46
10. Karypis, G., Aggarwal, R., Kumar, V., Shekhar, S.: Multi-level Hypergraph Partitioning: Applications in VLSI Design. Proc. ACM/IEEE DAC (1997) 526-529
11. Das, S., Chandrakasan, A., Reif, R.: Three-Dimensional Integrated Circuits: Performance Design Methodology and CAD Tools. Proc. International Symposium on VLSI (2003) 13-19
12. Das, S., et al.: Technology, performance, and computer-aided design of three-dimensional integrated circuits. Proc. ACM/IEEE ISPD (2004) 108-115
13. Rahman, A., Das, S., Chandrakasan, A., Reif, R.:Wiring Requirement and Three-Dimensional Integration of Field-Programmable Gate Arrays. Proc. ACM/IEEE SLIP (2001) 107-113

System-Level Modeling of Dynamically Reconfigurable Co-processors

Yang Qu[1], Kari Tiensyrjä[1], and Kostas Masselos[2]

[1] VTT Electronics, P.O. Box 1100 (Kaitoväylä 1), FIN-90571 Oulu, FINLAND
yang.qu@vtt.fi
[2] INTRACOM SA, Emerging Technologies and Markets Division, 19,5km Markopoulou Avenue, P.O.Box 68, GR 19002 Peania, Attika, Greece

Abstract. Dynamically reconfigurable co-processors (DRCs) are interesting design alternatives when both flexibility and performance are concerns. However, it is difficult to study the performance impact of including such devices into design when using traditional design methods and tools. In this paper, we present easily adaptable system-level techniques, which are able to perform fast exploration of different reconfiguration alternatives. A SystemC-based modeling method for DRCs and a high-level synthesis-based estimation tool to support system partitioning are presented.

1 Introduction and Related Work

The technology developments have made it possible to re-program configurable hardware at run time. Such device is generally referred to as dynamically reconfigurable logic (DRL). Unlike software or hardware implementation, DRLs spread computation over both time and space. The new feature requires various changes in the traditional design flow. At the system level, the problems are how to support HW/SW/DRL partitioning, how to evaluate different reconfiguration alternatives, how to model the DRLs with the aim of fast design space exploration, etc. In the era that the design level is moving higher and higher, the design of Reconfigurable System-on-Chip (RSoC) requires an easily adaptable solution to enhance traditional design methods and tools in order to reduce the time-to-market.

Authors in [1] proposed a VHDL modeling technique of the reconfigurable process that is simulatable and takes reconfiguration overhead into account, but the approach is not suitable for design space exploration. In [2], a system-level model of runtime reconfigurable system was proposed. However, the reconfiguration overhead was not addressed. In [3], the VCC tool was used to evaluate different design options of a reconfigurable platform, but context scheduling is not addressed.

Our research focuses on high-level design methodology of reconfigurable systems, where DRLs are used as co-processors. This paper presents a system-level modeling technique of DRCs and the associated tools. The work is an extension of [4]. The main advantage of the approach is that it can be easily embedded into a SoC design flow to allow fast design space exploration for different reconfiguration alternatives without going into implementation. The system-level model describes the behavior of

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 881–885, 2004.

the reconfiguration process and relates the performance impact of the reconfiguration process to a set of parameters extracted from reconfiguration technologies of interest. Thus, by tuning the parameters, designers can easily evaluate the trade-offs between different technologies. In simulation, the model can automatically detect the reconfiguration requests and trigger the reconfiguration process. The modeling methodology is supported by an estimation tool for the system partitioning and a transformation tool for reuse of existing SystemC code.

The structure of the paper is as following. Section 2 introduces the modeling technique and supporting tools. The validation work using a MPEG2 decoder case is described in section 3. Section 4 gives the conclusions.

2 Proposed System-Level Modeling Techniques

The important tasks in system-level design of RSoC are to identify candidate components and to reveal reconfiguration overhead. The candidate components are application functions that are considered to gain benefit from being implemented on DRCs. The decision whether a task should be a candidate component is clearly application dependent. The criterion is that the task should have two features in combination: flexibility (that would exclude an ASIC implementation) and high computational complexity (that would exclude a software implementation). Flexibility may come either from the point that the task will be upgraded in the future or in view of hardware resources sharing with other tasks with non-overlapping lifetimes for global area optimization. The reconfiguration overhead is the feature closely related to DRL technologies and run-time behavior of the candidate components.

Our modeling technique focuses on three issues: selection of candidate components, modeling of the reconfiguration overhead for fast design space exploration, and design reuse. They are separately addressed in following sections.

2.1 Estimation Approach to Support Identifying Candidate Components

We developed a high-level synthesis-based estimation tool [5], which can produce estimates of the execution time and hardware resources required for embedded FPGA type DRCs, in order to support the selection of candidate components with the aim of total area reduction. Traditional HW/SW partitioning methods will be involved when making a full HW/SW/DRL partitioning.

The input is C code of tasks to be studied. A SUIF-based front-end preprocessor is used to extract Control-Data Flow Graphs (CDFG), based on which well-known high-level synthesis tasks are carried out to produce the estimates. As-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling are used to determine the critical paths, from which we estimate the execution time. A modified version of Force-Directed Scheduling (FDS) is used to estimate the hardware resources required for the tasks. Finally, allocation algorithms are used to estimate the hardware resources required for interconnection with multiplexer type of interconnection units. The current estimator targets a Virtex2-like embedded FPGA in which main resources are LookUp-Tables (LUTs) and multipliers.

2.2 Modeling of Reconfiguration Overhead

The modeling of reconfiguration overhead is divided into two steps. In the first step, different technology-dependent features are mapped onto a set of parameters, which are the size of configuration data, the clock speed of configuration and the extra delays apart from loading of the configuration data. In the second step, a SystemC module that models the behavior of run-time reconfiguration process is created and is used in system-level simulation to reveal the reconfiguration overhead.

A general SystemC model of RSoC is shown in Fig. 1. The left side is an overview of the RSoC. The DRC is a single SystemC module, which implements the same bus interfaces in the same way as other HW/SW modules. A configuration memory is modeled, which could be an on-chip or off-chip memory that holds the configuration data. The right side shows the internal structure of the DRC, which is in fact a hierarchical SystemC module. Each candidate component (F1 to Fn) is an individual SystemC module, which implements the top-level bus interfaces with separate system address space, and is instantiated inside the DRC. Each candidate component has two extra ports. One is a DONE signal port routed to the Configuration Scheduler (CS). The port is used to acknowledge the CS that this task can be safely swapped out. The other is connected to a shared memory that saves the data to be preserved during reconfiguration. The Input Splitter (IS) is an address decoder and it manages all incoming Interface-Method-Calls (IMCs). The CS monitors the operation states of the candidate components and controls the reconfiguration process.

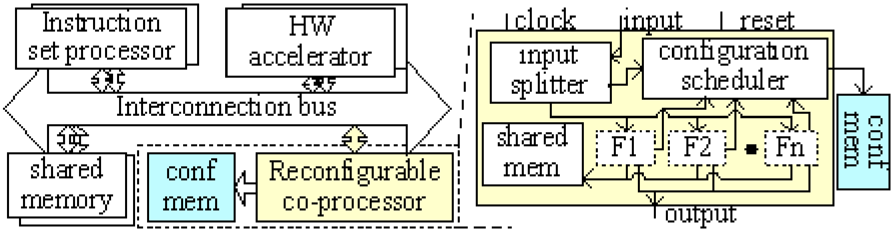


Fig. 1. System-level Modeling of Reconfigurable SoC

The main idea of the modeling method is as following. When the IS captures an IMC to a candidate component, it will hold the IMC and pass the control to the CS, which decides if reconfiguration is needed. If so, the CS will call a reconfiguration procedure that uses the parameters specified in step 1 to generate memory traffic and associated delays to mimic the reconfiguration latency. When the CS is done, the IS will dispatch the IMC to the target module. If the module cannot be activated at the moment, a message of request to reconfigure the target module will be put into a FIFO queue and the IMC will return with the value of FALSE. When a module finishes its operation, it will send a DONE signal to the CS, and the CS will check if there is any waiting message in the FIFO queue. If so and it is possible to activate the waiting module, the CS will call the reconfiguration procedure. Concerning the practical implementation effort, the pre-emption of a running module is not supported. The modeling method is for non-blocking IMCs. The use of blocking IMC requires the behavior of the system bus to be changed in order to avoid the bus being locked when the called module is off the device.

There is a state diagram common to all candidate components, based on which the CS makes reconfiguration decisions. A state diagram of partial reconfiguration is presented in Fig. 2. For single context and multi-context DRCs, similar state diagrams can be used in the model. The main advantage of the modeling method is that the rest of the system and the candidate components need not to be changed between a static approach and run-time reconfiguration approaches, which makes this method very useful in making fast design space exploration.

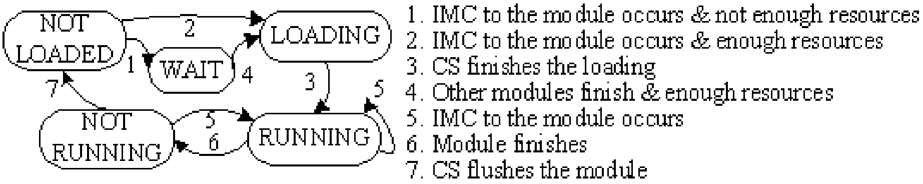

NOT LOADED
LOADING
WAIT
NOT RUNNING
RUNNING
1
2
3
4
5
6
7
1. IMC to the module occurs & not enough resources
2. IMC to the module occurs & enough resources
3. CS finishes the loading
4. Other modules finish & enough resources
5. IMC to the module occurs
6. Module finishes
7. CS flushes the module

Fig. 2. Reconfiguration state diagram

2.3 Transformation Tool to Support Reuse of Existing SystemC Modules

We developed a tool that can automatically transform SystemC modules, which however must follow a defined modeling pattern, into a SystemC module of a DRC. The inputs are SystemC files of a static architecture and a script file, which gives the names of the modules that are selected as candidate components and the associated design parameters. The outputs are SystemC files of a modified architecture, in which those specified SystemC modules have been replaced with a DRC module. The kernel of the tool contains a C++ parser to analyze the SystemC files, a script file parser and a template module of the DRC. There are two specific requirements for the input moduls. Firstly, modules should implement the bus interface methods with defined names. Otherwise the transformer would not have the knowledge of their meanings. Secondly, a port of DONE signal with specified name should exist in a candidate module in order to let the CS capture its status.

3 Case Study

A MPEG2 decoder case is chosen to prove the approach is very useful for the task of fast design space exploration. The starting point is a SystemC transaction-level model of a static architecture of the decoding system. Control-oriented tasks, such as variable-length decoding, are assigned to a RISC processor. Motion compensation is assigned to a DSP core. The color converter (CC), which processes 8 pixels in parallel, and the IDCT are assigned to two separate hardwired ASICs. A shared memory and a one-level system bus are used. The task is to study the possibility of moving the IDCT and the CC from ASIC implementation to a DRC.

The DRC is a Virtex2-like FPGA. The partial, single/multi-context reconfiguration are to be considered. Features of the target DRC are as following. There are 3200 LUTs and 40 multipliers available. The size of bitstream to configure the full device is 200k bytes. In partial reconfiguration, the size of configuration data is proportional to the number of LUTs required. In the multi-context reconfiguration, there are two layers of programming bits and 5 clock cycles are required for context switching. The configuration clock is running at 50MHz, and 8 bits are loaded every cycle.

We started with the estimation of the requirement of the configuration data in partial reconfiguration. The estimation tool showed 2983 LUTs and 2688 LUTs were required for the IDCT and the CC separately, which correspond to 186k and 168k configuration data. Three simulation packages were created using the modeling method described in section 2.2 and the simulation results are given in Table 1. The differences between three configuration styles are clearly revealed. Designers can easily make design decisions when information of ASIC area of the two functions and the estimates of design time are available.

The case study proves the approach is useful in helping designers to rapidly perform design space exploration. The estimation tool can produce results within minutes without any manual effort. In the SystemC modeling, the transformation tool can significantly reduce the amount of coding work. Designers need to edit only the script file of the design parameters, which can be easily done within a minute.

Table 1. Comparison of reconfiguration latencies

	Original	Single	Multi	Partial
Decoding time (ms/fr)	15.35	26.69	18.69	25.78
Conf. latency (ms/fr)	NA	8.00	2e-4	7,09

4 Conclusions

In this paper, we have presented a system-level modeling methodology of DRCs. The use of DRCs will create a flexible system and result in shorter time-to-market when comparing with equivalent ASIC-type SoC implementation. It is very important to have an approach that allows designers in the early phase of design to rapidly explore the differences of using different reconfiguration alternatives. Our easy-to-use approach has been proved with a MPEG2 case to be able to fulfill the task.

References

1. Robinson, D., Lysaght, P.: Methods of exploiting simulation technology for simulating the timing of dynamically reconfigurable logic. IEE Proc. Vol. 147, No. 3. (2000) 175-180
2. Rissa, T., et al.: System-level modeling and implementation technique for run-time reconfigurable systems. Proc. 10th Annual IEEE Symposium on FCCM. (2002) 295 – 296
3. Vanzago, L., et al.: Design space exploration for a wireless protocol on a reconfigurable platform. Proc. DATE (2003) 662 – 667
4. Pelkonen, A., et al.: System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. Proc. IPDPS'03 (2003) 174-181
5. Yang Qu, Soininen, J.-P.: Estimating the utilization of embedded FPGA co-processor. Proc. Euromicro Symposium on DSD. (2003) 214-221

A Development Support System for Applications That Use Dynamically Reconfigurable Hardware

João Canas Ferreira and José Silva Matos

FEUP/DEEC and INESC Porto
Rua Dr. Roberto Frias
4200-465 PORTO PORTUGAL
{jcf,jsm}@fe.up.pt

Abstract. ReDiFlex is a system that supports the development of applications that use dynamically and partially-reconfigurable hardware. The hardware functionality is specified by the flow of data between mutable operators. The system automatically creates the physical implementation after partitioning the model to fit the hardware constraints; during application execution new computation-dependent partial configurations can be created. ReDiFlex provides run-time support for reconfiguration and data transfer scheduling.

1 Introduction

The describe a development support system for applications that use dynamically reconfigurable hardware (DRH) called ReDiFlex. Embedded in an interactive programming environment, ReDiFlex allows the programmer to define the functionality of the hardware through a data-flow-like specification, maps this specification to dynamically reconfigurable hardware, generates the required configurations and enables application access the additional operations through regular function calls. Changes to the specification can be immediately and transparently propagated to hardware by partial reconfiguration. In addition, the implementation is automatically partitioned to accommodate hardware size restrictions. ReDiFlex includes a run-time support system that transparently schedules all reconfigurations and data transfers.

The prototype implementation of ReDiFlex uses a H.O.T. Works board from VCC equipped with a XC6216 FPGA [1]. The board is connected through the PCI bus to the CPU that runs the application (and the development system). One of the aims of this work was to produce a system where the use of the hardware infrastructure is transparent to the applications programmer, except for the need to specify explicitly what part of the application is to run on DRH.

After discussing some global design choices, section 2 describes the main modules of the ReDiFlex systems, with emphasis on the services provided to the user/programmer. Section 3 shows briefly how the behaviour of the DRH part is specified. Section 4 explains how the high-level description of the functionality is converted to DRH configurations. Brief concluding observations appear in sec. 5.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 886–890, 2004.

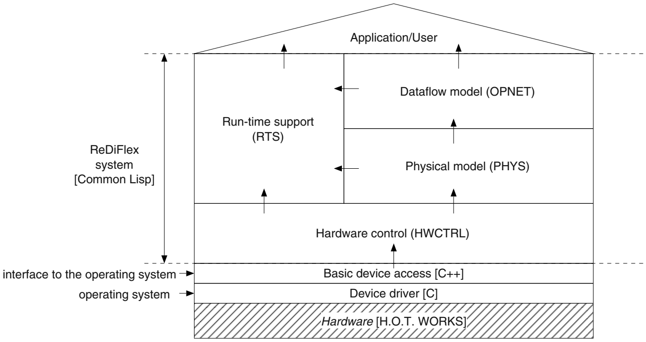


Fig. 1. ReDiFlex: main modules and general information flow

2 General Organisation of the ReDiFlex System

ReDiFlex was designed to be embedded in a general-purpose programming environment. We decided that the hardware functionality would be described explicitly by a data-flow-like model, where the processing nodes (but not the interconnection structure) might change with time. This enabled us to avoid having to perform generic run-time routing of interconnections.

The use of an explicit model implies that the hardware infrastructure is not transparent to the user. Instead, the user must have the two domains (changing hardware and software) in mind while designing the application. This is not necessarily bad, since a uniform model is difficult to find.

The data-flow model is a first-class "citizen" of the application, i.e., it is represented by an object that can be inspected and partially changed by the running application (in general, the changes may be done at run-time to account for the data that is being processed). Any changes to this representation of the model are automatically propagated to its hardware implementation by the runtime support system.

ReDiFlex needs detailed knowledge of both the computations performed by the hardware and the resources involved (including the exact position of the individual modules on the FPGA). In addition, some operations (like routing) may be partially performed at run-time (in restricted situations). Therefore, the system implements all the tasks necessary to go from the data-flow model to the individual (partial) reconfigurations (including model partitioning, placement and routing) without relying on any external tools.

ReDiFlex is a layered system, with upper layers building on those immediately below (Fig. 1). An exception is the run-time support system, which interfaces with all the other main modules.

The application programming interface (API) is provided jointly by the run-time support (RTS) system and by the module that handles data-flow models (the OPNET module). The API is embedded in the host programming environment, so all the features of the latter are available for application development.

Together, the two top modules provide the following services:

1. Incremental construction of dataflow-like models with consistency checks.
2. Automatic creation of the interface to the running hardware.
3. Scheduling of reconfiguration and data transfer operations to/from the DRH.
4. Access to detailed information on the mapping of the model to the hardware resources.

For each data-flow model, the system creates an appropriate implementation (without user intervention) and defines a set of functions that the application can use to send data to the reconfigurable hardware and to retrieve the results. To the application, these are regular functions of the host programming environment; internally they call on the reconfigurable hardware for the computations.

An additional set of functions that control hardware reconfiguration are also created. They translate changes of the data-flow model into reconfiguration operations at the hardware level. Changes are specified in terms of the data-flow model (the "visible" representation of the hardware at the application level); the application never deals directly with hardware resources.

The RTS module is responsible for the actual data transfers to/from the DRH, and for scheduling and carrying out the reconfiguration operations.

The OPNET module takes care of the internal representation of the data-flow models and associated processing (including model partitioning). The PHYS module handles the internal representation of the reconfigurable hardware resources. It is responsible for placement and routing of individual elements and performs incremental routing inside special elements called containers. A restricted type of partitioning can also be done at this level.

The HWCTRL module provides the raw access to the hardware. It handles: (i) hardware initialisation and reset; (ii) reading and writing of application or configuration data; (iii) clock control (both single-step and free-running). The basic access to the hardware is implemented by a thin layer of C++ code and a device driver.

3 Specifying the Functionality of Dynamically Reconfigurable Hardware

The specification of the DRH behaviour is based on the flow of data between operators. Each operator is characterised by the function it implements and its interface (unidirectional input/output channels). A complete description is an *acyclic* network of operators. An operator network has fixed connections, but the operators can change in well defined ways. Dynamic, partial reconfiguration is used to propagate changes of the operators to the underlying hardware.

Operators are abstract, parameterisable classes of objects. Each operator can have application-accessible state attributes. An operator's instance state attribute values are accessible for inspection while the application runs. (State attributes are a model for hardware registers.) Parameters either specify initial values of state attributes or define aspects of the hardware implementation. In general, the parameters of an operator can be changed at any time. If a parameter corresponds to a state variable, the modification takes effect immediately,

Table 1. Operator network partitioning

Circuit	Ops.	Sub-models	Time (s)
alu4	202	2	0.3
spla	581	11	0.6
des	3592	86	7.9

Table 2. Layout partitioning

Circuit	Ops.	Nets	Time (s)
cordic	80	78	6.47
inc	113	104	14.3
misex2	142	124	17.1

Running times of partitioning routines for some MCNC benchmarks (Pentium III, 730 MHz)

even if the operator is active, i.e. running on the hardware. For implementation-modifying parameters, changes are registered in the model, but only take effect when a reconfiguration is explicitly requested.

Some parameters determine aspects of the operator's interface, for instance, the number of bits of an input port. These parameters can only be specified once, before the underlying hardware configuration is produced. This follows from the decision not to support modifications of inter-operator connections at run-time.

The current prototype implementation includes a basic set of operators. Some of them are not mutable (i.e., have a fixed implementation) and correspond to simple logic gates or arithmetic circuits; others correspond to dynamically reconfigurable circuits. For instance, `adder` is a regular binary adder, `const-adder` is a circuit that adds a constant value to its input (a specialised version of `adder` for a fixed second argument) and `mutable-const-adder` is a `const-adder` that can be changed at run-time (i.e., successively specialised for different constants).

ReDiFlex also provides *containers*, mutable operators whose implementation details are delegated to other operators. The container is a "wrapper" around other operators, only one of which may be active at any one time. The "contained" operators must be compatible: they must have identical number of input and output ports and the respective sizes must be the same; the containers need not have the same physical dimensions. Containers have a special parameter that specifies the active "contained" operator. The change of active operator implies re-routing the connections to the container pins at run-time and saving the instance's state. Containers may be nested.

Feeding the FPGA with the correct data is a complicated process, because operators may be sequential circuits and it is necessary to provide them with the input data on precisely determined clock cycles. Similarly, the different results must be collected on specific clock cycles. The run-time support system takes care of these tasks automatically. The system also supports a pipelined mode of operation for processing large batches of data.

4 From Data-Flow Model to Physical Implementation

Given an operator network, ReDiFlex generates the required FPGA configurations and the scheduling information for the run-time support system. For each kind of operator, the system has a *generator*, a set of software routines that generate the configuration for a rectangular section of the FPGA (a “block”). The generators can create new configurations or alter existing ones. The external dimensions of the blocks are preserved in the process.

ReDiFlex generates the layout of the circuits by starting with a levelised graph of the data-flow model: for each level, a corresponding full-height vertical “slice” is created in the layout. Each slice contains the blocks associated with operators of the same level plus feed-through blocks that allow for communication between non-adjacent slices. This organisation allows for a direct mapping between operator model and layout and is well suited to the architecture of the XC6200 family. ReDiFlex then proceeds to route all connections between adjacent slices, adjusting the space between them as necessary.

The layout obtained by this procedure may not fit into the FPGA because it may be too wide or too high. The fist problem can solved by partitioning the physical layout in FPGA-sized clusters. Additional input and/or output registers have to be added to the layout of each cluster. At run-time, the support system stores the intermediate values produced by one cluster and feeds them to the next cluster after FPGA reconfiguration. This solution is implemented directly on the layout. To solve the second problem, ReDiFlex partitions the initial data-flow model into a set of implementable sub-models. The sub-models must form an implementable acyclic graph. This task is done by a variation of the partitioning algorithm of [2]. The current implementation stresses finding a viable layout rapidly, as suits the exploratory nature of the tool. Table 1 reports running times for model partitioning; table 2 does the same for physical-level partitioning.

5 Conclusion

ReDiFlex is a working, interactive support system for applications running on a CPU with fast connection to DRH. It was intended as an vehicle for research on execution models for DRH-based applications and associated run-time management issues. New platform FPGAs with embedded CPUs (like the Virtex-II Pro) seem particularly attractive targets for extensions of the work presented, with the PHYS layer requiring the largest modification. An important problem that remains to be solved is how to describe the functionality of the DRH at an abstract level, while being able to exploit specific resources like internal block RAMs.

References

1. Xilinx: XC6200 Field Programmable Gate Arrays. Xilinx. (1997)
2. Cardoso, J.M.P., Neto, H.: Macro-based hardware compilation of Java bytecodes into a dynamic reconfigurable computing system. In Pocek, K.L., Arnold, J.M., eds.: Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines. (1999), pp. 2–11

Interconnect-Aware Mapping of Applications to Coarse-Grain Reconfigurable Architectures*

Nikhil Bansal[1], Sumit Gupta[2], Nikil Dutt[1], Alex Nicolau[1], and Rajesh Gupta[3]

[1] University of California, Irvine,
[2] Tallwood Venture Capital, Palo Alto,
[3] University of California, San Diego

Abstract. Coarse-grain reconfigurable architectures consist of a large number of processing elements (PEs) connected together in a network. For mapping applications to such coarse-grain architectures, we present an algorithm that takes into account the number and delay of interconnects. This algorithm maps operations to PEs and data transfers to interconnects in the fabric. We explore three different cost functions that largely affect the performance of the scheduler: (a) priority of the operations, (b) affinity of operations to PEs based on past mapping decisions, and (c) connectivity between the PEs. Our results show that a priority-based operation cost function coupled with a connectivity-based PE cost function gives results that are close to the lower bounds for a range of designs.

1 Introduction

Coarse-grain reconfigurable architectures have been proposed as co-processors for accelerating compute intensive portions (generally loops) of applications in embedded system platforms. These architectures are attractive to system designers because they provide the high performance of ASICs with the ease of reconfiguration of fine-grain FPGAs. As a result, we have seen the emergence of a wide range of coarse-grain reconfigurable architectures over the last few years [1,2,4,5,6]. (to name a few). Mapping applications to coarse-grain architectures is a combination of assigning time cycles for operations to execute in (scheduling), mapping these operation executions to specific PEs (mapping), and routing the operands or input data by mapping and scheduling data communications to specific interconnects in the fabric (routing).

In this paper, we present an algorithm that performs these tasks by taking into account the spatial locality or connections between the PEs and the temporal locality between the data used by the operations. We examine different metrics that affect this algorithm – specifically (a) the *priority* of operations based on the length of the dependency chain or critical path through the code, (b) the *affinity* of operations to PEs, i.e., operations are more likely to be mapped to a PE if their predecessor operations are also mapped to that PE or one of its connected neighbors, and (c) the *connectivity* of the PEs, i.e., we first map operations to PEs that are connected to the maximum number of other PEs, thereby, exploiting their spatial locality. The main contribution of this paper is in examining the

* This work was partially supported by NSF grants CCR-0203813, ACI-0204028, and Hitachi Corporation.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 891–899, 2004.

```
/* Schedules & maps operations in basic block currBB */
ScheduleMapBB(currBB, currCycle, PEList)

$\mathcal{A}_{avail} \leftarrow$ List of all available operations of $currBB$
        in $currCycle$
while ($\mathcal{A}_{avail} \neq \phi$ )
    Calculate $C_{op}$ $\forall$ ops in $\mathcal{A}_{avail}$ and $C_{PE}$ $\forall$ PEs in $PEList$
    $\mathcal{A}_{ordered} \leftarrow$ Order $\mathcal{A}_{avail}$ by cost function $C_{op}$
    $PEList \leftarrow PEList$ ordered by cost function $C_{PE}$
    foreach ($candPE \in PEList$ with lowest cost $C_{PE}$)
        $\mathcal{A}_{candPE} \leftarrow \mathcal{A}_{ordered}$
        while ($\mathcal{A}_{candPE} \neq \phi$ AND $candPE$ not scheduled)
            Pick $candOp \in \mathcal{A}_{candPE}$ with lowest cost $C_{op}$
            $\mathcal{A}_{candPE} \leftarrow \mathcal{A}_{candPE}$ - $candOp$
            if ( IsRoutable($candOp$, $candPE$, $currCycle$) )
                $\mathcal{A}_{ordered} \leftarrow \mathcal{A}_{ordered}$ - $candOp$
                $\mathcal{A}_{avail} \leftarrow \mathcal{A}_{avail}$ - $candOp$
                Schedule $candOp$ on $candPE$ in $currCycle$
                Update route usage information in $currCycle$
    $currCycle \leftarrow currCycle + 1$
    $\mathcal{A}_{avail} += $ List of all available operations of $currBB$
            in $currCycle$
```

Fig. 1. Algorithm to schedule a basic block

```
/* Verifies routability of candOp on candPE*/
IsRoutable(candOp, candPE, currCycle)

foreach ($predOp \in PREDs(candOp)$)
    $predPE \leftarrow$ PE on which $predOp$ is mapped
    foreach ($route \in GetRoutes(predPE, candPE)$)
        if ($currCycle <$ EndTime($predOp$) + Delay($route$))
            or (RouteNotAvailable($route$, $currCycle$))
                return false
return true
```

Fig. 2. Algorithm for verifying routability of data

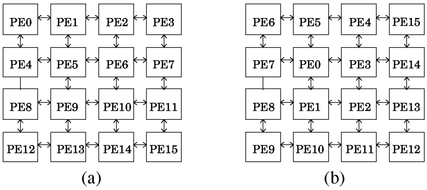


Fig. 3. (a) PE indexing in base algorithm and **(b)** PE indexing in connectivity based algorithm

different metrics that influence the mapping results and demonstrating which ones do better than others.

The rest of the paper is organized as follows. Section 2 outlines the related work. We present the base mapping algorithm in Section 3 and then explore three different cost functions which affect the performance of the algorithm. In Section 4 we present our experimental setup and results. Section 5 concludes the paper.

2 Related Work

Several efforts have focused on algorithms for mapping applications to coarse-grain architectures. Huang et al. [12] proposed a methodology to map loops on the architecture and then merge all the data paths corresponding to different loops. Venkataramani et al. [15] presented an algorithm for mapping loops written in SA-C language to the MorphoSys architecture [5] which uses a similar notion of *affinity*, but no detailed results are available.

RaPid [2] uses a C-like language to program loops which requires a considerable knowledge about the underlying architecture. Mei et al. [17] proposed a modulo loop scheduling approach to map loops on a generic reconfigurable architecture. A list scheduling based approach enriched with a priority based heuristic was used for PipeRench architecture [4] in which priority was defined on the basis of distance from the nearest *non-routing node*. Our algorithm bears some resemblance with this work, however, we differ in terms of the heuristics used and the target architecture which consists of a mesh based array of PEs in our case.

In this paper, we examine different cost functions that have an impact on the mapping of applications to a generic mesh-based coarse-grain architecture using a list scheduling based mapping approach. Based on our experimental results, we provide new insights into the metrics that affect this mapping.

3 Base Mapping Algorithm

Our base algorithm traverses the control-data flow graph of the application and schedules and maps one basic block at a time. $ScheduleMapBB$, the heuristic for scheduling the operations in a basic block and mapping them to PEs, is listed in Figure 1. The heuristic takes as input basic block to be considered ($currBB$), a global clock cycle ($currCycle$), and the list of all the PEs ($PEList$) in the architecture.

The $ScheduleMapBB$ heuristic starts by collecting a list of available or ready operations, $\mathcal{A}_{avail}$, in the current cycle. *Available operations* are operations whose data dependencies are satisfied and can be scheduled in the current cycle. $\mathcal{A}_{avail}$ and $PEList$ are then ordered by the cost functions C_{op} and C_{PE} to store back in $\mathcal{A}_{ordered}$ and $PEList$ respectively. We examine the effect of varying the C_{op} and C_{PE} cost functions in the following sections. The heuristic then maps operations to PEs starting with the PE $candPE$ having the minimum cost C_{PE}. We first make a copy of the available operation list as $\mathcal{A}_{candPE}$ for $candPE$ (lines 6 and 7 in Figure 1) since operations that cannot be mapped to $candPE$ will be removed from the $candPE$'s available list. Next, the heuristic chooses the operation ($candOp$) with the lowest cost, C_{op}, from $\mathcal{A}_{candPE}$.

Once an operation and a PE is selected, $ScheduleMapBB$ calls the $IsRoutable$ function to verify if there is a route available for the data required by $candOp$ to reach $candPE$ in $currCycle$. This function is presented in detail in the next section. If $IsRoutable$ does not find any route, then $ScheduleMapBB$ considers the next operation in $\mathcal{A}_{candPE}$ till it maps an operation to $candPE$ or no more operations are left. If $IsRoutable$ returns a true result, $candOp$ is mapped on $candPE$ and scheduled to execute in $currCycle$. We also store the usage information of different connections for each cycle. This information is used by $IsRoutable$ function to check the availability of different routes. Once we map an operation on a PE, usage information of all the connections used for this operation is updated. (lines 11 to 16 in Figure 1).

In this way, the $ScheduleMapBB$ heuristic schedules and maps operations on each PE in $PEList$ and then increments $currCycle$ when $PEList$ is exhausted. Note that, in each cycle we restart the mapping of PEs in the same fashion. This process is continued until all the available operations in the current basic block have been scheduled.

3.1 Routing Algorithm

The $IsRoutable$ function, outlined in Figure 2, verifies the ability to route data from the predecessors of $candOp$, to $candPE$ in $currCycle$. Thus, the $IsRoutable$ function checks all the routes from each $predPE$ (on which a predecessor operation is mapped) to $candPE$ by calling the function $GetRoutes$ (lines 2 and 3 in Figure 2). These routes and delays on them are determined statically before scheduling so that there is no additional run-time overhead of finding routes in terms of complexity of the algorithm. A route from $predPE$ to $candPE$ cannot be used if: either the cycle in which the predecessor operation finishes execution ($EndTime(predOp)$) summed with the delay of the route ($Delay(route)$) is larger than the current cycle ($currCycle$), or if the route is not available, i.e., some connection on the route is used by another data communication in the same cycle (lines 4 to 6 in Figure 2).

The routability verification algorithm is a constant time algorithm as routes are determined statically at the start of the scheduling. The worst case complexity of the scheduling and mapping algorithm is $O(m * n^2)$ where m is the number of PEs in the architecture and n is the number of operations in the basic block. The actual run-times of our algorithm for an architecture having 16 PEs is in the range of 10 user seconds for the designs considered (see Section 4).

Apart from verifying the routability of the operands, two other aspects that affect the performance of the scheduler are the cost functions C_{op} and C_{PE}. Over the next few sections, we present analysis of different cost functions.

3.2 Base PE and Operation Cost Functions

PEs are selected from the $PEList$ based on the cost function C_{PE}. C_{PE} for the base algorithm is defined as equal to the index of that PE; we assign indices to the PEs from PE_0 to PE_{15} (as shown in Figure 3(a)). Operations are selected from the list of available operations based on the cost function C_{op}. For the base algorithm, cost is randomly assigned to all the operations, i.e., operations are randomly selected. Next, we analyze other C_{PE} and C_{op} functions.

3.3 Priority-Based C_{op}

Since we are trying to optimize performance of the applications mapped to the coarse-grain architecture, it is intuitive to give preference to the operations that lie on the critical path through the code. Hence, we assign a priority to each operation in the input description based on the length of the chain of operations that depend on it. The *priority* of an operation is calculated as one more than the maximum of the priorities of all the operations that use its result. Operations whose results are not read (primary outputs) have a priority of one. The operation cost function (C_{op}) is taken as the negative of its priority. In other words, higher the priority, lower the cost.

3.4 Affinity Based C_{op} and C_{PE}

If operation Op_i is mapped on PE PE_m, then communication delay can be minimized by mapping operation Op_j that reads the result of Op_i on a PE PE_n that is either directly connected or connected through the fewest intermediate links to PE_m. This leads us to the notion of *affinity* between operations and PEs. We define affinity, $Aff(Op_i, PE_m)$, of an operation Op_i to a processing element PE_m as the sum of the number of *parents* of Op_i that were mapped on any PE adjacent to PE_m. Processing element PE_m is considered *adjacent* to PE PE_n if they have point-to-point connection between them. Note that, a PE is adjacent to itself. Note also that affinity of operations have to be calculated at the beginning of each cycle during the scheduling and mapping process.

Thus, affinity captures the data dependency information along with past operation to PE mapping decisions. We can, thus, use affinity to map operations to PEs with which they have the highest affinity and in the process minimize communication delays. That is, we calculate the operation cost function C_{op} as the negative of its

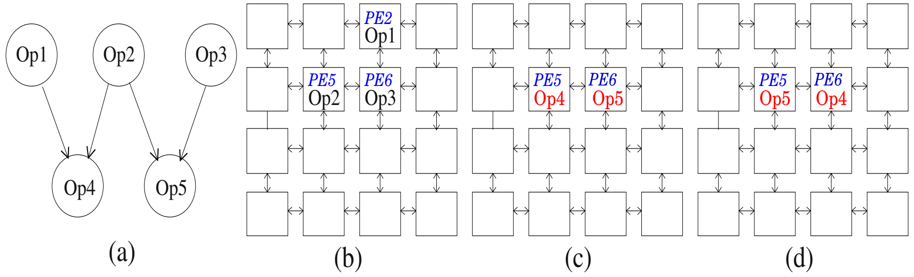


Fig. 4. **(a)** Example DFG, **(b)** Mapping of parent ops., **(c)** Mapping with a delay, **(d)** Mapping without delay

affinity. If two operations have same cost then we select the operations on the basis of their priorities. Note that, a similar notion of affinity has been discussed earlier by Venkataramani et al. [15]. We also associate an affinity with each PE. Affinity of a processing element PE_m is the sum of affinities of all the operations to PE_m, that is:

$$Aff_{PE}(PE_m) = \sum Aff(Op_i, PE_m) \,\forall\, Op_i \in \mathcal{A}_{avail}$$

We then take the cost of a PE, C_{PE}, as equal to its affinity. We schedule operations on PEs starting with the PE having *lowest affinity*. The reason for this can be understood by the example DFG shown in Figure 4(a). The target architecture we consider is shown in Figure 4(b). If we map Op_1 on PE_2, Op_2 on PE_5, and Op_3 on PE_6 as shown in Figure 4(b) then both Op_4 and Op_5 have affinity of two for PE_6. However, only Op_4 has affinity of one for PE_2 and only Op_5 has affinity of two for PE_5. By definition, the affinities of different PEs are: PE_2 has 2, PE_5 has 3, and PE_6 has 4. Now if we choose to schedule PE_6 first (i.e. PE with the highest affinity), then we *may* choose to map operation Op_5 on it, instead of Op_4 since they both have an affinity of 2 to PE_6. The resultant mapping is shown in Figure 4(c). However, this means that Op_4 will have to be mapped to PE_5 (or any other PE). This in turn means that the result of Op_1 will suffer a communication delay of one cycle to reach Op_4. However, according to the proposed algorithm, we first choose PE_2 only to find that no operation is routable on this PE. Then we choose PE_5 and find that Op_5 is routable on it, so we map it on PE_5. Now there is only one choice – that of mapping Op_4 on PE_6. The resultant mapping is shown in Figure 4(d) which has no communication delay.

Thus, we first map PEs (having non-zero affinity) in increasing order of affinity and then the rest of the PEs based on the PE indices, (starting from top left corner to bottom right corner).

3.5 Connectivity Based C_{PE}

We noticed that in mesh architectures like the one shown in Figure 3(a), the PEs at the corner of the grid (PE_0, PE_3, PE_{12}, PE_{15}) have only 3 directly connected neighbors. In contrast, PEs at the center of the grid have 5 neighbors. Mapping operations to PEs in increasing order of their indices means that the operations with highest priority and/or affinity are first mapped on the sparsely connected PEs on the edge (first row) of the grid. Thus, the data produced by these operations can be routed to a smaller number of PEs than if the operations were mapped to the PEs at the center of the grid. This observation led us to develop a PE ordering in which operations are first mapped to PEs that are *better connected*. Thus, in our *connectivity-based* cost function, we give higher preference to the PEs with more number of connections, i.e., the PEs at the center of the grid. We assign the indices to the PEs starting from the PE at the center, as shown in Figure 3(b). The PE cost function C_{PE} is equal to the index of the PE, but the indexing of PEs is changed which, in turn, changes the cost function.

4 Experimental Setup and Results

In order to evaluate the applicability of the algorithms proposed in this paper, we implemented them in a prototype compiler framework. This framework accepts an application code in C and applies basic compiler transformations such as copy propagation and dead code elimination. We used a set of seven designs drawn from the DSP domain for our experiments. All these designs consist of straight-line code with a loop (or nested loops).

For all the experiments in this paper, we consider an architecture with 16 PEs connected in a 4x4 array. We found little change in the relative numbers with larger arrays [11]. Each PE has one functional unit, which is capable of executing any operation in one cycle. The interconnect delay on direct connections is taken as 0 cycle, unless otherwise specified. The typical run time of our algorithm is 10 user seconds on a 400 MHz UltraSparc-II machine.

In all the experiments presented in this paper, we make some assumptions: (a) there is enough memory bandwidth to fetch data without any delay, (b) there are enough registers to store all the intermediate and final results, and (c) the architecture supports cycle-by-cycle reconfiguration. We plan to address these assumptions in future work.

4.1 Comparison Algorithm

In an attempt to demonstrate the efficacy of the proposed algorithm, we created an *Integer Linear Programming* (ILP) formulation of the mapping problem. To solve these ILP formulations, we used publicly available solvers; LP_SOLVE and CAP (Contig Assembly Program). Despite their reported efficiency, neither of these solvers were able to solve ILP formulations (within a few days), corresponding to a realistic application. Still, to provide some comparison baseline, we devised a heuristic that uses a zero-delay routing model. Specifically, all the PEs are assumed to be connected to each other with a full crossbar interconnect. We use a priority based list scheduling heuristic to map operations on this architectural model. This heuristic gives a lower bound on the mapping results as there is no delay induced by routing. We manually looked at the results of this heuristic and found little or no opportunity to improve them. We compare all our results with the lower bounds generated by this heuristic.

Design	Operations	No. of Cycles	IPC
Lowpass	652	106	6.15
SOR	630	115	5.48
Hydro	1290	107	12.06
ATR	508	100	5.08
FFT	286	103	2.78
Predictor	618	104	5.94
PDE	463	99	4.68

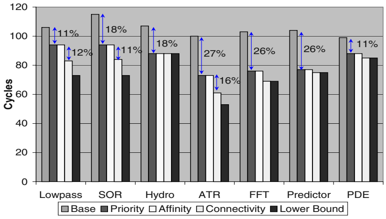


Fig. 5. (a) Mapping results for 7 DSP designs using base algorithm, **(b)** Comparison of mapping results using different cost functions for 0-delay interconnect model

4.2 Results for Base Algorithm

In order to expose the parallelism of the application, we unroll the loops that increases the number of operations to map. In case of nested loops, we unroll the innermost loop. For all the experiments in this paper we present the results with an unrolling factor of 10 since we have shown earlier that unrolling factor of 10 is sufficient to explore the inherent parallelism of the designs [10]. Table 5(a) shows the mapping results for the designs using base algorithm. Third column in this table represents the number of cycles needed to execute the design using base algorithm.

4.3 Comparison of Priority, Affinity, and Connectivity-Based Cost Functions

The results shown in Figure 5(b) demonstrate that a simple cost function based on the priority of the operations gives significant improvement (up to 27%) over base algorithm. But surprisingly a more sophisticated cost function based on affinity does not give any improvement over priority based algorithm. The reason is that the routing function $IsRoutable$ (explained in Section 3.1) implicitly considers the data dependency information when it finds the shortest routes in terms of communication time; this obviates the need for complex affinity-based cost functions.

In contrast, connectivity based algorithm gives a further improvement of up to 16% (in case of ATR) over the priority based algorithm. This is because of our earlier claim of exploiting the better connectivity of PEs at the center of the grid. In fact, in most cases, the connectivity based algorithm gives results that are close to the lower bounds.

4.4 Results for Varying Interconnect Delays

In order to support our claim about the applicability of the algorithm for different interconnect delays, we performed experiments with a delay of 1 cycle (instead of 0) on point-to-point connections. Figure 6(a) shows the performance results corresponding to different cost functions with this interconnect model. The results in this figure are similar to the results corresponding to the zero delay interconnect model. This demonstrates the effectiveness of our mapping algorithm and usefulness of the priority and connectivity based mapping strategies.

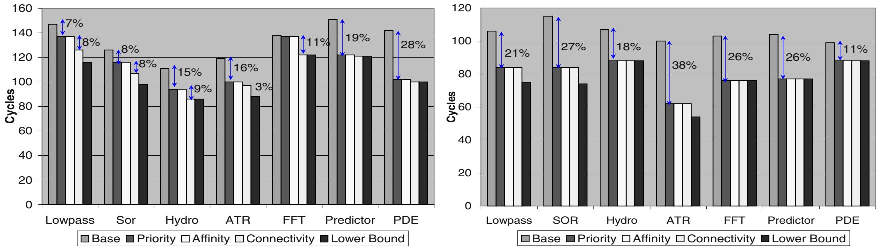


Fig. 6. (a) Comparison of mapping results using different cost functions for 1-cycle delay interconnect model, **(b)** Comparison of performance for different cost functions for the torus architecture.

4.5 Results for Torus Architectures

We introduced the notion of the connectivity-based PE cost function to use the information about the differing number of connections between PEs during mapping. However, there are some aggressive architectures in which all the PEs have same number of connections. For example, in the architectures having *torus* shaped interconnects [1], PEs in the first row (column) are also connected to the PEs in the last row (column) using wrap-around connections.

Figure 6(b) shows the performance results corresponding to this architecture model (delay on direct connection is zero cycle). These results show that with the torus architecture, as expected, the connectivity-based cost function does not give any improvement over the priority-based function. In fact the priority-based algorithm now gives results that are close to the lower bounds.

5 Conclusion

We explored three different cost functions which affect the performance of mapping applications on to coarse-grain reconfigurable architectures: (a) a priority-based function in which operations on the longest dependency chain are given preference, (b) an affinity-based function in which an operation gets preference for a PE, if any of its predecessors was mapped to that PE or its adjacent PEs, and (c) a connectivity-based function in which preference is given to PEs that have more connections to other PEs. Although the affinity-based strategy seems intuitive and useful, our experimental results show that the priority-based operation cost function coupled with connectivity-based PE cost function is sufficient enough to give results that are close to the lower bounds for most of the designs considered.

References

[1] R. W. Hartenstein, R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. *ASP-DAC*, 1995.

[2] C. Ebeling et al. Mapping applications to the rapid configurable architectures. In *FCCM*, 1997.

[3] W. Lee et al. Space-time scheduling of instruction-level parallelism on a RAW machine. In *ASPLOS*, 1998.

[4] S. Cadambi, S. C. Goldstein. Fast and efficient place and route for pipeline reconfigurable architectures. *ICCD*, 2000.

[5] H. Singh et al. Morphosys: an integrated reconfigurable system for data parallel and computation-intensive applications. In *IEEE Transactions on Computers*, 2000.

[6] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *DATE*, 2001.

[7] T. Miyamori and K. Olukotun. Remarc: Reconfigurable multimedia array coprocessor. In *FPGA*, 1998.

[8] J. Becker, M. Glesner, A. Alsolaim, J. Starzyk. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. *FCCM*, 2000.

[9] P. Schaumont, I. Verbauwhede, M. Sarrafzadeh, and K. Keutzer. A quick safari through the reconfigurable jungle. In *Design Automation Conference*, 2001.

[10] Omitted for blind review.

[11] Omitted for blind review.

[12] Z. Huang, S. Malik. Exploiting operational level parallelism through dynamically reconfigurable datapath. *DAC*, 2002.

[13] T.J. Callaham and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *CASES*, 2000.

[14] K. Bondalapati and V. K. Prasanna. Loop pipelining and optimization for run-time reconfiguration. In *RAW*, 2000.

[15] G. Venkataramani et al. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *CASES*, 2001.

[16] J. Lee, K. Choi, N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE D&T*, 2003.

[17] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures ucing modulo scheduling. *DATE*, 2003.

[18] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice Hall, 1991.

[19] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[20] R. A. Bittner, P. M. Athanas, M. D. Musgrove. Colt: An experiment in wormhole run-time reconfiguration. *SPIE*, 1996.

Analysis of a Hybrid Interconnect Architecture for Dynamically Reconfigurable FPGAs[*]

Renqiu Huang, Manish Handa, and Ranga Vemuri

University of Cincinnati, Cincinnati OH 45220, USA,
{huangr, mhanda, ranga}@ececs.uc.edu,

Abstract. Dynamically reconfigurable devices allow run-time reconfiguration to permit execution of incoming tasks or task fragments. One of the important issues in run-time reconfiguration is the fragmentation of the device area as the reconfigurable blocks are allocated and released when tasks are placed, executed and deleted. Due to those scattered, unused resources, an incoming application may not be placeable or routable. A cluster-based reconfigurable FPGA architecture is proposed to alleviate this difficulty. We present an assessment of the proposed architecture. We develop a fast evaluation tool to simulate on-line placement and routing effects on a run-time reconfigurable platform. The simulation results show the efficiency of the proposed architecture in relieving the fragmentation problem at the price of a modest increase in the number of switches.

1 Introduction

Run-time and partial reconfiguration capabilities of the state-of-the-art FPGAs [7] have shown potential for a large number of novel applications of the FPGAs. Due to the obvious advantages, a large number of research projects focus on improvements in the FPGA architecture. A large number of these projects propose coarse-grained architectures which are column-based or word-level configurable. Major benefit of course-grained architectures is the drastic reduction of placement and routing complexity and the reduction of configuration time. However, coarse-grained architecture exhibits less flexibility than the corresponding fine-grained architectures, and is suffered from area fragmentation. In this paper, we present a cluster-based reconfigurable architecture which supports multi-granular reconfiguration by taking advantages of both mesh and tree routings topologies. We evaluate proposed architecture by simulating run-time placement and routing.

[*] This work is sponsored in part by the Dayton Area Graduate Studies Institute (DAGSI) and the Air Force Research Laboratory (AFRL) research program under contract number IF-UC-00-07.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 900–905, 2004.

2 Architecture Modeling

The two-dimension mesh architecture works well for the shorter connections and is easy to be targeted by the CAD tools for the physical synthesis. For the longer wires, the tree interconnect is more attractive. A combination of those two topologies may take advantage of both mesh and tree based routing architectures. The proposed architecture consists of a mesh array of reconfigurable units (RUs), that have have CLB-like internal structures. In the proposed architecture, the RUs are grouped into clusters. The short routes are accommodated inside the cluster by using the mesh interconnect and routings between the clusters are performed in either mesh or tree-like fashion.

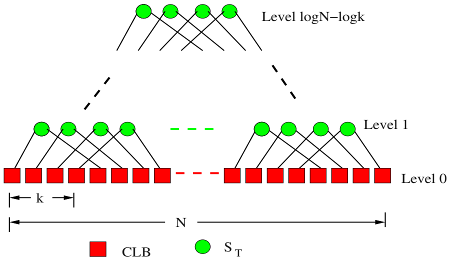


Fig. 1. One dimensional illustration of connection

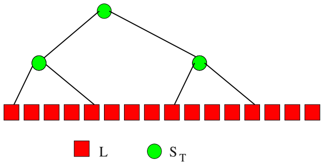


Fig. 2. A tree with N/k leaves

2.1 Connection Hierarchy

The reconfigurable units (RUs) are arranged as a$N \times N$ array. Let $k \times k$ be the size of cluster, there are k leaves (from henceforth, we use the terms leaf and RU interchangeably) for one dimension of cluster. Each leaf in a cluster has two trees connected to the corresponding leaves in the other clusters. Namely, in the mesh array of proposed architecture, every two corresponding RUs located on two clusters with distance k are formulated as two leaves of a binary tree. The same policy is applied to the root nodes from level 1 to level $\log N - \log k$ recursively and a $\frac{N}{k} \times \frac{N}{k}$ mesh-of-tree (MoT) is built. Since the trees are organized on a cluster-based array (shown in Fig.1), the total number of MoTs is $k \times k$. MoT [2] has a good property that the leaves of trees are exactly the nodes of mesh. In each row and each column, wires and additional nodes are connected to form a complete binary tree. Mesh of tree has polylogarithmic $\Theta(\log N)$ or $\Theta(\log^2 N)$ running time on a wide range problems described in [2], which is much faster than the typical running times of $\Theta(N)$ or $\Theta(N^2)$ for algorithms on meshes or trees. After construction, N/k leaves are put together remotely through a tree network (Figure 2), where $N = 16$ and $k = 4$.

For the mesh array (level 0), we assume the same interconnection structure as a typical island-style FPGAs. However, as mentioned earlier, the RUs are grouped as a cluster in which local routing resources are provided. The long connections between clusters can be routed through tree network. Further, grouping RUs into clusters have two obvious benefits from configuration point of view. Since most of the applications need reconfiguration of more than one RU, RU clustering prove to be more efficient. Moreover, the cluster organization would fast locate target area and reduce the size of hardware decoder.

Normally, in a mesh array, each RU has four neighbors (east, south, west and north). By applying our hybrid network, the neighbors have been expanded from four to six. Two more neighbors are the siblings of that RU in tree connections. Note that although we have not drawn those neighbors, all neighbor connections can be implemented by only one (through tree) or two (through mesh) switches. Increased neighbors can help in realizing multi-granular configurations.

2.2 Switch Overheads

Let us consider two FPGAs with two different architectures: the conventional 2-D island array and the proposed architecture, which is a hierarchical combination of mesh and tree interconnect topologies. Let m denotes the number of pins of a RU, and I is the number inputs and outputs from one side of RU to the channel. We have $I = m/4$ for mesh array. The sizes of two FPGAs are set to $N \times N$. In order to be comparable, all bisection widths at highest level are set to be equal, which is pmN^2. Here, mN^2 is the total number of possible signals. Since any signal may cross the middle line of chip several times or not pass it at all, we multiply it by a parameter p. Let W be the channel widths of conventional model. Since there are N connection boxes and $N-1$ switch boxes per row/column either along horizontal or vertical dimension. The bisection width can be represented as $N \times I + 2 \times (N-1) \times W$. Therefore, we have:

$$W = \frac{mN(4pN-1)}{8(N-1)} \tag{1}$$

Following the definitions in [1], if we assume the flexibility of connection block is W and the flexibility of switch block is F_s, then for the conventional FPGA, the total number of switches is estimated as:

$$SW_A = N^2 \times (2 \times 2 \times I \times W + \binom{4}{2} \times F_s \times W) = \frac{m(m+6F_s)(4pN-1)N^3}{8(N-1)} \tag{2}$$

Note that in the above calculations, we associate each logic block with one switch box and two connection boxes. Therefore, the total switches are the N^2 times of summation of switches in the switch and connection boxes.

For the proposed model, the total number of switches required for the tree interconnect can be calculated as:

$$SW_T = 2kN \sum_{i=1}^{\log \frac{N}{k}-1} \left(2 \times m + m^2\right) \times 2^{-i} \times \frac{N}{k} = 2(m^2+2m)(N-2k)N \tag{3}$$

In a tree network, the switches at the box at the ith level (where $1 \le i \le \log \frac{N}{k} - 1$) are given by $m^2 + 2m$. Note that we assume the fully connected switch box which is the worst case scenario. The numbers of switch boxes are $2^{-i} \times \frac{N}{k}$, and the tree has $\log \frac{N}{k}$ levels. From Eq.2 and Eq.3, we see that the number of switches needed for the two architectures are $O(N^3)$ and $O(N^2)$ respectively. For the typical settings ($N = 128, p = 0.5, F_s = 3, m = 30$, and $k = 4$), the overheads are under 5%.

3 Evaluation Methodology

In this section, we discuss the experiments conducted to evaluate the proposed architecture and to compare it with traditional mesh architecture. In a dynamically reconfigurable system, the applications (tasks) may come at arbitrary intervals, and those tasks need to be placed on the reconfigurable platform at run time. In our model, the FPGA is modeled as a two dimensional array and we use the algorithm [6] to find maximal empty rectangles (MER) as possible task placement locations. Each task consists of a set of hard macros. A macro can be configured at a location where both placement and routing constraints are satisfied. We use the fitting strategies [5] as our placement rules and model the routing delay proportional to the Manhattan distance between two rectangles. If a macro can not be placed due to either lack of suitable MER or limited routing resources, the routine flags failed status flag. We denote it as task rejection (shown in Fig.4).

We randomly generated the tasks with their sized generated randomly with uniform distribution between [1, Max_size]. The set of macros for each task are randomly generated with uniform distribution of their size in interval [1, Max_size] until the total area of all generated macros reaches the area of that task. For the life time of each task and the interval time between any two consecutive tasks, we follow [6]. We generate 100 tasks for one execution and we ran that experiment 100 times and report average values of results. We need to point out that all tasks are assumed to be data independent of other tasks.

We use two different scheduling methodologies. In first approach, if a task cannot be placed due to placement or routing constraints, that task is considered rejected and no further attempt is made to place it. After placement of each task, we calculate the free area left (Shown in Fig.3). In the second scheduling methodology, if a task cannot be placed at its arrival time, it is put into a queue and attempt is made to place it again after deletion of a task from the FPGA. All tasks are executed on the FPGA at the expense of delay in their execution(Shown in Fig.4).

4 Simulation Results

Three kinds of reconfigurable models are evaluated. Model A is Xilinx Virtex-like FPGA, which is partially reconfigurable in vertical, chip-spanning columns. Model B is fine-grained FPGAs, which has the same structure as model A while reconfigurable in each RU (CLB). Model C is the proposed architecture. We assume all three models have the same number 128 × 128 RUs or CLBs. The Max_size of a task is set to 45 × 45.

We have different policy for selection of a maximal empty rectangle for placement of macro in different models. In model A, we choose a MER with minimum width that can accommodate a macro and for model B and C, we choose a rectangle with minimum area to place a macro. Bottom right point of a macro is placed at the bottom right point of the selected MER for Model A and B. For Model C, the right bottom point of macro is placed at the right-est and bottommest point of the cluster in that rectangle. In model A, if a macro is placed in a column, that column cannot be reused for placement of other macros until the previously placed macro is removed. For model B, all unoccupied area can be allocated to any macro. In case of model C, macros are placed at the boundary of cluster only. So, the bottom right point of a macro is placed at the bottom right location of the bottom right cluster in the selected MER.

Fig. 3 shows that the empty area after each configuration is more than 60% of the chip area. This implies all three models are suffering fragmentation seriously. We draw the ratios of rejected tasks, left area per unit time and extra waiting time per execution of three models in Fig. 4. It is clear that our model has significant improvements in terms of task rejection and the waiting time as compared to the other models. The improvement in area utilization is also noticeable. Since reconfigurable area is pretty large, some percentage improvements are still significant.

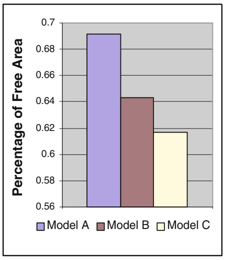


Fig. 3. Free area per configuration

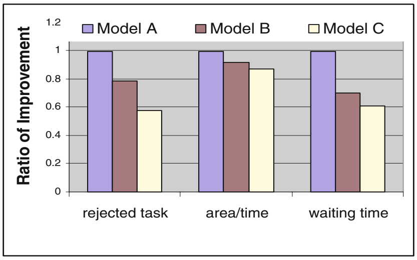


Fig. 4. Ratios of improvement

5 Conclusions

In this paper, we present a hybrid interconnect FPGA. The proposed model is evaluated using an evaluation tool developed to compare its run time configuration properties with other architecture models. Results shows that the proposed architecture is superior to other architectures for run-time reconfiguration.

References

1. Brown, S., Francis, R., Rose, J., and Vranesic, Z.: Field-programmable gate arrays. Kluwer Academic Publishers, 1992
2. Leighton, F.: Introduction to parallel algorithms and architectures: array, trees, hypercubes. Morgan Kaufmann Publishers Inc., 1992
3. Lai, Y., and Wang, P.: Hierarchical interconnection structures for field programmable gate arrays. IEEE Trans. on VLSI. v.5 n.2 (1997) 186–196

4. Rubin, R., and DeHon, A.: Design of FPGA interconnect for multilevel metalization. 11th international symposium on FPGAs, (2003) 154–163
5. Walder, H., Steiger, C., and Platzner, M.: Fast online task placement on FPGAs: free space partitioning and 2D-hashing. IPDPS 2003
6. Manish, H., and Vemuri, R.: an efficient algorithm for finding empty space for online placement. DAC, 2004
7. Xilinx, Inc.: Virtex-II Pro platform FPGA handbook, Jan. 2002

Mapping Basic Recursive Structures to Runtime Reconfigurable Hardware

Hossam ElGindy and George Ferizis

School of Computer Science & Engineering
The University of New South Wales
Sydney, NSW, Australia
{hossam,gferizis}@cse.unsw.edu.au
FAX: +61 2 9385 5995

Abstract. Recursion is a powerful method that is used to describe many algorithms in computer science. Processing of recursion is traditionally done using a stack, which can act as a bottleneck for parallelising and pipelining different stages of recursion. In this paper we propose a method for mapping recursive algorithms, without the use of a stack structure, into hardware by pipelining the stages of recursion. The use of runtime reconfigurable hardware to minimise the amount of required hardware resources, and the related issues to be resolved, are addressed.

1 Introduction

Recursion is a powerful tool that is heavily used for the development of programs today, that allows a programmer to compactly and easily design code that would be much more complex in design if it is done using iteration. Recursion makes it easier to develop programs and makes it inherently easier to debug due to much simpler code being present for reviewing. Recursive descriptions can be seen in many elegant algorithms such as tree traversals and "divide-and-conquer" geometrical and mathematical problems in multi-dimensions [1]. Iterative equivalents for such algorithms are nowhere near as elegant. Furthermore recursion is fundamental to functional language paradigms which rely on recursion due to an absence of common iterative operators.

On a general-purpose processor, recursive descriptions are implemented by the use of a stack that is used to temporarily store arguments and results between stages of a recursive function. This solution could be implemented easily on an FPGA-based system. However the ability of such systems to provide customised pipelining and parallelism, which software implementations on a general-purpose processor cannot provide, will not be possible with such an implementation.

A solution for this problem is not provided by the majority of high level language development tools such as Handel-C [2], which do not support recursive procedures. The exclusion of recursive constructs from such tools is a testimony to the difficulty of the process of mapping them into hardware.

Previous work into mapping recursive functions into FPGAs without the use of a stack has relied on transforming the function into a loop [3]. However it

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 906–910, 2004.

makes no attempt to parallelise the recursive calls that are made in instances where a function calls itself multiple times.

In this paper we present a method for mapping *"basic"* recursive functions into reconfigurable hardware that unrolls the recursion with the use of runtime reconfiguration, and hence does not use a stack.

Our approach builds on the previous work on unrolling iterative loops for the purpose of mapping them into runtime reconfigurable hardware [4] and addresses the additional difficulties that are unique to recursion. These difficulties include hardware allocation which does not have the constant growth rate as in iterative loops, and minimising the cost of runtime reconfiguration. In this paper we allude to the use of special-purpose logic to predict hardware requirements for the function being unrolled at the earliest time possible. More information can be found in the full technical report [5].

We begin by defining what is meant by *"basic"* recursive functions. A *"basic"* recursive function is a function that, calls itself zero or a constant number of times, at any depth of recursion all instances of the function call themselves the same number of times, and when given initial arguments the maximum depth of recursion can be accurately calculated, or can be calculated at any time during the processing of an input stream of arguments.

We begin by presenting a method for mapping this conceptual model of recursion into hardware solutions on FPGAs that minimises the effect of runtime reconfiguration delay. Details of two case studies, merge sort [6] and an implementation of Strassen's matrix multiplication algorithm [7] can be found in [5]. Merge sort provides a motivating example, while the matrix multiplication case study illustrates how hardware can be allocated unevenly between recursive instances.

2 General Recursive Problem

We shall now describe a model of recursive processes that illustrates our proposed solution. We model a basic recursive function as a tree. This is shown in figure 1(a), which has an example of a function that calls itself twice.

Our implementation is based on pipelining the operation between consecutive levels with an area on the chip dedicated to processing the nodes contained in each level. To allocate the minimum amount of hardware the number of levels in the recursive tree must be estimated accurately at runtime. The necessary logic to meet this estimate can then be configured at runtime. This requires runtime reconfiguration. Therefore an effective implementation of this process requires the ability to hide the delay that is produced by runtime reconfiguration.

The example in figure 1(a) is not tail recursive, with values being returned back up the recursive tree. This communication pattern presents problems in pipelining if only a single area of logic is dedicated per level, as this area will be blocking while waiting for subsequent logic to return data. A solution to this is presented later in the paper.

We also point out that the recursion in figure 1(a) is balanced, whereas not all recursive trees are balanced. Unbalanced recursion presents problem in the scheduling of data being sent through the pipeline, as well as introduces an increased complexity in hardware allocation. This is due to the difficulty in predicting the node population of a level of recursion due to the irregular growth rate that is a result of unbalanced recursion. The problem presented by unbalanced recursion is not discussed in this paper and has been left for future work.

3 General Solution

The tree in figure 1(a) models a balanced recursive function with two recursive calls. As indicated by the arrows directed from the children nodes back to their parent nodes, the recursive calls return values that are processed by the parent nodes.

Our approach is similar to previous approaches [8,9] that transform general recursive calls into two tail recursive calls. Such transformations have been suggested in the past as software compiler level optimisations [8,9]. These earlier techniques rely on transforming a recursive call into two tail recursive calls and using a stack to hold arguments generated in the first tail recursive call for the second tail recursive call. Such methodology does not attempt to make use of parallelism and pipelining opportunities that modern hardware offers. Our approach eliminates the need for a stack by taking advantage of multiple processing elements and networking that current reconfigurable hardware makes possible to implement and utilise on-demand.

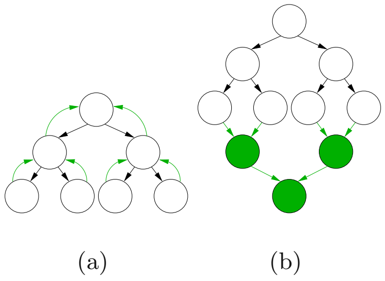

(a) (b)

Fig. 1. A recursive tree, and the corresponding DAG produced.

Our mapping begins by removing upward returns in the tree by adding another tree that shares the leaves with the original tree. The result is a DAG as shown in figure 1(b).

Construction of the new graph involves the splitting of the statements in a recursive function into two disjoint sets. The first set contains the statements that occur before the recursive calls. These statements correspond to the unshaded nodes in the graph. The second set contains the statements that occur after the recursive calls. These statements correspond to the shaded nodes in the graph. For each set we define a different logic type as follows:

1. *PreRecursion*: This corresponds to logic that can compute during the expansion portion up to and including the point of truncation. This is in effect the first tail recursive function.
2. *PostRecursion*: This corresponds to logic that can compute during the collapsing of the tree. This is in effect the second tail recursive function.

An instance of a *PreRecursion* unit and a *PostRecursion* unit, creates a complete instance of the original function. As can be seen in figure 1(b), the shaded nodes which correspond to units of type *PostRecursion* are created by mirroring the unshaded nodes. The nodes that are related in this mirroring are named twins. Thus a twin corresponds to a complete function instance.

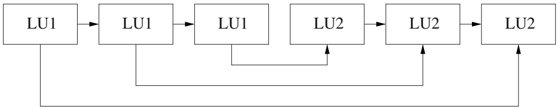


Fig. 2. Logic unit allocation

All instances of each unit communicate between levels as shown in figure 1(b), with extra communication between twin units. They all take in the necessary arguments to compute the values they are to output.

One possible layout in hardware for this DAG is an array as shown in figure 2, with the communication between logic units set so that the concept of a twin can be seen. The logic to compute the results of a node may be replicated in proportion to the node population in that level of the graph to maintain a constant throughput. However there are instances where the communication between different levels of recursion incurs more cost than the actual computation. In this case throughput is bounded by the communication between logic units, and thus throughput remains constant irrespective of the amount of logic configured. Following this observation a minimum amount of logic is configured, which is the amount needed to compute the resulting computation of a single node.

As the depth of recursion increases, our mapping dedicates the minimum amount of logic needed as dictated by the maximum depth reached. This is achieved by the use of runtime reconfiguration to configure logic on demand. Rruntime reconfiguration is a task that requires time orders of magnitude longer than the time for performing computation. It will not be desirable to have the system stall and wait for more logic to be reconfigured when logic for a new recursive level is required.

To combat this problem, we implement a prediction mechanism that monitors input into the system and detects the need for more logic before it is actually needed. All input items pass through this prediction circuitry, before being placed into the compute logic. The collected information is used to hide as much of the performance penalty introduced by runtime reconfiguration as possible. The prediction logic will also be responsible for deciding how much hardware should be allocated when reconfiguring new logic, in respect to the node population at that depth of recursion.

When the prediction logic of the pipeline decided that more hardware is needed when a new item enters the system it has time related to the length of the currently configured pipeline to configure new logic. Prior configuration of a suitable pipeline depth will make it possible to hide this cost completely.

4 Conclusion

A method for mapping basic recursive structures to runtime reconfigurable hardware has been demonstrated. Case studies have been conducted to show the method's validity and correctness, based on an ability to predict the need for runtime reconfiguration well before it is required. Whereas only basic recursive structures have been mapped to reconfigurable hardware it is believed that similar techniques can be used to map more complex and general recursive structures. The cases of unbalanced recursion and recursive problems where prediction may not be optimal, are left as topics for future research.

References

1. Bentley, J.L., Shamos, M.I.: Divide-and-conquer in multidimensional space. In: Proceedings of the eighth annual ACM symposium on Theory of computing, ACM Press (1976) 220–230
2. Handle-C language reference manual. (http://www.celoxica.com/)
3. Maruyama, T., Hoshino, T.: A C to HDL Compiler for Pipeline Processing on FPGAs. In: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society (2000) 101
4. Bondalapati, K., Prasanna, V.K.: Loop pipelining and optimization for run time reconfiguration. Reconfigurable Architectures Workshop (2000)
5. ElGindy, H., Ferizis, G.: Mapping basic recursive structures to runtime reconfigurable hardware. Technical Report 419, School of Computer Science and Engineering, University of NSW (2004)
6. Orenstein, J., Merret, T., Devroye, L.: Linear sorting with O(log N) processors. BIT **23** (1983) 170–180
7. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **13** (1969) 354–356
8. Arsac, J., Kodrato, Y.: Some techniques for recursion removal from recursive functions. (1982) 295–322
9. Liu, Y.A., Stoller, S.D.: From Recursion to Iteration: What are the Optimizations? In: Partial Evaluation and Semantic-Based Program Manipulation. (2000) 73–82

Implementation of the Extended Euclidean Algorithm for the Tate Pairing on FPGA

Takehiro Ito, Yuichiro Shibata, and Kiyoshi Oguri

Department of Computer and Information Sciences, Nagasaki University
{takehiro,shibata,oguri}@pca.cis.nagasaki-u.ac.jp

Abstract. The Tate pairing is a mapping which has good functionality for constructing elliptic cryptosystems, while its computation is a hard task. Especially, calculation of an inverse element using the extended Euclidean algorithm over a finite field $\mathbb{F}_p$ tends to be a bottleneck. In this paper, several kinds of implementation of the extended Euclidean algorithm on an FPGA are shown and compared. Effects of introducing Montgomery multiplication methods are also analyzed.

1 Introduction

The Tate pairing [2], which is a mapping on an elliptic curve, has been attracting much attention since it provides good functionality for constructing cryptosystems such as the identity-based key exchange. However, the computation of the Tate pairing is a hard task, therefore it is essential to find efficient implementation in case of practical use [3]. One of the promising approaches is hardware implementation [7],[6],[1]. Especially, reconfigurable implementation [1] is efficient since the architecture can be flexibly tailored for an adopted elliptic curve. Most of these existing architectures for cryptosystems deal with elliptic curves over an extension field $\mathbb{F}_{2^m}$, since multiplication and division in $\mathbb{F}_{2^m}$ can be reduced to shift operations. However, this means that elliptic curves that can be adopted in cryptosystems are restricted. To keep the generality of the Tate pairing, implementation in a prime field $\mathbb{F}_p$ where p is a large prime that has not fewer than 160 bits is desirable [3]. In this paper, we show some FPGA implementation of the extended Euclidean algorithm and provide area and performance tradeoff analysis.

2 Background

The Tate pairing was introduced into cryptography by Frey and Rück [2]. Here, we show an outline of the Tate pairing. Detailed definition can be seen in [3] and [4]. Let consider an elliptic curve over a finite field $\mathbb{F}_p$. Let l be a positive integer coprime to p. Let k be a positive integer such that the field $\mathbb{F}_{p^k}$ contains the lth roots of unity. This means $(p^k - 1) \bmod l = 0$. The Tate pairing operates on l-fold divisors, i.e. divisors D such that lD is principal (see [8] for an introduction to divisors). Given two l-fold divisors P and Q defined over an extension field $\mathbb{F}_{p^k}$,

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 911–916, 2004.

there exist f_P such that $\div(f_P) = lP$. The Tate pairing of P and Q is defined as $t_m(P,Q) = f_P(Q)^{(p^k-1)/l}$.

According to our primary implementation and analysis, calculation of an inverse elements over $\mathbb{F}_p$ with the extended Euclidean algorithm dominates the computation time of the Tate paring. Therefore, in this paper, we address efficient implementation of the extended Euclidean algorithm on an FPGA.

Let $b \in \mathbb{F}_p$. The inverse element of b over $\mathbb{F}_p$ is written as $b^{-1} \in \mathbb{F}_p$, and defined as $b \times b^{-1} \equiv 1 \pmod{p}$. To calculate an inverse element over $\mathbb{F}_p$, the extended Euclidean algorithm (Fig. 1) is widely used.

```
Input: $b, p$
Output: $t = b^{-1}$ over $\mathbb{F}_p$
Initialize: $t_0 = 0, t = 1$
$q = \lfloor \frac{p}{b} \rfloor$
$r = p - q \times b$
while $r > 0$
    $tmp = t_0 - q \times t$
    $t_0 = t$
    $t = tmp$
    $p = b$
    $b = r$
    $q = \lfloor \frac{p}{b} \rfloor$
    $r = p - q \times b$
return $t$
```

Fig. 1. The extended Euclidean algorithm

3 Implementation and Evaluation

3.1 Structure of the Extended Euclidean Hardware

The extended Euclidean algorithm consists of the four basic operations of arithmetic as mentioned above. However, since the Tate pairing requires p has not fewer than 160 bits, we must efficiently implement multiple precision arithmetic units on an FPGA. Here, let p have 256 bits to ensure high degree of security.

Fig. 2 shows the overall structure of the extended Euclidean hardware that we designed. The hardware consists of a 256-bit multiplier, a 256-bit subtractor and two 256-bit dividers. The loop body operations consisting of a multiplier, a divider and a subtractor are repeated while the value of r remains positive. We chose Xilinx xc2vp70 as a target FPGA. The designs are synthesized and mapped on the FPGA using Xilinx ISE tool.

3.2 Addition Module

There are several implementation alternatives for a 256-bit addition module in terms of granularity. While a 256-bit adder can be implemented as combinational circuits, sequential implementation using a smaller adder is also possible. Fig. 4 shows an example of the sequential approach which uses a 128-bit adder. Note that the adder module also includes a subtractor, since the addition must be done in a prime field $\mathbb{F}_p$.

Table 1. Addition modules

bits	critical path	cycles	proc time
16	4.193 ns	66	276.7 ns
32	4.263 ns	34	144.9 ns
64	5.293 ns	18	95.3 ns
128	7.629 ns	10	76.2 ns
256	12.311 ns	4	49.2 ns

As the granularity of a basic adder becomes finer, required clock cycles are increased while the operational frequency is improved. In order to analyze the tradeoff, we implement and evaluate five addition modules. The results are summarized in Table 1. The module using a 16-bit adder shows the shortest critical path and achieves 2.9 times frequency compared with the 256-bit module. The number of required clock cycles is increased by 16.5 times and the processing

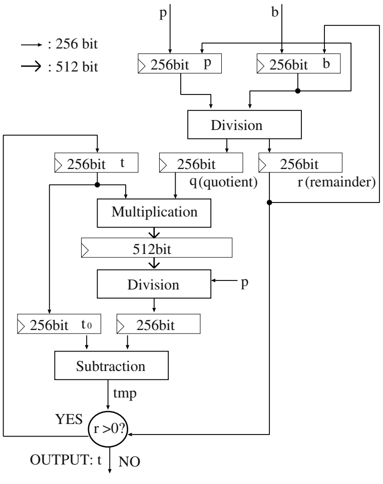


Fig. 2. The extended Euclidean hardware

```
Input: $A = (a_{k-1}, \ldots, a_1, a_0)$,
    $B = (b_{k-1}, \ldots, b_1, b_0), p$
Output: $Z = AB2^{-n} \pmod p$
Advance calculation: $q = -p^{-1} \pmod{2^r}$
$Z = 0$
for $i = 0$ to $k - 1$
    $t = (z_0 + a_i b_0)q \pmod{2^r}$
    $Z = (Z + a_i B + tp)/2^r$
if $Z \geq p$ then $Z = Z - p$
return $Z$
```

Fig. 3. Algorithm (MM1)

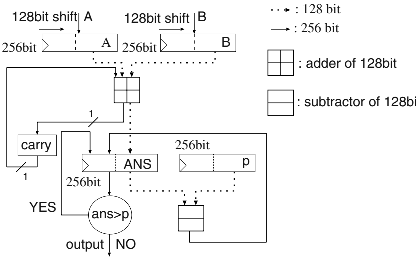


Fig. 4. The 128bit-adder

```
Input: $A = (a_{k-1}, \ldots, a_1, a_0)$,
    $B = (b_{k-1}, \ldots, b_1, b_0)$,
    $p = (p_{k-1}, \ldots, p_1, p_0)$
Output: $Z = AB2^{-n} \pmod p$
Advance calculation: $q = -p^{-1} \pmod{2^r}$
$Z = 0$
for $i = 0$ to $k - 1$
    $w = 0$
    $t = (z_0 + a_i b_0)q \pmod{2^r}$
    for $j = 0$ to $k - 1$
        $S = (z_j + a_i b_j + tp_j) + w$
        if $(j \neq 0)$ then $z_j = S \pmod{2^r}$
        $w = S/2^r$
    $z_{k-1} = w$
if $Z \geq p$ then $Z = Z - p$
return $Z$
```

Fig. 5. Algorithm (MM2)

time is also degraded by 5.6 times. As a result, the module using a 256-bit combinational adder shows the best processing time among the evaluated design alternatives.

3.3 Multiplication Module

In modulo p arithmetic, addition of multiple number of p does not have any effects. Making the best use of this property, it is possible to make the lower n bits of the multiplication results to be 0 by adding an appropriate multiple of p. In order to obtain $AB \pmod p$, a result value of the Montgomery [5] multiplication must be shifted by n bits, then the modulo in p must be calculated. However, the Montgomery multiplication itself is able to accept $AB2^{-n} \pmod p$ instead of $AB \pmod p$. Therefore, modular exponentiation, which is

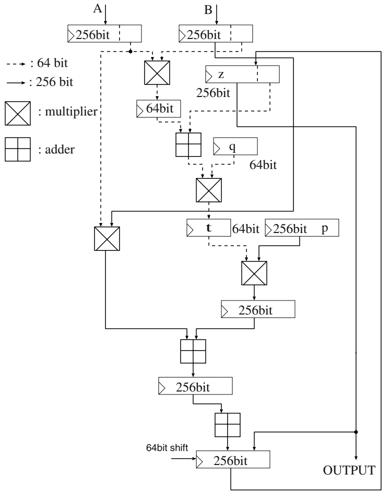


Fig. 6. MM1 multiplier

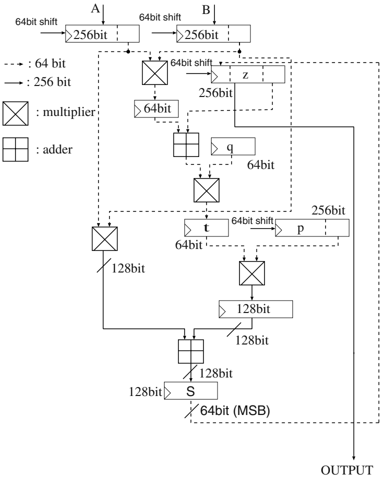


Fig. 7. MM2 multiplier

a basic operation of RSA cryptography algorithm, is efficiently calculated by repeating the Montgomery multiplication algorithm [7]. On the other hand, in the extended Euclidean algorithm, results of the multiplication module are used by the subtraction module as shown in Fig. 2. This means that we need $AB \pmod{p}$, and the Montgomery multiplier still must be coupled with a division module for the modulo calculation. We evaluate the following four multipliers.

Montgomery multiplier 1 (MM1): This multiplier is implementation of the basic algorithm shown in Fig. 3. In the Montgomery multiplication, A and B are divided into k r-bit. Here, we use a_i to show the i-th r-bit block of A (from the LSB side). We chose $r = 64$ since it has been reported this is efficient for a Xilinx FPGA [9]. As Fig. 6 shows, MM1 includes a 64x256-bit multiplier as well as normal 64-bit multipliers.

Montgomery multiplier 2 (MM2): As Fig. 5 shows, this algorithm unifies the size of multipliers to 64 bits by introducing a nested loop. While the clock cycles are increased compared to MM1, it could achieve higher frequency eliminating the 64x256-bit multiplier. The structure of MM2 is shown in Fig. 7.

256-bit combinational multiplier (COMB): This module simply multiplies inputted 256-bit values using combinational circuits.

Sequential multiplier (SEQ): A product is sequentially calculated using a 64-bit combinational multiplier and a 256-bit adder. As shown in Fig. 8, 128-bit partial products are accumulated in the Z register, while A, B and Z registers are rotated by 64 bits. This process is repeated 16 times to generate the final result.

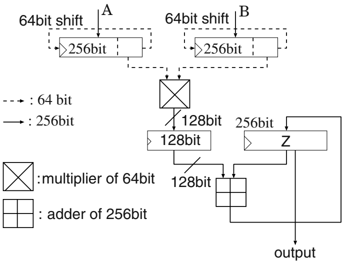


Fig. 8. SEQ multiplier

Table 2. Multipliers and a divider

type	critical path	cycles	proc time
MM1	18.690 ns	28	523.3 ns
MM2	12.361 ns	127	1569.8 ns
COMB	31.455 ns	1	31.5 ns
SEQ	13.977 ns	32	447.3 ns
divider	12.686 ns	1029	13053.9 ns

Table 3. Extended Euclidean hardware

type	critical path	cycles	proc time
MM1	22.040 ns	290,939	6.41 ms
MM2	22.040 ns	300,146	6.62 ms
COMB	31.455 ns	288,428	9.07 ms
SEQ	22.040 ns	291,311	6.42 ms

For each type of the multipliers a divider is required to calculate the modulo in p. Here, we choose a normal nonrestoring divider. Table 2 shows implementation results of the multipliers and the divider. The results show the critical paths of multipliers are longer than those of adders shown in Table 1. While COMB shows the best processing time, its operational speed is the worst among the evaluated multipliers. Meanwhile, MM1 achieves 3.0 times better processing time compared with MM2. This implies a large combinational multiplier does not critically degrade the circuit performance. However, as far as we focus only on multipliers, the Montgomery approach does not show the effect.

3.4 Evaluation of the Extended Euclidean Hardware

Here, we evaluate and compare the whole extended Euclidean hardware shown in Fig. 2. The implementation results are summarized in Table 3. Unlike the results in Table 2, the selection of a multiplier does not affect the critical path except for COMB. This is due to relatively complicated control structure including long word comparators. Therefore, the adoption of COMB shows the worst processing time, although it achieves the best performance in Table 2. Table 3 shows MM1 is the most efficient among the evaluated implementation alternatives in terms of processing time and hardware amount. As Table 2 shows, MM1 generates its result faster than SEQ by 4 cycles, and this difference results in speed up of 10 μs in the whole algorithm.

Table 4. Required resources

type	Slices	Flip Flops	MULT18Xs
MM1	9028/33088	7155/66176	136/328
MM2	11597/33088	9079/66176	42/328
COMB	11181/33088	7043/66176	255/328
SEQ	9868/33088	6940/66176	67/328

Table 4 shows the required hardware resources for the extend Euclidean hardware. According to the results, MM1 reduces the number of required slices by 8.5% compared with SEQ. On the other hand, MM1 doubles the number of occupied 18-bit built-in multiplications in the FPGA. As a result, it is shown that the Montgomery approach is efficient when the extended Euclidean algorithm is implemented on an FPGA that has rich resources of built-in multiplications.

4 Conclusion

Implementation alternatives for the extended Euclidean algorithm in $\mathbb{F}_p$ on a Xilinx FPGA were evaluated and analyzed. It is shown that the Montgomery approach is efficient for an FPGA that has rich built-in multiplications.

Acknowledgment. The authors express their sincere gratitude to Dr. Harasawa at Nagasaki University for his fruitful suggestion and discussion.

References

1. Bednara, M., Daldrup, M., Gathen, J., Shokrollashi, J., Teich, J.: Reconfigurable implementation of elliptic curve crypto algorithms. Proc. RAW (2002) 0157b
2. Frey, G., Rück, H.: A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. Math. Comp. **62**, 206 (1994) 865–874
3. Galbraith, S., Harrison, K., Soldera, D.: Implementing the Tate pairing. Proc. ANTS (2002) 324–337
4. Izu, T., Takagi, T.: Efficient computations of the Tate pairing for the large MOV degrees. Proc. ICISC (2002) 283–297
5. Montgomery, P.: Modular multiplication without trial division. Math. Comp., **44**, 170 (1985) 519–521
6. Orlando, G., Parr, C.: A scalable $GF(p)$ elliptic curve processor architecture for programmable hardware. Proc. CHES (2001) 349–363
7. Satoh, A., Takano, K.: A scalable dual-field elliptic curve cryptographic processor. IEEE Trans. Computers **52**, 4 (2003) 449–460
8. Silverman, J.: The arithmetic of elliptic curves" Springer GTM. **106** (1986)
9. Suzuki, D., Ichikawa, T., Kasuya, T.: Montgomery multiplier on FPGA using embedded multiplier and memory. Tech. IEICE Reconf., **1** (2003) 81–87 (in Japanese)

Java Technology in an FPGA

Martin Schoeberl

JOP.design, Vienna, Austria
martin@jopdesign.com

Abstract. The application of Field Programmable Gate Arrays (FPGA) has moved from simple glue logic to complete systems. The potential for FPGA use in embedded systems is steadily increasing continuously opening up new application areas. Low cost FPGA devices are available in logic densities where the CPU with necessary peripheral device can be integrated in a single device. Java, with its pragmatic approach to object orientation and enhancements over C, got very popular for desktop and server application development. Some features of Java, such as thread support in the language, could greatly simplify development of embedded systems. However, due to resource constraints in embedded systems, the common implementations of the Java Virtual Machine (JVM), as interpreter or just-in-time compiler, are not practical. This paper describes an alternative approach: JOP (a Java Optimized Processor) is a hardware implementation of the JVM with short and predictable execution time of most bytecodes. JOP is implemented as a configurable soft core in an FPGA. With JOP it is possible to develop applications in pure Java on resource constraint devices.

1 Architecture

JOP is the implementation of the Virtual Machine (JVM) [3] in hardware. JOP is intended for applications in embedded real-time systems and the primary implementation technology is in an FPGA, which results in the following design constraints:

- Every aspect of the architecture has to be time predictable
- Low worst-case execution time is favored over average execution speed
- The processor has to be small enough to fit in a low cost FPGA device

JOP is a full-pipelined architecture with single cycle execution of microinstructions and a novel approach to map Java bytecode to these microinstructions. Fig. 1 shows the datapath of JOP. Three stages form the core of JOP, executing JOP microcode. An additional stage in the front of the core pipeline fetches Java bytecodes, the instructions of the JVM, and translates these bytecodes to addresses in microcode. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. The third pipeline stage performs, besides the usual decode function, address generation for the stack ram. Since every instruction of a stack machine has either *pop* or *push* characteristics, it is possible to generate the address for fill or spill for the *following* instruction in this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 917–921, 2004.

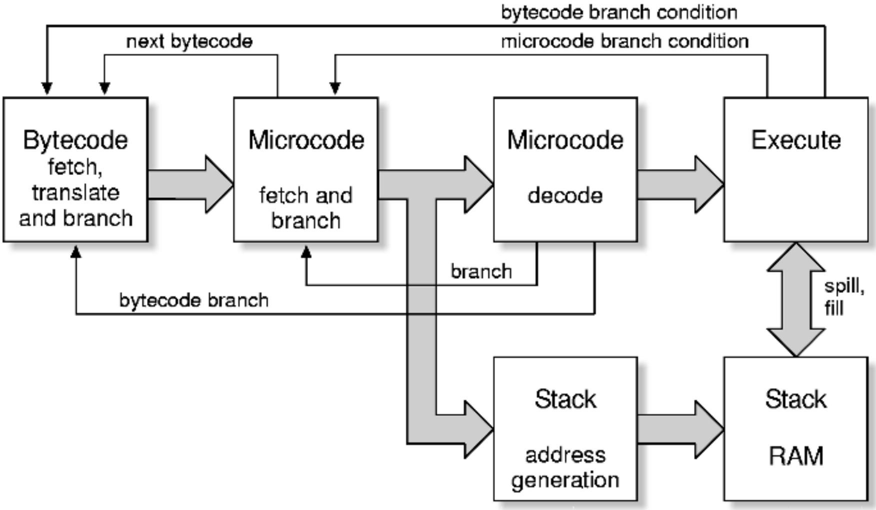

bytecode branch condition
next bytecode
microcode branch condition
Bytecode fetch, translate and branch
Microcode fetch and branch
Microcode decode
Execute
branch
bytecode branch
spill, fill
Stack address generation
Stack RAM

Fig. 1. Datapath of JOP

Memory blocks in an FPGA are usually small (e.g. 0.5 KB) with two independent read/write ports of configurable size. With these constraints, a stack machine is an attractive architecture in an FPGA:

- The stack can be implemented in internal memory
- A register file in a RISC CPU needs two read ports and one write port for single cycle instructions. A stack needs only one read and one write port
- Instruction set is simpler and instruction coding can be reduced to 8 bit
- No data forwarding is necessary

The basic stack is implemented in a FPGA memory block. The two top elements of the stack are implemented as register *A* and *B*. Every arithmetic/logical operation is performed with *A* and *B* as source and *A* as destination. All load operations (local variables, internal register and memory) result in the value loaded in *A*. Therefore no write back pipeline stage is necessary. *A* is also the source for store operations. Register *B* is never accessed directly. It is read as implicit operand or for stack spill on push instructions and written during stack spill and fill. Instructions of a stack machine can be categorized with respect to stack manipulation in pop or push:

Pop instructions reduce the stack. Register *B* (TOS-1) from the execution stage is filled with a new word from stack RAM. The stack pointer is decremented. In short:

```
A op B → A, stack[sp] → B, sp-1 → sp
```

Push instructions generate a new element on the stack. Register *B* is spilled to stack RAM and the stack pointer is incremented:

```
data → A, A → B, B → stack[sp+1], sp+1 → sp
```

An instruction needs either read or write access to the stack RAM. Access to local variables, also residing in the stack, need simultaneous read and write access:

```
stack[vp+0] → A, A → B, B → stack[sp+1], sp+1 → sp
```

2 Microcode

There is a great variation in complexity of Java bytecodes, the instructions of the JVM. There are simple instructions like arithmetic and logic operations on the stack. However, the semantic of bytecodes like *new* or *invoke* are too complex for hardware implementation. These bytecodes have to be implemented in a subroutine. One common solution, used in Suns picoJava-II [5], is to execute a subset of the bytecode native and trap on the more complex ones. This solution has an overhead (a minimum of 16 clock cycles in picoJava) for the software trap.

A different approach is used in JOP. JOP has its own instruction set (the so called microcode). Every bytecode is translated to an address in the microcode that implements the JVM. If the bytecode has a 1 to 1 mapping with a JOP instruction, it is executed in one cycle and the next bytecode is fetched and translated. For more complex bytecodes, JOP just continues to execute microcode in the following cycles. At the end of this instruction sequence the next bytecode is requested. This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions. Fig. 2 shows an example of this indirection. The fetched bytecode is used as an index into the jump table. The jump table contains the start addresses of the JVM implementation in microcode. This address is loaded into the JOP program counter for every executed bytecode.

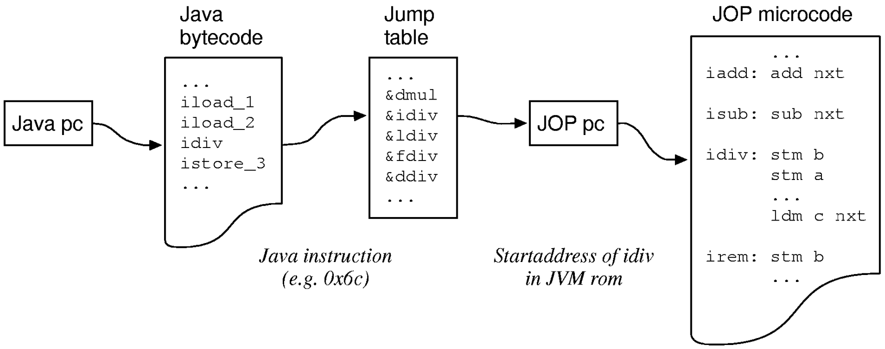


Fig. 2. Data flow for a bytecode instruction

The example in Fig. 3 shows the implementation of single cycle bytecodes and a bytecode as a sequence of JOP instructions. In this example, *ineg* takes 4 cycles to execute and after the last instruction (add) for *ineg*, the first instruction for the next bytecode is executed. The microcode is translated with an assembler to a memory initialization file, which is downloaded during FPGA configuration.

```
iadd:    add nxt    // 1 to 1 mapping
isub:    sub nxt
ineg:    ldi -1     // there is no -val
         xor        // function in the
         ldi 1      // ALU
         add nxt    // fetch next bc
```

Fig. 3. Implementation of iadd, isub and ineg

3 HW/SW Co-design

Using a hardware description language and loading the design in an FPGA, the traditional strict border between hardware and software gets blurred. Is configuring an FPGA not more like loading a program for execution?

This looser distinction makes it possible to move functions easily between hardware and software resulting in a highly configurable design. If speed is an issue, more functions are realized in hardware. If cost is the primary concern these functions are moved to software and a smaller FPGA can be used. Let us examine these possibilities on a relatively expensive function: multiplication. In Java bytecode *imul* performs a 32 bit signed multiplication with a 32 bit result. There are no exceptions on overflow.

Since single cycle multiplications for 32 bits are far beyond the possibilities of current FPGAs, we can implement *imul* with a sequential booth multiplier in VHDL. Three JOP instructions are used to access this function. If we run out of resources in the FPGA, we can move the function to microcode. The implementation of *imul* needs 73 JOP instructions and has an almost constant execution time. JOP microcode is stored in an embedded memory block of the FPGA. This is also a resource of the FPGA. We can move the code to external memory by implementing *imul* in Java bytecode. Bytecodes not implemented in microcode result in a static method call from a special class (com.jopdesign.sys.JVM). The class has prototypes for every bytecode ordered by the bytecode value. This allows us to find the right method by indexing the method table with the value of the bytecode. The additional overhead for this implementation is a call and return with the cache refills.

Table 1 lists the resource usage and execution time for the three implementations. Executions time is measured with both operands negative, the worst-case execution time for the software implementations. Only a few lines of code have to be changed to select one of the three implementations. The showed principle can also be applied to other expensive bytecodes like: *idiv*, *ishr*, *iushr* and *ishl*. As a result, the resource usage of JOP is highly configurable and can be selected for every application.

Table 1. Different implementations of imul

	Hardware [LC]	*Microcode [Byte]*	*Time [Cycle]*
VHDL	300	12	37
Microcode	0	73	750
Java	0	0	~2300

4 Results

Table 2 shows resource usage for different soft-core processors and different configurations of JOP implemented in an EP1C6 FPGA from Altera [1]. All configurations of JOP contain a memory interface to 32-bit static RAM and an 8-bit FLASH for the Java program and configuration data. The minimum configuration implements multiplication and the shift operations in microcode. In the core configuration, these operations are implemented as sequential Booth multiplier and a single-cycle barrel shifter. The typical configuration contains some useful I/O devices such as an UART and a timer with interrupt logic for multi threading. Lightfood [6] is a Java processor targeted at Xilinx FPGA architectures. As a reference NIOS [2], the RISC soft-core from Altera, is also included in the list. Version A is a minimum configuration. Version B adds an external memory interface, multiplication support and a timer.

Table 2. Different FPGA soft cores

Processor	*Resource [LC]*	*Memory [KB]*	*fmax [MHz]*
JOP Minimal	1238	3.25	101
JOP Core	1670	3.25	101
JOP Typical	2036	3.25	100
Lightfoot	3400	1	40
NIOS A	1828	6.2	120
NIOS B	2923	5.5	119

5 Conclusion

Java possesses language features as safety and object orientation that can greatly improve development of embedded systems. However, implementation as interpreter with a JIT-compiler are usually not practicable in resource constraint embedded systems. This paper presented the architecture of a hardware implementation of the JVM. The flexibility of FPGAs and HW/SW co-design makes it possible to adapt the resource usage of the processor for different applications. Predictable execution time of bytecodes enables usage of Java in real-time applications. JOP has been used in three real-world applications showing that it can compete with standard microcontrollers. JOP encourages usage of Java in embedded systems. Full source (VHDL and Java) of JOP can be found at [4]. The main features of JOP are summarized below:

- Small core that fits in a low cost FPGA
- Configurable resource usage through HW/SW co-design
- Predictable execution time of Java bytecodes
- Fast execution of Java bytecodes without JIT-Compiler
- Flexibility for embedded systems through FPGA implementation

References

[1] Altera Corporation. *Cyclone FPGA Family*, Data Sheet, ver. 1.2, April 2003.

[2] Altera Corporation. *Nios Soft Core Embedded Processor*, Data Sheet, ver. 1, June 2000.

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison Wesley, 2nd edition, 1999.

[4] M. Schoeberl. *JOP - a Java Optimized Processor*, http://www.jopdesign.com.

[5] Sun microsystems. *picoJava-II Processor Core*, Data Sheet, April 1999.

[6] Xilinx Corporation. *Lightfoot 32-bit Java Processor Core*, Data Sheet, September 2001.

Hardware/Software Implementation of FPGA-Targeted Matrix-Oriented SAT Solvers

Valery Sklyarov, Iouliia Skliarova, Bruno Pimentel, and Joel Arrais

University of Aveiro, Department of Electronics and Telecommunications, IEETA,
3810-193 Aveiro, Portugal
{skl, iouliia}@det.ua.pt, {a21394, a21540}@alunos.det.ua.pt

Abstract. The paper describes two methods for the design of matrix-oriented SAT solvers based on data compression. The first one provides matrix compression in a host computer and decompression in an FPGA. It is shown that although some improvements have been achieved in this case, there exists a better solution. The second method makes possible to execute operations required for solving the SAT problem over compressed matrices.

1 Introduction

The complexity of FPGAs is not always sufficient for implementing SAT solvers and the required resources have to be partitioned between software running on a general-purpose computer and hardware. This involves multiple data exchange, which is either costly or time consuming. To overcome this drawback the following two methods have been explored: 1) the technique for matrix compression/decompression permitting to reduce the size of matrices in software, to transmit them to an FPGA and to restore the original matrices in hardware; 2) matrix transfer in a relatively simple compressed form and solving in hardware a 3-SAT problem [1, 2] over the transmitted matrices avoiding the decompression step.

2 Matrix Compression/Decompression Techniques

It is known that the SAT problem can be formulated over different models [3] and we will consider for such purposes ternary matrices [4, 5]. They have been chosen because of the reasons reported in [4, 5]. Fig. 1 demonstrates how the considered technique for matrix compression/decompression has been applied. This technique is useful if the following conditions are valid:

- Additional hardware that is necessary to implement the decompressing circuits is reasonable. The latter can be estimated as follows. Let us assume that FPGA resources can be defined as R_{FPGA}, the resources needed to handle matrices with n(2n) columns are $R_n(R_{2n})$. If $R_n \leq R_{FPGA} < R_{2n}$ and after implementing the SAT solver for an n-column matrix the remaining part of the FPGA (i.e. R_{FPGA}-R_n) is sufficient to build the decompressing circuits, then additional hardware is

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 922–926, 2004.

reasonable. The values n and 2n are considered because we assume that matrices are kept in FPGA embedded memory blocks with reprogrammable numbers of inputs and outputs. Incrementing an address size (i.e. adding one input) causes the number of the available memory block outputs to be reduced by a factor of 2.

- Let T be the time of data transfer without the use of compression/decompression and the following expression (see Fig. 1) is satisfied: $T>(T_t+T_h)$.

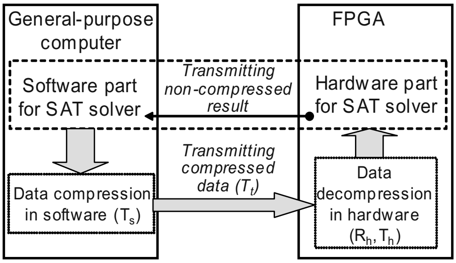

General-purpose computer
FPGA
Software part for SAT solver
Transmitting non-compressed result
Hardware part for SAT solver
Transmitting compressed data (T_t)
Data compression in software (T_s)
Data decompression in hardware (R_h, T_h)

Fig. 1. Using compression/decompression technique for data exchange between hardware/software parts of SAT solver

In order to validate these conditions the following technique has been applied.

- Matrix compression and decompression have been employed for the hardware/software SAT solver considered in [5].
- Data compression/decompression have been provided with the aid of slightly modified Huffman coding, which a) settles repeating coefficients for matrix *don't care* values because as a rule the number of *don't cares* is significantly greater than the number of *ones* and *zeros*; and b) is based on effective recursive procedures considered in detail in [6].

Table 1 demonstrates the results of experiments (for data exchange through the Ethernet for RC200 board [7]), which make possible to estimate the advantages and disadvantages of the technique considered in Fig. 1 for different matrices **M**(n×m), where n is the number of columns and m is the number of rows. A ratio is defined as the size of compressed data divided by the size of non-compressed data and R is the percentage of FPGA resources required for the decompressing circuit. Note that the compression in software can be done in parallel with executing the SAT-solver algorithm in hardware (that is why the last column shows just the time T_h+T_t).

Table 1. The results of experiments for the circuit in Fig. 1

M (n × m)	T (ms)	R	Ratio	T_t (ms)	T_s (ms)	T_h (ms)	T_h+T_t (ms)
128×128	0.903	12%	0.130	0.117	20	0.943	1.060
256×500	3.310	12%	0.139	0.460	220	3.800	4.260
256×256	3.310	12%	0.105	0.347	60	3.715	4.062
256×1000	12.940	12%	0.087	1.126	230	15.380	16.506
256×1500	19.260	12%	0.077	1.483	872	23.024	24.507

An analysis of Table 1 has shown that independently of the good ratio for data compression the total time is increased, i.e. for all the examples $T_h+T_t>T$. Note that the Ethernet-based data exchange is very fast and in case of using other interfaces the compression permits the total time to be shortened. For example, in case of parallel interface the values T and T_t will be increased approximately in 10 times and consequently for all the examples $T_h+T_t<T$. However, we cannot provide significant improvements.

3 Executing Algorithms over Compressed Ternary Matrices

In accordance with [5] a matrix-oriented SAT solver executes operations over rows/columns of a ternary matrix applying the following set of rules:

1. If a column contains just *don't care* values it must be deleted from the matrix.
2. All rows that are orthogonal to an intermediate vector **w** (that incrementally forms a solution) must be removed from the matrix. All columns that correspond to the components of the vector **w** with values 1 and 0 must be deleted from the matrix.
3. If the matrix contains a row with just one component 0 (1) with an index i then the element i of the vector **w** must be assigned the value 1 (0), i.e. the inverted value.
4. If there is a column j in the matrix without values 1 (0) then the element j of **w** can be assigned the value 1 (0).

The considered SAT algorithm implemented in hardware is depicted in Fig. 2. On the one hand the majority of the involved operations are similar to [5]. On the other hand the algorithm has a number of distinctive features which make possible the required hardware resources to be reduced without a degradation of performance. These features are the following:

1. The rule 1 was avoided because it consumes time but does not simplify the operations over matrix rows/columns.
2. Instead of dynamic selection of the next decision variable (column), a static selection has been employed in such a way that all the columns have been sorted by the number of *non-don't care* values in an ascending sequence and the selection has been performed from the first to the last matrix column. Our experience has shown that such predefined sequence minimizes the required FPGA resources and does not reduce performance (in hardware).
3. Any matrix is addressed in the memory by rows, which means that any row can be read/written in one clock cycle.
4. The rules 2 and 3 are sequentially checked for all matrix rows starting from the first row (see Fig. 2). Note that the relevant to the rules 2 and 3 operations can be executed in parallel over any complete row.
5. During sequential operations over rows a vector, which identifies columns containing just *ones* and *don't cares* (or *zeros* and *don't cares*), is incrementally constructed. It permits the rule 4 to be applied after all rows have been examined (see Fig. 2). Thus the matrix transpose is no longer required.

All the other operations are exactly the same as in [5] and we will not replicate them to keep the description short. Note that any compression technique leads to non-equal sizes of different rows (vectors) and this conducts to irregularity of different SAT solver blocks, such as the matrix memory, the combinational circuit, etc. To cope with this problem the following approach has been employed.

1. The software transforms any matrix that is going to be dispatched to an FPGA in such a way that all the matrix rows contain not more than three *non-don't care* values. It is known that such a technique is called 3-SAT [2] and the respective transformation can be done in polynomial time.
2. Any *non-don't care* value (i.e. any *one* or *zero*) is coded by its index followed by the value. For example, the vector [0-------1-1----] is coded as 0000 *0* 1000 *1* 1010 *1*.
3. If the vector has less than 3 *non-don't care* values then the flag containing all *ones* in the respective code is used. For example, the vector [---------------] can be coded as 1111 0 1111 0 1111 0. Thus an r-bit code can be used for any matrix, which has no more than 2^r-1 columns and the number of matrix rows is limited just by available FPGA resources.

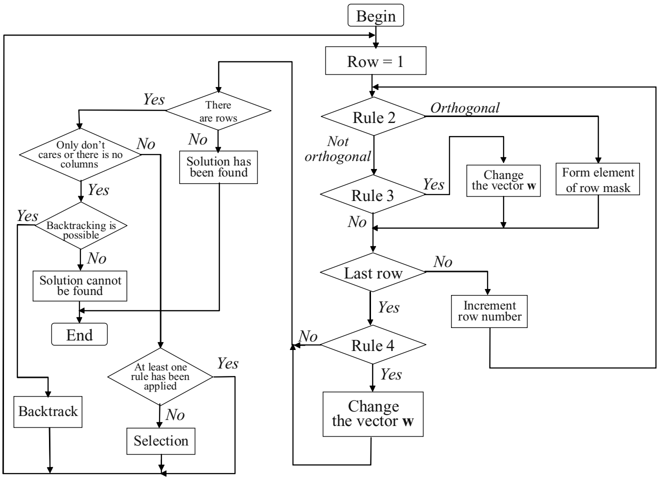


Fig. 2. The SAT algorithm implemented in hardware

4 The Results of Experiments and Implementation Details

A SAT solver, which implements the algorithm in Fig. 2 has been designed in DK2 environment [7] from specification in Handel-C and implemented in Xilinx Virtex-II XC2V1000 FPGA (Celoxica RC200 prototyping board). Mapping, placement, routing and generating the bit-stream for FPGA from an EDIF file created by DK2 have been performed in ISE 6.2.2 of Xilinx. The implemented in FPGA circuits permit to process in hardware matrices containing up to 255 columns and 1500 rows. The clock frequency was set to 45 MHz. As we can see from data in Table 2 the considered circuit has a high performance. In all the examples we have used randomly generated 3-SAT formulae.

Table 2. The results of experiments with the compressed-matrix-oriented SAT solver

Matrix (n × m)	The result	Time for solving the problem in FPGA (s)	% of the used FPGA resources
127 × 128	Satisfiable	0.00087	30
127 × 500	Unsatisfiable	0.127	30
255 × 256	Satisfiable	0.00357	54
255 × 1000	Unsatisfiable	0.195	54
255 × 1500	Unsatisfiable	0.264	54

5 Conclusion

One of the problems inherent to matrix-oriented SAT solvers is the relatively high volume of data that have to be transferred from a host computer to the accelerator, especially in the case of partitioning the problem between general-purpose software and hardware. Due to the complexity of practical SAT problem instances this partitioning is very common. To reduce the influence of data exchange on the total time of computations, two methods have been explored and analyzed.

Acknowledgment. This work was partially supported by the Portuguese Foundation of Science and Technology under grant POSI/43140/CHS/2001.

References

1. J. de Sousa, J. P. Marques-Silva, and M. Abramovici, A configware/software approach to SAT solving, in Proc. 9th IEEE Int. Symp. on Field-Programmable Custom Computing Machines, 2001.
2. P. Zhong, "Using Configurable Computing to Accelerate Boolean Satisfiability", Ph.D. dissertation, Department of Electrical Engineering, Princeton University, 1999.
3. I. Skliarova, A.B. Ferrari, Reconfigurable Hardware SAT Solvers: A Survey of Systems, Proceedings of the 13th International Conference on Field-Programmable Logic and Applications – FPL'2003, Lisbon, Portugal, September, 2003, pp. 468-477.
4. I. Skliarova, A.B. Ferrari, The Design and Implementation of a Reconfigurable Processor for Problems of Combinatorial Computation, Journal of Systems Architecture, Special Issue on Reconfigurable Systems, vol. 49, 2003, pp. 211-226.
5. I. Skliarova, A.B. Ferrari, A Software/Reconfigurable Hardware SAT Solver. IEEE Transactions on VLSI Systems, vol. 12, no. 4, Apr. 2004, pp. 408-419.
6. V.Sklyarov, FPGA-based implementation of recursive algorithms. Microprocessors and Microsystems, Special Issue on FPGAs: Applications and Designs, vol. 28/5-6, 2004, pp. 197-211.
7. Available: http://www.celoxica.com/

The Chess Monster Hydra*

Chrilly Donninger and Ulf Lorenz

Universität Paderborn,
Faculty of Computer Science, Electrical Engineering and Mathematics
Fürstenallee 11, D-33102 Paderborn

Abstract. With the help of the FPGA technology, the boarder between hard- and software has vanished. It is now possible to develop complex designs and fine grained parallel applications without the long-lasting chip design cycles. Additionally, it has become easier to write coarse grained parallel applications with the help of message passing libraries like MPI. The chess program Hydra is a high level hardware-software co-design application which profits from both worlds. We describe the design philosophy, general architecture and performance of Hydra. The time critical part of the search tree, near the leaves, is explored with the help of fine grain parallelism of FPGA cards. For nodes near the root, the search algorithm runs distributed on a cluster of conventional processors. A nice detail is that the FPGA cards allow the implementation of sophisticated chess knowledge without decreasing the computational speed.

1 Introduction

The early chess programs tried to mimic the human chess style. In the 1970s Chess 4.5 [7], however, demonstrated that emphasizing the search speed might be more fruitful. Belle[1], Cray Blitz, Hitech, and Deep Thought[4] were the top programs in the 1980s. From 1992 on, PCs dominated the world of computer chess: ChessMachine, Fritz, Shredder etc. With one well known exception: In 1997, IBM's Deep Blue[3] won the historical 6-game match against Garry Kasparov. This highlight is still of singular quality, although the playing strengths of present top programs seem to cross the borderline beyond the strongest human players. Only one big point is vacant: The final, generally accepted, victory over the humans. The previous Computer Chess World Championship in Graz showed that the race will probably be performed by only four programs: The top four of that championship (Shredder, Fritz Junior, and Brutus/Hydra) scored more than 95% of the possible points against the further 12 participants. Typically, a game playing program consists of three parts: a move generator, which computes all possible chess moves in a given position; the evaluation procedure implements a human expert's chess knowledge about the value of a given position (these values are quite heuristic, fuzzy and limited) and the search algorithm, which organizes a forecast: At some level of branching, the by the game defined game tree (the complete one) is cut. The artificial leaves at this top are evaluated by a heuristic evaluator, because the real values are usually not known, and these values are propagated to the root of the game tree, as if they were real ones. The

* sponsored by: PAL Computer Systems, Abu Dhabi, http://www.hydrachess.com

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 927–932, 2004.

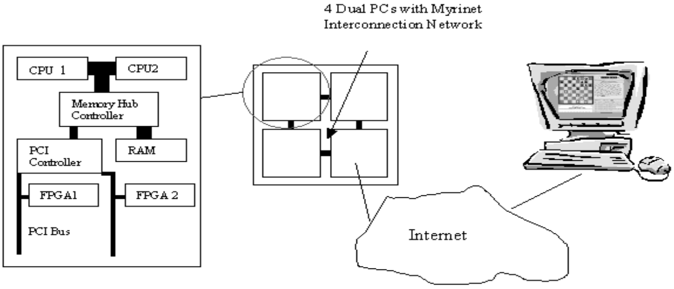


Fig. 1. The System Architecture: 4 Dual Pentium with 2 FPGA cards per node are interconnected with an end-user PC via the Internet.

astonishing observation over the last 40 years in the chess game, and some other games is: the game tree acts as an error filter. The faster, and the more sophisticated the search algorithm, the better the search results!

In professional game playing programs, mostly the Negascout [6] variant of the Alphabeta algorithm [5] is used. The Alphabeta algorithm and its variants, are depth first search algorithms, which use information collected in left parts of the search tree in order to reduce searching time in right parts. The form of the search tree, as well as the efficiency with that it is examined, strongly determines the quality of the search result.

Encoding the move generator, the evaluation procedure and the search algorithm in a hardware design has the following advantages: first, the procedures can be processed in a very few cycles such that the execution speed can significantly be increased. Second, on standard PCs there occurs a tradeoff between the search speed and the implementation of chess knowledge. This tradeoff does not exist in a hardware design. Most evaluation features can be processed in parallel. Therefore, we only need additional space, but no extra time. In contrast to fixed wired hardware, FPGA serves with the advantage that debugging is easier and any improvements can be added instantly.

In this paper, we discuss the design philosophy, general architecture and performance of one of the strongest chess programs in the world. Section two starts with a hardware overview. Then in section 3, we will discuss the software architecture. Firstly the FPGA related stuff, thereafter the parallel search algorithm on the PC side. Last but not least, Section 4 deals with experimental results.

2 System

Hydra uses the ChessBase/Fritz graphical user interface, running on a PC with WindowsXP. It connects with the help of a secure shell via the Internet to our Linux cluster, which itself consists of 4 Dual PC server nodes being able to handle two PCI busses simultaneously. Each PCI bus is supplied with one FPGA card. Each MPI-processes is mapped onto one of the processors and one of the FPGA cards is associated with it, as well. The server nodes themselves are interconnected by a Myrinet network.

3 Software Architecture

Hydra partially runs on PCs, and partially on FPGA cards. The reason is that the bottleneck of the system is the PCI bus. Therefore, on the one hand we cannot make the whole search run on the PC side. On the other hand, it is not clever to put all the search on the FPGA card, because it is much more difficult to develop a sophisticated search algorithm in hardware. We perform the last 3 plies of an n-ply search on the FPGA side, inclusively the quiescence search and all evaluations. Such a depth-3 search is initiated about 100.000 times per second per processor.

3.1 The FPGA

At the present, we use a XilinX based VirtexV1000E card from Alphadata. We use 67 out of 96 BlockRAMs, 9,879 of 12,288 Slices, 5,308 of 12,544 TBUFS, 534 of 24,576 Flip-Flops, and 18,403 of 24,576 LUTs. We can run the FPGA with 33MHz, the longest path consists of 51 logic levels. An upper bound for the number of cycles per search node is 9 cycles. Hydra essentially contains a big piece of combinatorial logic, controlled by a finite state machine (FSM) with 54 states for the search.

The **evaluation** consists of many small features which are summed up by the help of one large adder tree. All features are simultaneously determined. Because of its structural simplicity, we refrain from further details.

A **move generator** is usually implemented in software as a quad-loop: One loop over all piece types, an inner loop over pieces of one type, a more inner loop for all directions in that the piece can move, and the most internal loop for the squares to which the piece can move under consideration of the current direction. This is quite a sequential procedure, especially, when we consider that e.g. taking moves should be sorted to the beginning of the move list. In hardware, there is a nice, fast and small move generator which works completely different. The move generator is, in principle, an 8×8 chessboard. The so called GenAggressor module and the GenVictim module, both instantiate 64 square modules, one for each square. Both determine to which neighbor square incoming signals have to be forwarded. The squares send piece-signals (if there exists a piece on them) resp. forward the signals of the far-reaching pieces. Additionally, each square can output the signal ’victim found’. Then we know that this square is the ’victim’ (i.e. a to-square) of a legal move. The collection of all ’victim found’ signals is the input for an arbiter (indeed a comparator tree), which selects the most attractive, not yet examined victim. The GenAggressor module takes the arbiter’s output as input and sends the signal of a super-piece (a combination of all possible pieces). If e.g. the rook-move signal hits a rook of our own, we will find an ’aggressor’ (i.e. a from-square of a legal move). Thus, many legal moves are generated in parallel. These moves must be sorted and we have to mask, which of them have already been examined. We sort the moves by the help of a comparator tree. The winner is determined within 6 levels of the tree. Sorting criteria are the values of attacked pieces and whether or not a move is a killer move. We refer to [3] for further details about this kind of move generator.

Figure 2 shows a simplified version of the finite state machine that controls the **search** on the FPGA card. The search works as follows. We enter the search at FS_INIT. If there is anything to do, and if a nullmove is not applicable, we come to the start of the full

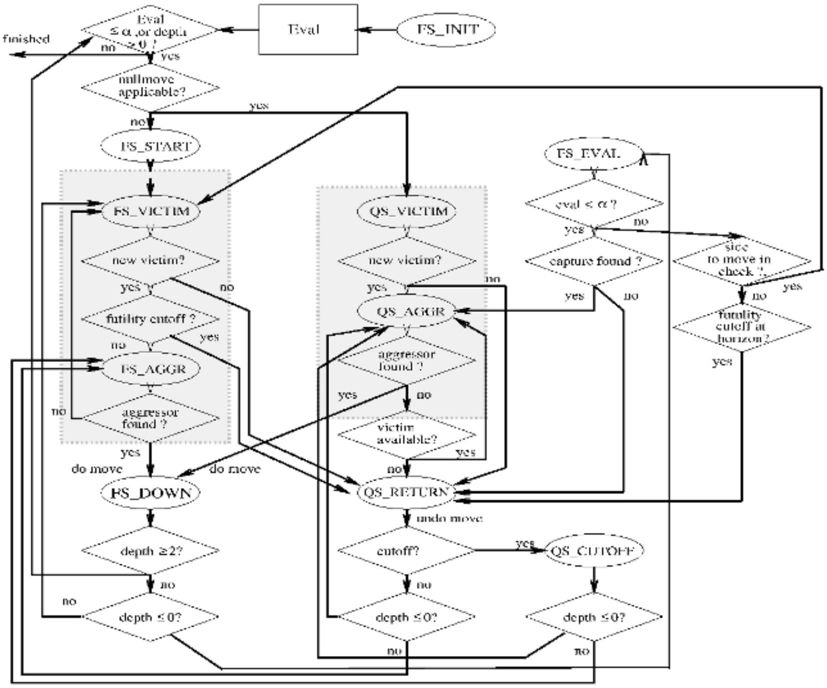


Fig. 2. Flowchart for the Finite State Machine for the Search.

search. After possible increasing the search depth (not shown in figure 2), we enter the state FS_VICTIM, where the output of GenVictim is inspected. If we find a to-square of a valid move and a futility cutoff is not possible, we will reach the state FS_AGGR, where the GenAggressor output is inspected, and if we find a from-square, we will make the next move and reach FS_DOWN. This corresponds to a recursive call of the Alphabeta algorithm with with search window $[\alpha, \alpha + 1]$. If the search remaining depth is greater than 0, we start with looking for a move in the state FS_START. Otherwise we enter the quiescence search, which starts with the examination of the evaluation output. If the evaluation is not greater than alpha, we continue with a capture move, if available. If there is a piece which can be taken, we reach the QS_AGGR state, and if we additionally get a from-square by the help of the GenAggressor module, we will make a further move etc. Moves are unmade and we leave a recursion level, whenever we reach the state QS_RETURN. A recursive algorithm like the Alphabeta algorithm needs a stack for its procedure. The stack in Hydra is realized by six blocks of dual port block RAM. The RAM is organized as 16-bit RAM. Thus we can either write two 16-bit data into the RAM, or one 32-bit word at one point of time. A ply variable which is controlled by the search FSM controls the data flow. Different tables capture different local variables of the recursive search.

3.2 The Distributed Search Algorithm

The basic idea of our parallelization is to decompose the search tree, to search parts of the search tree in parallel and to balance the load dynamically by the help of the work stealing concept. First, a special processor P_0 gets the search problem and starts performing the Negascout algorithm as if it would act sequentially. At the same time, the other processors send requests for work to other randomly chosen processors. When a processor P_i that is already supplied with work, catches such a request, it checks, whether or not there are unexplored parts of its search tree, ready for evaluation. These unexplored parts are all rooted at the right siblings of the nodes of $P_i's$ search stack. Either, P_i sends back that it cannot serve with work, or it sends such a node (a chess position with bounds etc.) to the requesting processor P_j. Thus, P_i becomes a master itself, and P_j starts a sequential search on its own. The processors can be master and worker at the same time. The relationships dynamically change during the computations. When P_j has finished its work (possibly by the help of other processors), it sends an answer message to P_i. The master-worker relationship between P_i and P_j is released, and P_j becomes idle. It again starts sending requests for work into the network. When a processor P_i finds out that it has sent a wrong window to one of its workers P_j, it makes a window message follow to P_j. P_j stops its search, corrects the window and starts its old search from the beginning. If the message contained a cutoff, P_j just stops its work. We refer to [2] for further details.

4 Results

Experiments are performed on the hardware of the Paderborn University. Every processor is a Pentium IV/2.8GHz running RedHat Linux. We measure speedups of our program with the help of the BT2630 test set.

	time(s)	SPE	work %
1	24213	1	0
2	12139	1.99	0
4	6888	3.5	3.6
8	3488	6.94	-1

Speedups on BT2630

We compare the running times and the number of used search nodes of parallel Hydra with the one processor configuration. The speedup (SPE) is the sum of the times of the sequential version divided by the sum of the times of a parallel version. The search overhead is given in percent (work %) seen from 100% of the sequential version.

Tournaments and games: Hydra's predecessor Brutus reached the third rank on the Paderborn Computer Chess Championship in February 2003 and it has won the Lippstadt FIDE Grandmaster Tournament with 9 out of 11 points, reaching a performance of 2768 ELO, which is in the range of the human top players.[1] Internal test matches show that Hydra has made further progress: 140 test games with only 4 processors against Fritz8 and Shredder8, playing on a Pentium 2.4 GHz PC, showed an advantage of more than 110 ELO points. Last but not least, Hydra won the 13^{th} International Paderborn Computer Chess Championships, in the presence of World Champion Shredder8 and Fritz8.

[1] The ELO-system is a statistical measure for the strength of chess players.

5 Conclusion

We presented the professional chess program Hydra, which is one of the top chess programs in the world. It uses the FPGA technology and combines the hardware design methods of Deep Blue with fully dynamic game tree search approaches. The FPGA technology seems to serve with a good compromise, as on the one hand it is possible to make use of hardware parallelism. On the other hand we have no long development cycles. In computer chess it is essential to have the possibility to frequently change and improve the program's code.

References

1. J.H. Condon and K. Thompson. Belle chess hardware. *Advances in Computer Chess III, M.R.B. Clarke (Editor), Pergamon Press*, pages 44–54, 1982.
2. R. Feldmann, M. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. *In proc. of SPAA'94*, pages 94–104, NY, 1994.
3. F-H. Hsu. Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–80, 1999.
4. F-H. Hsu, T. S. Anantharaman, M.S. Campbell, and A. Nowatzyk. *Computers, Chess, and Cognition*, chapter 5 Deep Thought, pages 55–78. Springer Verlag, 1990.
5. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
6. A. Reinefeld. *Spielbaum - Suchverfahren.* Springer, 1989.
7. D.J. Slate and L.R. Atkin. Chess 4.5 - the northwestern university chess program. *Chess Skill in Man and Machine, P.W. Frey (ed), Springer* , pages 82–118, 1977.

FPGA-Efficient Hybrid LUT/CORDIC Architecture*

Ireneusz Janiszewski, Hermann Meuth, and Berhard Hoppe

Fachhochschule/University of Applied Sciences Darmstadt,
Schöfferstraße 3, 64295 Darmstadt, Germany
{janiszew, meuth, hoppe}@fh-darmstadt.de
http://www.fh-darmstadt.de/

Abstract. The paper presents a hybrid architecture for digital polar-to-Cartesian (i.e. phase-to-I/Q) designs. The hybrid LUT/CORDIC architecture allows design partitioning between logic and storage based FPGA resources. FPGA resource utilization, timing and power consumption as well as accuracy of calculated results may be optimised consistently in comparison to conventional pure CORDIC algorithm implementations.

1 Introduction

Implementing algorithms for rotating a two-dimensional (2-D) Cartesian coordinate system is a common requirement in the fields of communication technologies, signal and image processing, for precision measurement instrumentation and the analysis of time varying systems. These transformations may be performed either by using look-up tables for low-latency, low-resolution applications or by algorithmic procedures for medium latency, high precision designs. The basic idea behind the work reported here was to combine the algorithmic as well as the look-up table approach into a unified hybrid hardware model. This approach offers new optimisation potential. Pure look-up or algorithmic approaches fall far below the features obtained with hybrid architectures. The hybrid models reported here are synthesizable and may be ported on arbitrary technologies as a functional block in a compound system design, be it on FPGA or on a chip. Example results of implementations in Flex10k ALTERA devices will be presented in Sec. 4.

2 How Does the CORDIC Work?

Sinusoidal digital signal generation, also referred to as direct digital frequency synthesis (DDFS) regularly requires, in addition to a time-advancing 'phase accumulator', a polar-to-Cartesian transformation, or simpler yet, phase-to-Sine at constant magnitude. Signal detection and information extraction, in turn,

* This work was supported by the German Federal Ministry of Education and Research under contract FKZ17 103 03.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 933–937, 2004.

implies an inverse Cartesian-to-polar (magnitude and phase) coordinate transformation. Dedicated hardware implementations for these tasks are the only choice, if signal and information bandwidths, agility, size, power considerations and real-time put-through rule out software-based standard signal processing schemes. Aside from direct signal-value look-up from table, hardware algorithms, like COordinate Rotation DIgital Computer (CORDIC), may meet these novel needs for high precision digital sinusoidal signal generation from a phase input. The CORDIC algorithm may operate in either a vector-rotation mode, or an angle-accumulation mode, so-called *rotating* or *vectoring* mode respectively. In its *rotating* mode, frequently referred to only as *CORDIC*, CORDIC yields both sine (or y) and cosine (or x) values for a given input phase angle ϕ. The CORDIC algorithm was developed by Jack E. Volder in 1956 [1]. John Stephen Walther [2], presented the general Unified CORDIC equations capable of performing computations in either circular, hyperbolic or linear coordinate systems. Both *rotation* and *vectoring* CORDIC modes result in the same set of iterative equations:

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i \cdot y_i \cdot 2^{-i} \\ y_{i+1} &= y_i + \sigma_i \cdot x_i \cdot 2^{-i} \\ z_{i+1} &= z_i - \sigma_i \cdot \alpha_i \end{aligned} \tag{1}$$

In geometrical terms, executing this algorithm thus amounts to successive rotations by decreasing angle steps of $\alpha_i = \arctan(2^{-i})$. Arithmetically, it amounts to successive summations and binary shifts (i.e. divisions by 2), i.e. a very hardware-efficient scheme involving no multiplication. The positive/negative sense of each rotation must be chosen such that the procedure converges. This is assured by the parameter σ_i, which for the *rotating* mode is defined as $\sigma_i = \operatorname{sign} z_i$, where $\operatorname{sign} z_i = 1$ if $z_i \geq 0$, else $\operatorname{sign} z_i = -1$, implying that the iteration converges against $z_\infty \to 0$. In *rotation* mode, CORDIC provides cosine (x-path) and sine (y-path) of an input angle, if the initial value of the x input is 1, and of the y input is 0 with the initial z being the input angle. The results contain the CORDIC *algorithm gain* factor $G_N = 1/K_N$, where correspondingly, K_N, is a *distortion* or *scaling factor*, defined by $K_N = \prod_{i=0}^{N-1} 1/\sqrt{1+2^{-2i}}$. This scaling factor does not depend on the actual value sequence of $\sigma_i \alpha_i$.

3 Phase-to-Sine at Constant Magnitude Transformer

The CORDIC algorithm in the *rotating* mode is suitable for sinusoid signal generation. Direct Digital Frequency Synthesis (DDFS) based on Numerically Controlled Oscillators (NCO) is an established method for generating quasi-periodic sinusoid signals. A typical NCO consists of an overflowing phase accumulator (PA), generating the phase sequence. The subsequent phase summing block (PSB) may perform in-flight phase jumps, and finally function generator (FG) produces the sine and cosine amplitude values for the actual phase. Via the PSB, the PA output addresses the FG, which in most applications is a look-up table (LUT), yielding amplitudes of bit width *FGO*. The resolution *AW* of the internal phase, a, is generally much coarser than the *FW* precision at the output of the PA block (phase truncation). With increasing output resolution of FG, a LUT based implementation may become prohibitively large. At the expense of output multiplexer the LUT entries may be restricted to the first octant of the unit circle. LUT size may further be compressed, implying additional hardware, which also could introduce additional spurious contributions to the output spectrum. High-resolution FGs in contrast may be implemented with LUT-free hardware by means of CORDIC algorithms. However CORDIC is an iterative algorithm, its obvious drawback being the delay or latency in pipelined implementations. A CORDIC/LUT hybrid architecture [3],[4],[5] will essentially remedy this drawback (see Fig 1). The standard CORDIC iterations of equs. (1) start with index $i_{start} = 0$, thereby scanning the polar angle space between $-\pi/2$ and $\pi/2$. Again, due to symmetry, only phase values from the first octant have to be considered. Exploiting symmetry leads to a starting index $i_{start} = 2$. By further restricting the CORDIC iterations, fewer iterations and fewer pipeline stages are required to achieve a given accuracy in the amplitude outputs. The algorithm then has to be started in the respective wedge by suitable x and y initial values. These values must come from an additional LUT. As size and power budgets for LUTs on one hand and CORDIC on the other scale differently with amplitude resolution, there is potential for optimising hybrid NCO designs by suitably balancing LUT size against the number of CORDIC stages. Figure 1 presents a block diagram of such a *hybrid function generator*. Here, *FS* is the index number i_{start} of the first activated CORDIC stage.

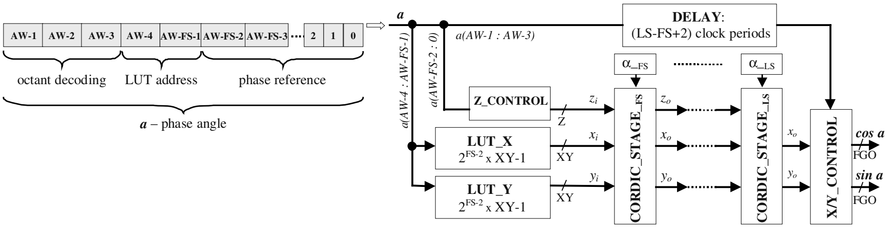


Fig. 1. Hybrid FG block scheme based on CORDIC in *rotating* mode. The inlay shows the bit assignment for a in the case of $FS = 4$. [4]

The relevant addresses for the LUT are obtained by decoding the corresponding bits from input phase word. Two look-up tables LUT_X and LUT_Y, provide the initial inputs for the first activated CORDIC stage. The final amplitudes from x- and y-paths are mapped to the target octant by X/Y_CONTROL.

4 FPGA Implementation

There are various architectures being applied for a CORDIC implementation. Each is suitable for different design requirements. A survey of architectures with focus on FPGA as target technology, and also potential CORDIC applications can be found in reference [6]. Depending on the actual application and design constraints, an optimum solution could be an iterative architecture with a small resource demand or even a more conservative bit-serial iterative architecture. Although such architectures are very efficient in resource exploitation, they, however, cannot provide as high data-throughputs as unrolled architectures and especially pipelined unrolled architectures. Aside from the actual architecture applied, there are many different factors influencing the final result. Such contributors may be the design tools used, libraries of primitive components and IPs and macros, the number coding used, and finally the target technology and the technology or device provider.

In the Table 1 implementation results are juxtaposed for an NCO design based on both pure CORDIC and our hybrid LUT/CORDIC architecture, as presented in Sec. 3. All these NCO design implementations are equivalent in that they produce quadrature, 16-bit sine and cosine data sequences based on a 31-bit frequency control word (FW=31). The 32-bit phase accumulator is truncated to 20-bit internal phase word (AW=20) and followed by a 3-bit phase summing block (PW=3). The CORDIC-based function generator uses 19-bit x- and y-paths and returns results after 18 CORDIC iterations (i.e. the last stage of the CORDIC pipeline has the index LS=17).

Table 1. Results for implementations of NCO designs based on CORDIC

			Flex10k30	Flex10k50	Flex10k70
Pure CORDIC	scheme	LCs	-	2058	2093
		EABs	not fitted	0	0
		Fclk [MHz]	-	12,64	11,00
	FS01	LCs	1504	1504	1520
		EABs	0	0	0
		Fclk [MHz]	35,58	40,65	34,96
Hybrid CORDIC	FS10	LCs	820	820	822
		EABs	5	5	5
		Fclk [MHz]	46,08	46,72	42,73
	FS11	LCs	-	749	751
		EABs	not fitted	9	9
		Fclk [MHz]	-	46,72	44,44

The columns labeled Flex10k30, Flex10k50 and Flex10k70 present the implementation results with ALTERA devices EPF10K30RC240-3, EPF10K50RC240-3, and EPF10K70RC240-2, respectively, as a targets. The three row categories specify the number of utilized design logic cells (LCs), embedded array blocks (EABs), and the maximum clock performance (Fclk [MHz]) reached. These categories are specified for a schematic based design (scheme), and three VHDL-based designs differing in the first activated CORDIC stage FS=1 (full CORDIC), FS=10, and FS=11 respectively. Note that last two implementations save 10 and 11 from 18 total CORDIC iterations and, at the same time, reduce latency of the pipelined design by the same factor. The two top rows correspond thus to a pure CORDIC algorithm implementation, and the remaining two to the hybrid LUT/CORDIC architecture presented here. The schematic-based design is built from ALTERA-provided primitives and macros. The logic synthesis of the design was performed with ALTERA's Max+plusII tool. The same tool was used for physical synthesis in all implementations. For logic synthesis from VHDL, Synopsys FPGA Compiler II was used.

The designs with hybrid architecture utilize dedicated blocks of memory for implementing the LUT while use of logic cells is substantially decreased. The LUTs thus save or rather take over the functionality of the eliminated initial CORDIC stages. Properly balancing the degree of hybridisation by simply tuning the FS parameter in the VHDL design permits to control the partitioning of the design between memory block and logic cell resources. Increasing the first activated CORDIC stage *FS* entails higher memory block utilization but simultaneously lowers logic cell utilization. Significant latency reduction and increase of maximum switching speed will result. It was also shown in [4],[5] that power performance as well as accuracy may be improved with the hybrid architecture. However, increasing the hybridisation degree has its limits, depending on application, target technology and others constraints. For *FS*=11 and target device EPF10K30RC240-3, e.g., the design cannot be fitted. Here, all or almost all memory blocks are occupied by LUTs. Comparing the results for the two pure CORDIC implementations, they differ in occupied LCs and in maximum switching speed performance, depending on the design flow.

5 Conclusions

Our evaluation of hybrid implementations showed that this approach outperforms traditional pure algorithmic (CORDIC) as well as pure look-up table solutions (LUT) in terms of layout/FPGA area and power budget, especially for fast-switching designs. The hybrid schemes may considerably improve on the latency of a pipelined implementation, as well as on obtainable switching speeds. They also allow for control over FPGA resources partitioning.

References

1. Volder, J.E.: The CORDIC computing technique. In: Proc. of the Western Joint Computer Conference. (1959) 257–261
2. Walther, J.S.: A unified algorithm for elementary functions. In: Proc. of the Spring Joint Computer Conference. (1971) 379–385
3. Dachroth, M., Hoppe, B., Meuth, H., Steiger, U.H.: High-speed architecture and hardware implementation of a 16-bit 100-MHz numerically controlled oscillator. In: Proc. of the ESSCIRC, Den Haag, Holland (1998) 456–459
4. Janiszewski, I., Hoppe, B., Meuth, H.: VHDL-based design and design methodology for reusable high performance direct digital frequency synthesizers. In: Proc. of the 38th Design Automation Conference (DAC), Las Vegas, NW, USA (2001) 573–578
5. Janiszewski, I., Hoppe, B., Meuth, H.: Numerically controlled oscillators with hybrid function generators. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control **49** (2002) 995–1004
6. Andraka, R.: A survey of CORDIC algorithms for FPGA based computers. In: Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA-98), New York, ACM Press (1998) 191–200

A Multiplexer-Based Concept for Reconfigurable Multiplier Arrays

Oliver A. Pfänder, Roland Hacker, and Hans-Jörg Pfleiderer

University of Ulm, Microelectronics Department,
Albert-Einstein-Allee 43,
D-89081 Ulm, Germany
{oliver.pfaender, hans-joerg.pfleiderer}@e-technik.uni-ulm.de

Abstract. In this paper, a multiplexer-based concept for creating a run-time configurable array of multipliers capable of accommodating different input data word lengths is presented. In our approach, each element of a $m_1 \times m_2$ multiplier array is a parallel-parallel multiplier itself, each again comprising a number of basic arithmetic primitive cells and featuring multiplexers as controllable interconnects. Also, we distinguish between multiplier elements for unsigned and signed numbers which differ in algorithm and design. Diverse architectures are being reviewed and an estimate of hardware complexity and area consumption is given.

1 Introduction

Hardware multipliers are among the most intricate and area consuming pieces of circuit design [1]. The underlying algorithms and their corresponding hardware implementations have gone through a comprehensive optimization process and reached a high level of maturity [2]. Even today, in the age of dynamically reconfigurable hardware, it is beneficial to embed a sophisticated multiplier structure as part of an ASIC or as an FPGA sub-block, as shown by the 18×18-bit signed two's complement multiplier in modern XILINX® Virtex™-II platform FPGAs [3].

However, depending on the application, it can be advantageous to have an array of smaller multipliers rather than a monolithic block with large bit width. A flexible approach with reconfigurability at run time can help to save hardware resources and increase the efficiency of arithmetic operations, for example providing either a higher throughput at low levels of precision or – when needed – a higher precision by grouping multiple elements together [4]. In this paper, we will discuss what overhead is needed to create a multiplier element with a connectivity option, thus enabling each element to either work separately and in parallel with others, or a group of elements to be concatenated to form a superior multiplier with increased word length.

2 Concatenation Concept

To achieve the required uniformity, scalability and connectivity of a $n_1 \times n_2$-bit multiplier element, we utilize a parallel array structure with carry-ripple technique [5] as a highly regular core. The basic arithmetic primitive is composed of an AND gate to calculate the

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 938–942, 2004.

partial product and a full adder. Multiplexers with corresponding control input signals are placed both on top and on the left-hand side of the structure. By introducing a few supplementary gates, we are able to amend the multiplier element to handle signed numbers directly, thus eliminating any conversion steps when the task is sourced out to the environment [6].

2.1 Signed Numbers

To realize the multiplication of signed numbers in signed-magnitude representation, the basic scheme can obviously be lead back to an unsigned multiplication with a separate handling of the sign bits. The sign handling can either be performed outside of the structure or be embedded, both incorporating only little overhead compared to the unsigned case. The result is negative and its MSB is '1' when the operands have opposite signs, therefore the result's sign bit can be calculated from the input sign bits using the XOR function. On the other hand, the more practical approach to accommodate signed operands is to use two's complement numbers, also because of the unique representation of the null operand [4]. To account for the sign extension in the parallel-parallel multiplication scheme with uniform basic cells [2], an extra column of basic cells is added to the structure. One proposed structure for a configurable $n \times n$-bit multiplier element for two's complement numbers has been previously published in [7]. Depending on the element's position in a concatenated array, the additional column of basic cells lies idle for most cases. But these cells can be reused and configured to act as an additional standard column and thus increment the multiplicand **A**'s bit width by 1. The structure of an $[(n_1+1) \times n_2]$-bit multiplier element, which can be configured either to perform a sign extension or calculate with an additional bit, is depicted in figure 1.

Controllable inverters are shown as multiplexers with a negated input and behave identical to an XOR gate. When the multiplier is to accept partial sum signals from another element (S_IN), the vertical control signal CTRL_V is set to '1', also when there are partial sum signals from an adjacent element (PSL_IN), the horizontal control signal CTRL_H is set to '0'. The multiplier is also able to accept additional carry signals from an adjacent element (C_IN) and will supply outward carry signals when the horizontal connectivity is enabled. According to [8] and [4], the value of the product **M** of **A** and **B** in two's complement number representation can be re-written as

$$M_v = a_{n_1-1} b_{n_2-1} 2^{n_1+n_2-2} + \sum_{i=0}^{n_1-2} \sum_{j=0}^{n_2-2} a_i b_j 2^{i+j} \tag{1}$$

$$+2^{n_1-1} \left(-2^{n_2} + 2^{n_2-1} + 1 + \sum_{j=0}^{n_2-2} \overline{a_{n_1-1} b_j}\, 2^j \right) \tag{2}$$

$$+2^{n_2-1} \left(-2^{n_1} + 2^{n_1-1} + 1 + \sum_{i=0}^{n_1-2} \overline{a_i b_{n_2-1}}\, 2^i \right). \tag{3}$$

A modified Baugh–Wooley multiplier structure [8] representing this algorithm and the circuitry for achieving the connectivity is shown in figure 2. The perimeter cells in the left column and the bottom row are modified and have a controllable inverter inside to

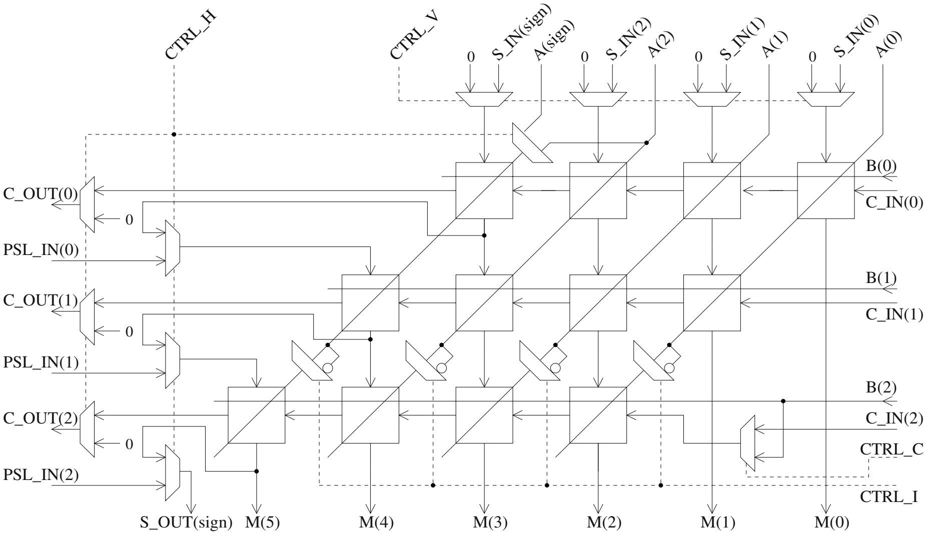


Fig. 1. A configurable multiplier structure with switchable sign extension

negate the partial product, denoted by a black square in the cell's bottom right corner. The cell in the bottom left position in figure 2 acts as a special cell only when one of the CTRL_I signals is '1', but not both at the same time, which is realized by the XOR.

A correcting term representing both $+1$ parts in (2) and (3) must be added to the multiplication result in order to get the correct value [9,4], since negating a two's complement number is done by inverting all bits and adding 1. Having a result bit width of $n_1 + n_2$, this correcting term **X** has a '1' at the positions $x(n_1 + n_2 - 1)$, $x(n_1 - 1)$ and $x(n_2 - 1)$. In the structure of figure 2 symbolizing the square case $n_1 = n_2$, this is lead back to two additional '1's: One at the MSB $x(2n - 1)$ and one at the position $x(n)$. Adding '1' is the same as negating the appropriate carry signal [10], again implemented by two controllable inverters addressed by CTRL_C_1 for the $(n + 1)$st bit and CTRL_C_2 for the MSB.

2.2 Estimation of Hardware Overhead

In order to compare the presented architectures, we acquire the number of transistors in either case. This metric enables us to compare the relative area usage of each variant objectively [11]. Considering the necessary number of transistors for each full adder, multiplexer and gate as given in [1], we get the estimated numbers compiled in table 1. The dominating factor is the actual bit width of each multiplier element and its according number of basic cells, thus resulting in a quadratic dependency on n when the element's architecture is symmetrical ($n_1 = n_2 = n$). In case of the two's complement multiplier shown in figure 1, however, the extra column for the sign extension mainly entails n_2 extra basic cells. When $m_1 \times m_2$ multiplier elements are concatenated, the total number of transistors becomes $T_{x,\mathrm{con}} = m_1 m_2 \cdot T_x$. The bit width has to be chosen carefully,

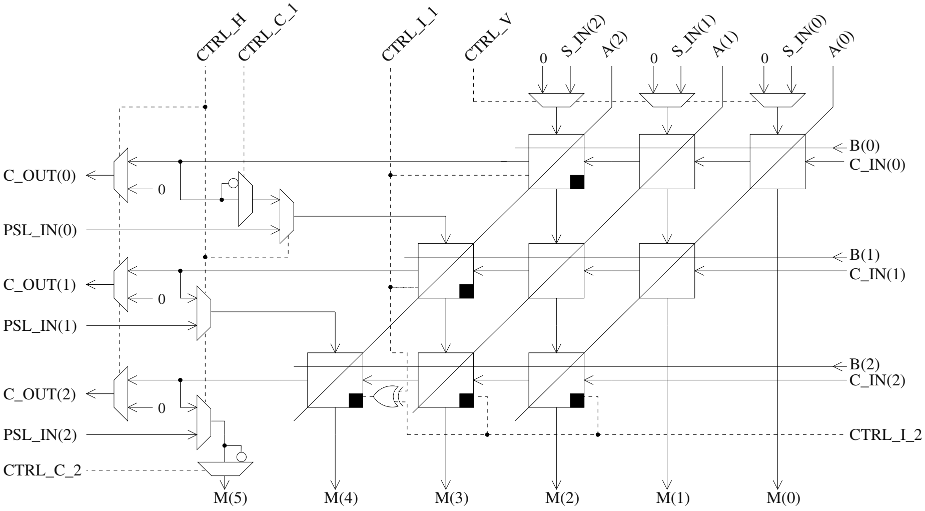


Fig. 2. A modified Baugh–Wooley multiplier structure with switchable special cells

and a reasonable tradeoff between core complexity and concatenation overhead has to be made. This decision of course depends heavily on the application.

3 Conclusion and Further Work

In this paper, we have presented a flexible concept for creating a run-time configurable array of multiplier elements, featuring an interconnection approach based on multiplexers. The major advantage of the parallel-parallel scheme is the highly regular structure and thus the reusability of its basic arithmetic primitives. A variety of signed and unsigned multiplication schemes has been discussed. When the application does not explicitly require two's complement numbers, our signed-magnitude implementation with an unsigned core and an extra XOR gate offers the least overhead. In case of two's complement numbers, our modified Baugh–Wooley architecture is highly recommendable, since it is very area efficient and offers a more regular structure than the variant using an extra column of basic cells. A possible application for our proposed architectures is an embedded dedicated multiplier/coprocessor block, providing high throughput both in stand-alone operation and as a concatenated multiplier. The use of a dedicated yet scalable multiplier helps to save hardware resources for other tasks [11] and the reusability of its modules can significantly increase the design efficiency and its performance. Reconfigurable multipliers with a variable number of bits will become essential e. g. for the coefficient calculation for holographic beam steering applications in future intelligent optical networks [7]. The upcoming task is to develop a concept for an advantageous realization of the surrounding circuitry to ensure an efficient data distribution and collection, in the style of programmable interconnections and hierarchical routing resources as shown in [3] or dedicated bus structures as proposed in [6]. Then, comparisons to other architectures in particular usage scenarios can be made.

Table 1. Estimates for transistor numbers T_x calculated for $n_1 = n_2 = n$ multiplier elements, connectivity overhead compared to the non-configurable core structure and area usage reported in [10] using 0.13μm/1.2V process and layouts based on transmission gates

	$\approx T_x$			$\approx$ overhead (%)			$\approx$ area usage (μm^2)		
n (bit)	4	8	16	4	8	16	4	8	16
unsigned	462	1686	6438	18.5	9.3	4.7	505.4	2077.6	n/a
signed-magn./ext. XOR	470	1694	6446	20.5	9.9	4.8	n/a	n/a	n/a
signed-magn./int. XOR	500	1724	6476	28.2	11.8	5.3	n/a	n/a	n/a
2's compl./reg. sign ext.	602	1954	6962	54.4	26.7	13.2	703.0	2813.4	n/a
2's compl./conf. sign ext.	616	1968	6976	57.9	27.6	13.4	n/a	n/a	n/a
2's compl./mod. B.-W.	550	1838	6718	41.0	19.2	9.2	592.5	2446.7	n/a

Acknowledgements. The authors would like to thank Professor Kamran Eshraghian and Sholeh Eshraghian of Edith Cowan University, Perth, Australia, and Dr. Fang Hongbing of Nanjing University of Science & Technology, Nanjing, China, for discussion, assistance and contributions. Their help is gratefully acknowledged and very much appreciated.

References

1. Weste, H. E.; Eshraghian, K.: "Principles of CMOS VLSI Design". Reading, MA, Addison–Wesley, 1994^2
2. Parhami, B.: "Computer Arithmetic: Algorithms and Hardware Designs". New York, Oxford University Press, 2000
3. Xilinx$^{®}$ Corporation: Virtex^{TM}-II Platform FPGAs; Product Specification, Detailed Description DS031-2 (v3.1) 10/14/2003
4. Bermak, A.; Martinez, D.; Noullet, J.-L.: "High-Density 16/8/4-bit Configurable Multiplier". In: IEE Proc. Circuits Devices Systems, Vol. 144, No. 5, 10/1997, pp. 272–276
5. Hwang, K.: "Computer Arithmetic – Principles, Architecture, and Design". New York, John Wiley & Sons, 1979
6. Koutroumpezis, G. et al: "Architecture Design of a Reconfigurable Multiplier for Flexible Course-Grain Implementations". In: FPL 2002, Montpellier (F), 09/2002, pp. 1027–1036
7. Eshraghian, S.; Lachowicz, S.; Eshraghian, K.: "Ultra High Bandwidth Image and Data Processing using 3-D Vertically Integrated Architectures". In: SCI 2003, Orlando, FL, 07/2003, Proc. Vol. X, pp. 189-195
8. Baugh, C. R.; Wooley, B. A.: "A Two's Complement Parallel Array Multiplication Algorithm". In: IEEE Trans. Computers, Vol. C-22, 1973, pp. 1045–1047
9. Hatamian, M.; Cash, G. L.: "A 70-MHz 7-bit$\times$8-bit Parallel Pipelined Multiplier in 2.5-μm CMOS". In: IEEE J. Solid-State Circuits, Vol. SC-21, No. 4, 08/1986, pp. 505–513
10. Eshraghian, S.: "Implementation of Arithmetic Primitives Using Truly Deep Submicron Technology (TDST)". Master Thesis, Edith Cowan Univ., Perth, Australia, 02/2004
11. Haynes, S. D.; Ferrari, A. B.; Cheung, P. Y. K.: "Flexible Reconfigurable Multiplier Blocks suitable for enhancing the Architecture of FPGAs". In: IEEE 1999 Custom Integrated Circuits Conference, San Diego, CA, 05/1999, Proceedings, pp. 191–194

Design and Implementation of a CFAR Processor for Target Detection

César Torres-Huitzil, Rene Cumplido-Parra, and Santos López-Estrada

Computer Science Department, INAOE, Apdo. Postal 51 & 216
Tonantzintla, Puebla, México
{ctorres, rcumplido, santosle}@inaoep.mx

Abstract. Real-time performance of adaptive digital signal processing algorithms is required in many applications but it often means a high computational load for many conventional processors. In this paper, we present a configurable hardware architecture for adaptive processing of noisy signals for target detection based on Constant False Alarm Rate (CFAR) algorithms. The architecture has been designed to deal with parallel/pipeline processing and to be configured for three versions of CFAR algorithms, the Cell-Average, the Max and the Min CFAR. The architecture has been implemented on a Field Programmable Gate Array (FPGA) with a good performance improvement over software implementations. Results are presented and discussed.

1 Introduction

The extraction of targets from signals is a complex task due to the uncontrolled and noisy environmental conditions. Adaptive digital signal processing techniques are often used to remove noise and to enhance the detectability of targets in many situations. For instance, in radar applications, the backscattering amplitude of the radar signal is used for target detection and it is usually assumed that a high magnitude of the backscattering radar signal comes from targets [1]. Since the background is not uniform and the backscattering amplitude from the background fluctuates due to noise, an adaptive scheme is required to extract targets according to a varying reference threshold and to maintain a constant false alarm rate.

The CFAR algorithms have been widely used to extract targets from the background under noisy environments in application areas such as image processing, medical engineering, power quality analysis, and sonar and surveillance systems, among others [2][3]. Although the theoretical aspect of CFAR detection is advanced, there are not practical hardware applications because the high computational requirements involved in applications such as radar signal processing.

The rest of the paper is organized as follows. Section 2 provides the theoretical foundation of the CFAR algorithm. Section 3 presents a data parallelism analysis of CFAR algorithms and details of the proposed hardware architecture. In section 4 the FPGA implementation and experimental results are presented. In section 5, a brief discussion on the performance improvements is presented. Finally, section 6 presents the concluding remarks.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 943–947, 2004.

2 CFAR Algorithm

The Cell-Averaged CFAR (CA-CFAR) is the most common CFAR detector used for target detection. The CA-CFAR detector is used to regulate the false alarm probability to a desired level in varying background environments through averaging. Figure 1 shows a block diagram of the CA-CFAR algorithm structure. In CA-CFAR detectors, a reference window of *N* samples which surround the cell or data under test is taken to compute the average value and some guard cells are incorporated in order to avoid targets that are close one to each other affect noise estimation [3][4].

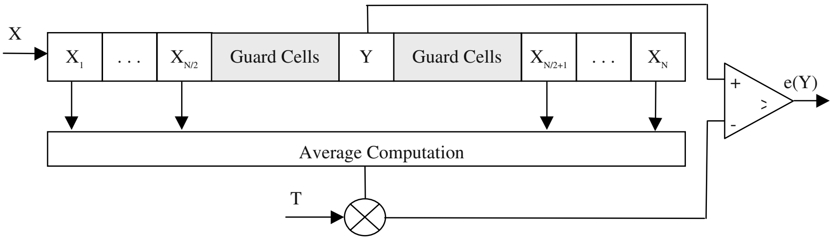


Fig. 1. Block diagram of a CA-CFAR processor. The main components of the processor are registers, a multiplier, an average computation module and a comparator

The average computation module sums up independently the data samples of both sides of the cell under test and computes their average, S_L and S_R, left and right respectively. Both average values are combined to estimate the local noise level in the signal. Three modalities, the average, the maximum and the minimum, are used for this purpose and they are defined according to equation 1. The noise estimation is multiplied by a scaling factor T and finally compared to the value of the cell under test *Y*. If the values of the cell under test exceed the computed value, then target detection is declared. The CFAR detector adapts the threshold automatically to the local information on the background noise. The scaling factor *T* sets up a desired false alarm probability and it is related to the noise distribution in the environment.

$$Y_{AC} = \begin{cases} \frac{1}{2}(S_L + S_R) \\ \max(S_L, S_R) \\ \min(S_L, S_R) \end{cases} \qquad e(Y) = \begin{cases} 1, & \text{if } \mathrm{Y} \geq \mathrm{T} \times \mathrm{Y}_{\mathrm{AC}} \\ 0, & \text{if } \mathrm{Y} < \mathrm{T} \times \mathrm{Y}_{\mathrm{AC}} \end{cases} \qquad (1)$$

3 CFAR Hardware Implementation

Let *X* be the raw data samples of the signal to be processed and *n* the number of reference cells in the CFAR detector. For the sake of simplify but without lost of generality, guard cells are not included in the explanation. Also, let consider a sequence of reference data samples around a cell under test as one-dimension windows shown as rectangles in figure 2. Each rectangle includes data from the reference cells around X_i, X_{i+1}, and X_{i+2}. As shown in figure 2, the windows share data and each time that the window slides leftwards one position on data, all its data samples, except the data located on the edges, belong to the domain of the next window. Therefore, the data dependencies and sharing can be exploited to reuse previous partial results. Once a window has been processed, preceding results can be used to compute the result of the next window without the need of recalculating partial result over the entire domain just by inserting and deleting values from the window boundaries.

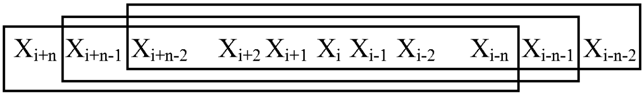


Fig. 2. A graphical view of the data dependencies for three adjacent reference data sets. A data set is obtained by sliding the previous data set and by inserting and deleting one data at the edges of the previous data set

If guard cells are included, it is still possible to exploit the data sharing through parallelism since data sets on each side of the cell under test exhibit the same data sharing and both data sets can be processed concurrently by two processing elements. The data sharing in CFAR algorithms can be efficiently handled through pipelining and systolic processing due to the regularity of computations [4].

A block diagram of the proposed architecture is shown in figure 3. The main components of the architecture are: a shift register, two processing elements for accumulating partial results, called APEs, and a processing element that performs the thresholding, called CTPE. The length of the shift register is equal to the number of reference cells *NRC* plus the number of guard cells *NGC* plus one cell under test.

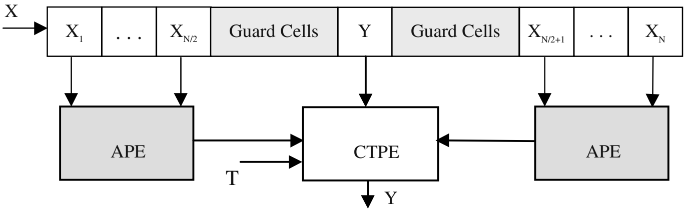


Fig. 3. Block diagram of the main core of the CFAR architecture. Two APEs compute the average of the reference cells and a CTPE computes the thresholding operation

Figure 4 shows a block diagram of the internal structure for the processing elements APE and CTPE. The APE is composed of an accumulator and a substracter. Two APEs computes the accumulation of the values of the reference cells, S_L o S_R in equation 1. The APE has three inputs: X_R the data that is inserted in a new reference window, X_D the data that is deleted from the previous accumulation, and E the signal that inhibits the activity of the accumulator in the latency period.

The CTPE is composed of an ALU-like sub-module, a multiplier and a comparator. The ALU-like sub-module provides three modalities for computing the thresholding: the average, the maximum and the minimum of the partial sums S_L and S_R. Thus, t architecture performs three modalities of CFAR algorithms, the CA-CFAR and the so called MAX family of CFAR detectors [3]. A multiplier scales up the ALU result with a fixed threshold *T* and the comparator decides if a target is present or absent.

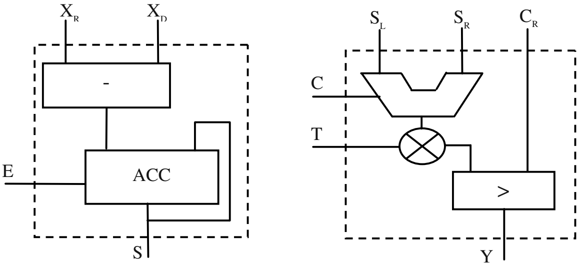


Fig. 4. Block diagram of the internal structure of the Processing Elements, a) main components of the APE and b) main components of the CTPE

In the proposed architecture, on each clock cycle, data moves rightwards and after a latency period, the APEs accumulate data of the reference cells. At the start of processing data, the APEs are inactive but after *NRC/2+NGC+1* clock cycles the APEs operates continuously. *NRC* and *NGC* stand for the number of reference cells and the number of guard cells employed in the current architecture, respectively.

4 FPGA Implementation and Results

The proposed architecture was modeled using the VHDL Hardware Description Language [5] and synthesized with Xilinx ISE targeted for a XC2V250 Virtex-II device. Table 1 summarizes the FPGA hardware resource utilization and timing performance. The default configuration of the CFAR processor uses 12-bit for data, 32 reference cells and 8 guard cells which is a common configuration used for most radar-based applications with a good performance-accuracy trade-off [1]. The internal temporal data in the accumulator uses 18-bit precision established for the worst case..

5 Discussion

The proposed architecture produces an output result on each clock cycle after the latency period and performs seven arithmetic operations concurrently. The latency period is proportional to the number of reference cells *NRC*, and the number of the guard cells *NGC* around the cell under test UCT. The latency arises at the start of processing since the pipeline or shift register must be full in order to output a result.

The architecture provides a throughput of 840 Millions of Operations per Second (MOPs). For instance, the architecture execution time to perform the CFAR processing in radar-based applications is 140 milliseconds on a data set 4096×4096 samples, using 32 and 8 reference and guard cells, respectively [1]. The architecture performance is over 18 times faster than the required theoretical processing time of 2.5 seconds. The software implementation of the CFAR algorithm was carried out in Visual C++ targeted to a personal computer with a Pentium IV processor running a 2.4 GHz and 512 Mbytes of main memory. The processing time for the CFAR algorithm on this platform is 1.2 seconds. In the software implementation a similar scheme for data reuse and optimization to the FPGA implementation was used. Also, the CFAR algorithm, coded in C language, was targeted to a TMS320C6203 DSP device from Texas Instruments. The processing time obtained for the DSP implementation was about 420 milliseconds. The performance improvement of the proposed architecture is about 10 times than the software implementation but a less extent improvement is obtained when compared to the DSP implementation.

Table 1. Synthesis summary and timing for the FPGA implementation of the CFAR processor.

Synthesis summary for the CFAR processor targeted for a XC2V250-6FG456 Virtex-II device	
Number of Slices	331 (21%)
Number of 4-input LUTs	177 (5%)
Number of flip-flops	540 (17%)
FPGA Occupation percentage	21%
Maximum clock frequency	120 MHz

6 Conclusions

In this work an efficient hardware implementation of a class of CFAR processors for adaptive signal processing and target detection was presented. The high performance of the architecture was feasible since the employment of a parallel processing model and the arithmetic digital logic and parallel structures provided by FPGAs. The proposed architecture efficiently implements a class of related CFAR algorithms, the CA-CFAR and the MAX-CFAR, MIN-CFAR algorithms. The architecture nature exploits the parallel nature in CFAR signal processing and it can be extended to more complex CFAR algorithms such as the statistic ordered algorithms.

References

1. Merrill Ivan Skolnik, *Introduction to Radar Systems*, Editorial McGraw-Hill, 2000
2. Pearse A. Ffrench, James R. Zeidler, and Walter H. Ku, "*Enhanced Detectability of Small Objects in Correlated Clutter Using an Improved 2-D Adaptive Lattice Algorithm*", IEEE Transaction on Image Processing, Vol. 6, No. 3, March 1997
3. Tsakalides Paniagotis, Trinci Flippo, and Nikias Crysostomos L., "*Performance Assessment of CFAR Processors in Pearson-Distributed Clutter*", IEEE Transactions on Aerospace and Electronics Systems, vol. 36, No. 4, October 2000, pp. 1377-1386.
4. Lei Zhao, Wexian Liu, Xin Wu and Jeffrey S Fu, "*A Novel Approach for CFAR Processor Design*", 2001 IEEE Radar Conference, pp. 284-288.
5. Stefan Sjoholm, and Lennart Lindh, "VHDL for Designers", Prentice Hall, First Edition, 1997.

A Parallel FFT Architecture for FPGAs

Joseph Palmer and Brent Nelson

Department of Electrical and Computer Engineering,
Brigham Young University
Provo, UT 84602, USA
{palmer,nelson}@ee.byu.edu

1 Introduction

The inclusion of block RAMs and block multipliers in FPGA fabrics has made them more ammenable for implementing the FFT. This paper describes a parallel FFT design suitable for such FPGA implementations. It consists of multiple, parallel pipelines with a front end butterfly-like circuit to preprocess the incoming data and distribute it to the parallel pipelines. Implementation of the parallel FFT on Virtex II shows superlinear speedups for a range of FFT sizes.

1.1 The Parallel FFT

Figure 1 shows the standard data-flow graph for a 16-point, decimation in frequency FFT. The first stage is characterized by data communications between distant rows but at each new stage the dependencies move closer, and branches of independent data flow appear. After two stages the network has been divided into four independent partitions as denoted by the horizontal lines. This suggests the use of four independent processing elements to compute those partitions in parallel. This problem, known as the *parallel*

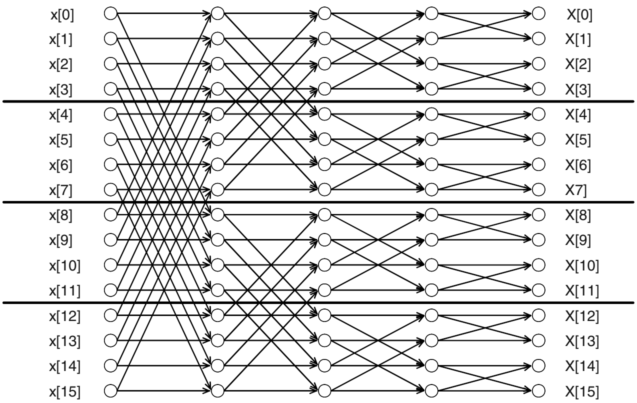


Fig. 1. 16-point FFT data-flow-graph

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 948–953, 2004.

FFT, has been extensively studied in the parallel computing community, and a number of algorithms for it have been proposed [1]. One method is to map groups of the rows of the data-flow-graph (Figure 1) onto multiple processors (four processors in the case of Figure 2. However, because of the interprocessor data dependencies found in the first two stages of the computation, special mechanisms must be employed to handle this. In the Binary-Exchange Algorithm[1], data must be exchanged between the processors during these first stages. An alternative algorithm is called the Transpose Algorithm[1] where the processors cooperate to compute the first two stages *without interprocessor communication* but a transposition of the data in memory is required after the second stage.

1.2 Mapping the Parallel FFT to FPGAs

We are generating a custom hardware implementation and are thus not limited to using n identical processing elements and standard interprocessor communication. Our approach is to create a custom front-end circuit to accomplish the first two stages of the computation and then feed those results to the parallel pipelines which do the partitioned calculations.

The Discrete Fourier Transform (DFT), for sample frame of size N, is defined as

$$X(\omega) = \sum_{n=0}^{N-1} x[n]W_N^{\omega n}, \tag{1}$$

where $W_N = e^{-j2\pi/N}$, known as the *twiddle-factor*.

We want to split our output, $X(\omega)$, into four blocks, and the DFT for size N can be decomposed into

$$X(4\omega) = \sum_{n=0}^{N-1} x[n]W_N^{(4\omega)n} \tag{2}$$

$$X(4\omega+1) = \sum_{n=0}^{N-1} x[n]W_N^{(4\omega+1)n} \tag{3}$$

$$X(4\omega+2) = \sum_{n=0}^{N-1} x[n]W_N^{(4\omega+2)n} \tag{4}$$

$$X(4\omega+3) = \sum_{n=0}^{N-1} x[n]W_N^{(4\omega+3)n}. \tag{5}$$

Since the FFT will need to input four samples for every four outputs, (2)-(5) need to be in terms of four separate input blocks. Beginning with (2),

$$\begin{aligned} X(4\omega) &= \sum_{n=0}^{N-1} x[n]W_N^{(4\omega)n} \\ &= \sum_{n=0}^{N/4-1} x[n]W_N^{4\omega} + \sum_{n=N/4}^{N/2-1} x[n]W_N^{4\omega} + \end{aligned}$$

$$\sum_{n=N/2}^{3N/4-1} x[n]W_N^{4\omega} + \sum_{n=3N/4}^{N-1} x[n]W_N^{4\omega}.$$

Now, using variable substitution in the summations, it follows that

$$\begin{aligned} X(4\omega) = &\sum_{n=0}^{N/4-1} x[n]W_N^{4\omega} \\ &+ \sum_{n=0}^{N/4-1} x[n+N/4]W_N^{4\omega n}W_N^{\omega N} \\ &+ \sum_{n=0}^{N/4-1} x[n+N/2]W_N^{4\omega n}W_N^{2\omega N} \\ &+ \sum_{n=0}^{N/4-1} x[n+3N/4]W_N^{4\omega n}W_N^{3\omega N}, \end{aligned}$$

and because $W_N^{Z\omega N} = 1$ and $W_N^{Z\omega n} = W_{N/Z}^{\omega n}$, where Z is some integer, the final solution becomes

$$\begin{aligned} X(4\omega) = &\sum_{n=0}^{N/4-1} (x[n] + x[n+N/4] + \\ &x[n+N/2] + x[n+3N/4])W_{N/4}^{\omega n}. \end{aligned}$$

The derivations for the other output blocks can be obtained in a similar fashion, resulting in

$$\begin{aligned} X(4\omega+1) = &\sum_{n=0}^{N/4-1} (x[n] - jx[n+N/4] + \\ &x[n+N/2] + jx[n+3N/4])W_{N/4}^{\omega n}W_N^{n}, \\ X(4\omega+2) = &\sum_{n=0}^{N/4-1} (x[n] - x[n+N/4] + \\ &x[n+N/2] - x[n+3N/4])W_{N/4}^{\omega n}W_N^{2n}, \\ X(4\omega+3) = &\sum_{n=0}^{N/4-1} (x[n] + jx[n+N/4] - \\ &x[n+N/2] - jx[n+3N/4])W_{N/4}^{\omega n}W_N^{3n}. \end{aligned}$$

Next, the following variables will be created,

$$\begin{aligned} a_0[n] = &(x[n] + x[n+N/4] + x[n+N/2] + \\ &x[n+3N/4]), \end{aligned} \tag{6}$$

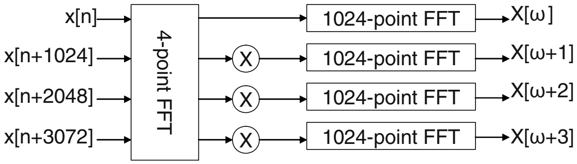


Fig. 2. 4096-point Quad-pipeline Radix-2^2 FFT

$$\begin{aligned} a_1[n] &= (x[n] - jx[n+N/4] + x[n+N/2] + \\ &\quad jx[n+3n/4])W_N^n, \quad (7) \\ a_2[n] &= (x[n] - x[n+N/4] + x[n+N/2] - \\ &\quad x[n+3n/4])W_N^{2n}, \quad (8) \\ a_3[n] &= (x[n] + jx[n+N/4] - x[n+N/2] - \\ &\quad jx[n+3n/4])W_N^{3n}, \quad (9) \end{aligned}$$

and substituting these into the derived block DFT equations produces

$$\begin{aligned} X(4\omega) &= \sum_{n=0}^{N/4-1} a_0[n]W_{N/4}^{\omega n} \\ X(4\omega+1) &= \sum_{n=0}^{N/4-1} a_1[n]W_{N/4}^{\omega n} \\ X(4\omega+2) &= \sum_{n=0}^{N/4-1} a_2[n]W_{N/4}^{\omega n} \\ X(4\omega+3) &= \sum_{n=0}^{N/4-1} a_3[n]W_{N/4}^{\omega n}. \end{aligned}$$

Note that these four results are each DFTs of length N/4, and share the same twiddle-factors of $W_{N/4}^{\omega n}$. Each can be computed by a separate FFT pipeline. The only additional hardware needed is to compute the set $\{a_1[n], a_2[n], a_3[n], a_4[n]\}$. The key observation is that this set of values can be computed using a conventional 4-point butterfly followed by modified twiddle factor mutliplications as shown in Figure 2 where the multiplication factors are the W's from equations (6) through (9).

This result is equivalent to that used for computing mixed radix FFT's [2]. The major contribution of this paper is to recognize that the method can be used to factor an FFT for parallel computation on an FPGA to achieve throughputs greater than 1. In addition, as will be show in the results below, the resulting parallel pipelines share significant logic and memory and achieve superlinear speedups (they use less than k times the hardware to achieve k-fold speedups). It should be obvious from the above discussion that breaking the FFT into four parallel pipelines is not the only choice — any k-way factorization (including non-power-of-two values) can be used followed by k parallel pipelines. Obviously such an approach can result in extremely high throughputs, limited only by the chip resources available. It also provides flexibility to trade off chip area for throughput. Both fixed point and block floating point versions of the FFT shown in Figure 2 have been implemented using a module generator written in JHDL, simulated, and tested on a Virtex II 6000-4. The module generator uses the Radix-2^2 algorithm [3] for the parallel FFT pipelines but any constant I/O FFT pipeline algorithm could be used.

Implementation results are shown in Table 1. All FFT's in the table use fixed-point computations and an 18-bit word size. They are constant-I/O, meaning that the samples to be processed can be input continuously without any breaks between frames. Throughput, in samples per second, can be computed for these FFT's by multiplying the clock rate by the number of pipelines. The table clearly shows that the quad-pipeline FFT is only 2-3 times as large as a single-pipeline FFT but has a 1/4th the transform time at essentially the same clock rate.

Table 1. Fixed-point FFT's on the XC2V6000-4

Frame	Pipeline	Slices	Block	Block	Speed	Latency	Throughput	Transform	Area × Transform
Size	Style		RAMs	MULTs	(MHz)	(cycles)	Msps	Time (μs)	Time Product
256	Single	2,233	6	9	163	547	163	1.57	3,506
	Quad	5,228	11	33	151	161	604	0.42	2,196
1K	Single	2,870	15	12	164	2,092	164	6.24	17,909
	Quad	7,656	27	45	151	554	604	1.70	13,015
4K	Single	3,838	33	15	155	8,245	155	26.4	101,323
	Quad	9,846	63	57	150	2,099	600	6.83	67,248

2 Comparison to Related Work and Conclusions

Multi-chip parallel FFT designs have been published [4] [5] but these focus on VLSI. At the time of the submission of this paper, two companies have released single-FPGA parallel FFT's — SiWorks and Pentek both advertise fixed-point FFT's with throughputs in the range of 400-500 Msps but provide few details on their internal architectures..

In contrast, this paper has proposed a Parallel FFT formulation, useful for creating parallel FFT's of essentially any size and throughput and suitable for custom hardware implementations using FPGA's. It exploits the characteristics of FPGA's (and custom hardware in general) as well as the structure of the FFT computation to significantly increase throughput.

Due to space constraints this paper has focused only on the parallel formulation of the FFT computation as completed in this work. This work also has included the development of block floating point FFT modules based on *convergent block floating point* [6] and provided an analysis of the cost and benefits of using block floating point. A parameterized module generator, written in JHDL, has been written which produces both fixed-point and block-floating-point FFT's of any size and parallelism desired (limited only by FPGA resources).

References

1. Ananth Gramam, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to Parallel Computing, Second Edition*, Pearson Education, Harlow, England, 2003.
2. Winthrop W. Smith and Joanne M. Smith, *Handbook of Real-Time Fast Fourier Transforms*, IEEE, New York, 1995.
3. S. He and M. Torkelson, "A New Approach to Pipeline FFT Processor," *The 10th International Parallel Processing Symposium(IPPS)*, pp. 766-770,1996.
4. Tom Chen, Glen Sunada, and Jian Jin. COBRA: A 100-MOPS single-chip programmable and expandable FFT. *IEEE Transactions on VLSI Systems*, 7(2), Jun 1999.
5. Yu Tai Ma. A VLSI-Oriented Parallel FFT Algorithm. *IEEE Transactions on Signal Processing*, 44(2), Feb 1996.
6. E. Bidet, D. Castelain, C. Joanblanq and P. Senn, "A Fast Single-Chip Implementation of 8192 Complex Point FFT", *IEEE Journal of Solid-State Circuits*, vol. 30, March 1995.

FPGA Custom DSP for ECG Signal Analysis and Compression

M.M. Peiró, F. Ballester, G. Paya, J. Belenguer, R. Colom, and R. Gadea

Departamento de Ingeniería Electrónica. Universidad Politécnica de Valencia,
46022 Valencia, Spain,
{mpeiro, fballest, guipava, rcolom, rgadea}@eln.upv.es

Abstract. This work describes a Virtex-II implementation of a custom DSP for QRS-Complex detection, ECG signal analysis and data compression for optimum transmission and storage. For QRS-Complex detection we introduce a custom architecture based on a modification of the Hamilton-Tompkins (HT) algorithm oriented to area saving. We also use biorthogonal wavelet transform for ECG signal compression and main ECG parameters estimation. In contrast with previously published works, our modified version of the HT algorithm offers best performance (a 99% of QRS-detection over normalized noisy ECGs). Moreover, a compression ratio of 20:1 is obtained when the wavelet-based engine is running. Results have been successfully verified by using a combination of MATLAB with SystemGen, Modelsim and a FPGA PCI-based Card (AlphaData ADM-XRC-II). The QRS-complex detector and compressor requires minimum area resources in term of LUT and registers, allowing a custom DSP as coprocessor in a SoC for biomedical applications.

1 Introduction

The medical sector has not taken advantage of the variety of light-weigh, low-power and low-cost microelectronics circuits that have been successfully applied in modern telecommunication systems. Main reason is the reduced quantities of classical ASIC required that are too small for mass production. To save this problem we propose as solution the use of a FPGA. The work describes a custom DSP for QRS complex detection or ECG analysis (ECG classification or ECG parameters estimation).

The ECG is one of the most common bioelectric signals with frequency range from 0.05 to 150 Hz an amplitude range between 5uV to 8mV. The recommended sampling rate is 500Hz whereas the sampled data should have 12-bit resolution that achieves about 4 – 5uV of resolution. Recently published papers [1] focuses on the fact that various morphologies of the ECG waveform are excited better at different frequency scales allowing both a compression of the ECG data and a detection of several important ECG parameters. These time-frequency analyses are accomplished by means of the wavelet transform that is a less computationally intensive method than the suggested in [2]. In addition, using wavelets the QRS complex can be efficiently detected [1]. The performance of the wavelet transform to detect the QRS complex is usually compared with detectors based on the Hamilton-Tompkins (HT) algorithm

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 954–958, 2004.

[3], results show that no one algorithm exhibit superior performance in all situation (with several noise addition). For the first time, this work compares both algorithms in its hardwired versions, estimating power consumption, area and frequency operations on a FPGA, test benches are based on the AHA (American Heart Association) tapes. Next section gives a brief review of the HT algorithm and describes our RTL modified version. Third section depicts the wavelet-based QRS detector chip. Finally, comparisons of both algorithms from the VLSI point of view are done.

2 A VLSI Implementation of HT QRS Detection Algorithm

The QRS detection algorithm developed by Hamilton and Tompkins recognizes QRS complexes based on the analysis of the slope, amplitude, and width. The algorithm proposes various cascaded high-pass and low-pass filters to attenuate noise (Fig. 1). Subsequent processes are differentiation, squaring, and time averaging of the signal, next step is differentiation and the squaring of signal samples to find and accentuate the QRS slopes. Finally, the window integrator and the adaptive amplitude estimate the peak signal level and the peak noise making the final determination as to whether or not a detected event was a QRS complex. Our version eliminates the moving average window and the squaring of the signal. Moreover, the adaptive amplitude threshold has been simplified too. Next paragraphs describe the algorithm.

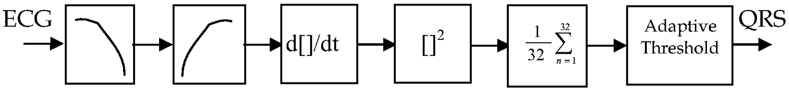


Fig. 1. The Hamilton-Tompkins basic building blocks.

2.1 Filters

The transfer function of the low-pass filter, high-pass and notch filters are described in eq. 1, 2 and 3 respectively. All the coefficients can be defined as power-of-two so the multipliers are reduced to shifts of the input data. As example, Fig. 2 shows the hardware realization of high-pass filter.

$$H_1(z) = -2 + 4 \cdot z^{-1} + 16z^{-2} + 16 \cdot z^{-3} + 4z^{-4} - 2 \cdot z^{-5} \tag{1}$$

$$H_2(z) = \frac{1023 - 1023 \cdot z^{-1}}{1024 - 1023 \cdot z^{-1}} \tag{2}$$

$$H_3(z) = 1 + z^{-5} \tag{3}$$

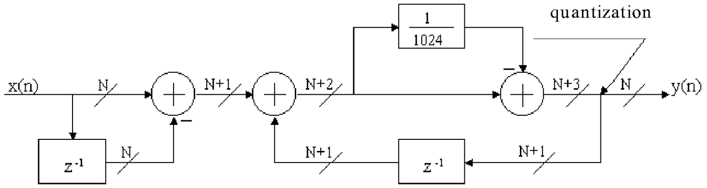


Fig. 2. Hardware organization of the high-pass filter.

2.2 Thresholding

Once the QRS has been filtered a threshold must be applied. The HT thresholding steps provided are:

1. Fix the first threshold = PEAK_LEVEL/2. Peak level = minimum signal value.
2. QRS-complex detected when signal amplitude is below the previous threshold.
3. Disable the detection up to 400ms to reduce interferences in the refractory period.
4. Set the Adaptive Threshold: New = 0.125·Peak_level+0.875· Old and go to step 2.

A multiplierless schema has been utilized: threshold is obtained rewriting 0.875 as 0.5 + 0.25 + 0.125 so only shifts and additions of the previous threshold are required.

2.3 Performance Measurement and Functional Verification

To test the modified version of the Hamilton Tompkins algorithm three functional verification levels have been provided by means of a MATLAB model verification, a RTL-code verification and the final verified chip by using a PCI-FPGA evaluation board. The output of the MATLAB code (floating-point precision) for several ECG signals with and without noise artefacts is evaluated for various parameters: TP or True Positive Detection, FP or False Positive, FN or False Negative, Se or Sensibility (eq. 4) and P+ or Positive Predictability (eq. 5).

$$Se = \frac{TP}{TP + FN} \tag{4}$$

$$P+ = \frac{TP}{TP + FP} \tag{5}$$

The results of the synthesizable QRS detector after simulation with ModelSim show a 99% of Se and a 100% of P+ of the QRS-complex detector compared to the MATLAB code. By using a PCI card the ECG signal is evaluated through the chip and the resulting QRS-complex detection is obtained from the PCI bus and shown on the PC screen.

3 A Wavelet-Based QRS-Complex Detector and Compressor

As the HT algorithm, the detection of QRS-complex is preceded by a filtering of the data in order to isolate the QRS frequency. The DWT proposed schema for a pair of filters is based on the polyphase decomposition. These filters are designed by using a folded structure [4]. The filter coefficients (Daubechies D_4) were specially selected for QRS detection. Once the ECG is wavelet decomposed, one of the last level outputs is selected and processed in a threshold detector or in a compressor engine (usually a run-length coder). The functionality of the threshold circuit is similar to the described in previous section.

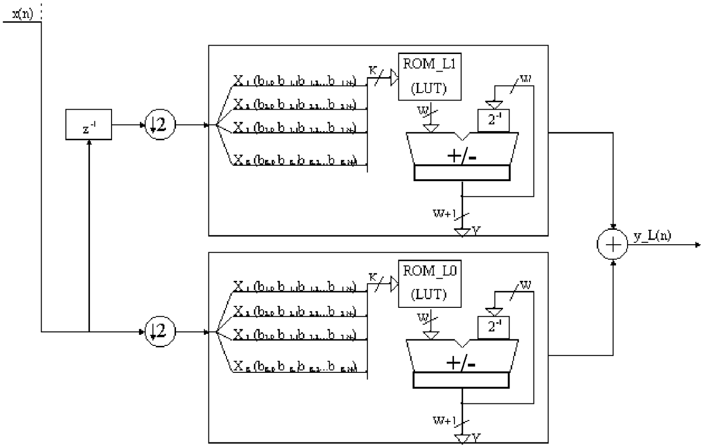


Fig. 3. The Low-Pass Distributed Arithmetic filter in its polyphase representation.

Fig. 3 shows the digit-serial distributed arithmetic (DA)-based basic cell. The filters are designed by using Look-up Tables (or ROM) and a scaling accumulator, additional information about the VLSI implementation of wavelets for FPGA by using DA can be read in our previously published work [5]. The results were extremely verified on the chip (see section 2.3). Table 1 offers post-synthesis results of the QRS detector chip for noisy ECG.

Table 1. Parameters obtained for noisy ECG.

Noise	#QRS	TP	FP	FN	Se(%)	P+(%)
Noiseless	*465*	*465*	*0*	*0*	*100*	*100*
Muscle	*465*	*459*	*1*	*6*	*99.78*	*98.71*
50Hz	*465*	*462*	*0*	*3*	*100*	*99.35*
Breathing	*465*	*462*	*0*	*3*	*100*	*99.35*
Body Movement	*465*	*439*	*141*	*26*	*76.85*	*94.41*
Mixed	*465*	*455*	*62*	*10*	*88.01*	*97.85*

4 Comparisons from the VLSI Point of View

This section offers results in terms of area, frequency operation and estimated power consumption of the described algorithms. The selected reconfigurable chip is a VIRTEX xc2v3000-6 from Xilinx with 28678 basic cells – one 4-input LUT (Look-Up Table) and one flip-flop build this cell –. Table 3 resumes the VLSI parameters obtained from the ISE Xilinx Tool. On behalf of ASIC implementation, in contrast with previously published woks [1], the Modified HT algorithm exhibits better performance, area and power consumption than the DWT for QRS detection (DWT + Threshold in Table 2). However, for ECG signal compression the DWT plus the run-length compressor can achieve an average compression ratio of 20:1. All algorithms described require less than a 1% of the total area resources of the chip.

Table 3. VLSI parameters for the Modified Tompkins and the DWT chips.

Algorithm	4-LUT	FlipFlop	Performance MHz	Power mW
Modified Tompkins	259	175	33.93	395
DWT + Threshold	645	284	30.19	411
DWT+ Compressor	514	251	32.15	402

5 Conclusions

The work describes modifications over two QRS-complex detector algorithms indented to its VLSI implementation. The functionality of the algorithms has been fully verified and more than 99% of predictability over noisy ECG has been obtained. Comparison in terms of FPGA parameters concludes than the modified HT is the best selection for area whereas the DWT-based algorithms can be use for both QRS-complex detection and compression with a penalty in the area and frequency operation. Nevertheless the area obtained in the three evaluated algorithms represents less than a 1% of the reconfigurable chip area. This fact implies reduced power consumption allowing the integration of portable biomedical systems in one chip.

References

1. S. Kadambe, R. Murray, G. Faye: Wavelet Transform-based QRS Complex Detector, IEEE Trans. On Biomedical Engineering, vol 46, nº7, July 1999.
2. S. Abeysekera: Detection and classification of ECG signals in the time frequency domain, Applied Signal Processing, vol. 1. Springer Verlag, 1994.
3. P. S. Hamilton, W.J. Tompkins: Quantitative investigation of QRS detection rules using the MIT/BIH arrhythmia database, IEEE Trans. On Bio. Eng, vol. 33, pp. 1157-1165, 1986.
4. M. Visvanathan: The Recursive Pyramid Algorithm for the Discrete Wavelet Transform. IEEE Trans. On Signal Processing, vol. 42, no. 3 (1994) 673-677.
5. G. Paya, M. M. Peiro, F. Ballester, F. Mora: Fully Parameterized Discrete Wavelet Packet Transform Architecture Oriented to FPGA, Lecture Notes on Computer Science, nº 2778, 2003, pp 533-542.

FPGA Implementation of Adaptive Multiuser Detector for DS-CDMA Systems

Quoc-Thai Ho and Daniel Massicotte

Université du Québec à Trois-Rivières, Department of Electrical and Computer Engineering
Laboratory of Signal and System Integration (http://lssi.uqtr.ca)
C.P. 500, Boul. des Forges, Trois-Rivières, Québec G9A 5H7, Canada
{Quoc_Thai_Ho, Daniel_Massicotte}@uqtr.ca

Abstract. A complete and transparent design flow targeted on FPGA components Virtex II and Virtex II Pro for multiuser detection in DS-CDMA wireless communication system is presented. The developed architectures are specific in order to exploit the modern technology of the programmable logic. The result of this work is modular, dedicated, specific and functional hardware architectures of an adaptive multiuser detector using a library of hard IP cores, optimized for these FPGA components, performing adaptive LMS filtering on normalized fixed-point complex signals. The developed architectures can be used as optimized cores to implement MUD function in a system-on-chip (SoC) for DS-CDMA wireless communication systems.

1 Introduction

Multiuser detection (MUD) is essential for an efficient high data rate 3G wireless network systems [1] deployment. It permits to cancel multi-access interference (MAI) due to the fact that several users have simultaneously accessed the same frequency band in such DS-CDMA (Direct Sequence Code Division Multi Access) systems. While algorithmic aspect of MUD is transparent, implementation aspect is non-trivial in order to satisfy real-time performances and hardware complexity [2],[3]. The bottleneck also consists of a transparent methodology of design [4]. In this paper we propose FPGA architecture and its implementation of an adaptive MMSE (Minimum Mean Square Error) algorithm. The rapid prototyping of more complex MUD based on iterative approach targeted on FPGAs is also proposed [6].

The paper is organized as follows. Section 2 presents briefly the MUD and the algorithm considered in this paper. Section 3 introduces the architecture of the MUD targeted on the Virtex II and Virtex II Pro devices. Section 4 presents the design flow while the results are presented in Section 5, and finally final remarks in Section 6.

2 Adaptive Multiuser Detectors for DS-CDMA

In a baseband DS-CDMA system, consider K users transmitting symbols from the alphabet $\Xi=\{-1,1\}$. Each user's symbol is spread by a pseudo-noise (PN) sequence of length N_c (signature or spreading code). Let T denotes the symbol period and T_c denotes the chip period where $N_c=T/T_c$ is an integer. The k^{th} user's continuous time

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 959–964, 2004.

spreading waveform is formulated by (1). The transmission channel $h_{k,l}^{(n)}(t)$ for user k is defined by (2) where L_k is the number of propagation paths, $h_{k,l}^{(n)}$ is the complex gain of the l^{th} path for user k at time n, $\tau_{k,l}$ is the propagation delay and $\delta(t)$ is the Dirac impulse function. The baseband received signal can then be written as (3) where N_b represents the number of the transmitted symbols, A_k is the transmitted amplitude of user k, and $\eta(t)$ is the additive Gaussian noise with variance σ_η^2. The discrete matrix-vector form of equation (3) (in baud space) is described by (4).

$$s_k^{(n)}(t) = \sum_{m=0}^{N_c-1} s_{k,m}^{(n)} \psi(t - mT_c) \tag{1}$$

$$h_k^{(n)}(t) = \sum_{l=1}^{L_k} h_{k,l}^{(n)} \delta(t - \tau_{k,l}) \tag{2}$$

$$\tilde{r}(t) = \sum_{n=0}^{N_b-1} \sum_{k=1}^{K} A_k b_k^{(n)} \sum_{l=1}^{L_k} h_{k,l}^{(n)} s_k^{(n)}(t - nT - \tau_l) + \eta(t) \tag{3}$$

$$\tilde{\mathbf{r}} = \mathbf{SHAb} + \boldsymbol{\eta} \tag{4}$$

Based on this multipath fading asynchronous DS-CDMA system, we will define an adaptive MUD receiver [5]. We consider the adaptive signature algorithm that is based on FIR (Finite Impulse Response) filters using the adaptive LMS (Least Mean Square) algorithm to detect all transmitted data b_k in a nonlinear manner. It was proposed as an alternative of MMSE algorithm. The proposed algorithm is so-called adaptive MMSE. The drawback of MMSE algorithm consists of algorithmic complexity due to matrix inversion. The adaptive MMSE is proved to be efficient as compared to the MUD algorithms in literature [1][5].

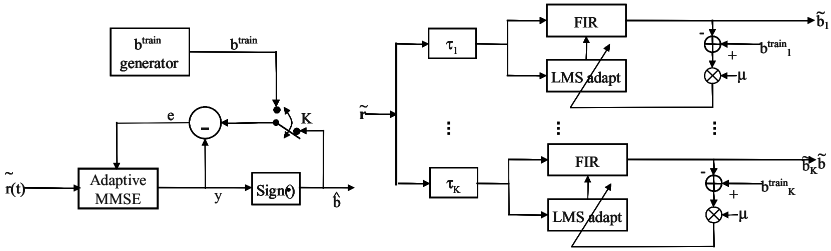


Fig. 1. Adaptive MMSE algorithm.

The general adaptive LMS algorithm is described by (5) and (6) for each user k. The coefficient vector $\mathbf{w}_k^{(n)}$, with $\dim(\mathbf{w}_k^{(n)}) = N_T$, contains the adaptive spreading code of one user and $\mathbf{x}_k^{(n)}$ represents the received signal $\tilde{\mathbf{r}}$ shifted by the delays τ_k. The output signal of adaptive LMS filters is produced to match the real desired BPSK signal based on a training sequence. Considering the complex number multiplication, the equation (5) is computed using (7). Since the desired signal is real, we can update the coefficients of (6) using (8).

$$y_k^{(n)} = \mathbf{w}_k^{T(n)} \mathbf{x}_k^{(n)} \tag{5}$$

$$\mathbf{w}_k^{(n+1)} = \mathbf{w}_k^{(n)} + \mu e_k^{(n)} \mathbf{x}_k^{(n)} \text{ with } e_k^{(n)} = b_k^{train\ (n)} - y_k^{(n)} \tag{6}$$

$$\begin{aligned} R_{re} &= \sum_{i=0}^{N_T-1} \left(\mathrm{Re}(x_{k,i}^{(n)}) \mathrm{Re}(w_{k,i}^{(n)}) - \mathrm{Im}(x_{k,i}^{(n)}) \mathrm{Im}(w_{k,i}^{(n)}) \right) \\ R_{im} &= \sum_{i=0}^{N_T-1} \left(\mathrm{Re}(x_{k,i}^{(n)}) \mathrm{Im}(w_{k,i}^{(n)}) + \mathrm{Im}(x_{k,i}^{(n)}) \mathrm{Re}(w_{k,i}^{(n)}) \right) \end{aligned} \tag{7}$$

$$\begin{aligned} \mathrm{Re}\left(\mathbf{w}_k^{(n+1)}\right) &= \mathrm{Re}\left(\mathbf{w}_k^{(n)}\right) + \mu \left(b_k^{train(n)} - R_{re}\right) \mathrm{Re}\left(\mathbf{x}_k^{(n)}\right) \\ \mathrm{Im}\left(\mathbf{w}_k^{(n+1)}\right) &= \mathrm{Im}\left(\mathbf{w}_k^{(n)}\right) + \mu \left(b_k^{train(n)} - R_{im}\right) \mathrm{Im}\left(\mathbf{x}_k^{(n)}\right) \end{aligned} \tag{8}$$

Re(x) and Im(x) define the real and imaginary parts of x, and R_{re} and R_{im} represent the accumulation registers for real and imaginary parts respectively. From these equations of the adaptive MMSE, we will describe the proposed FPGA-targeted architecture.

3 FPGA-Based Architecture of Adaptive MMSE

The modular architecture is based on a modular array of processing elements (PE). These PEs are hard IP cores performing adaptive LMS filtering on normalized-fixed complex signals according to (7) and (8). The core of these adaptive filters is the pipelined multiplier-accumulator using the dedicated multipliers available on the silicon die of Virtex II, Virtex II Pro devices. The timing scheduling is based on time-multiplexing (T_{MUX}=1,2,3,4). The data and address paths are independent. The principle of direct association is used to solve the hazard problem of memory access. The utilization of distributed RAM implemented by LUTs permits to implement multiport memory buffers of MUD accessing simultaneously to external memories. BRAM (Block RAM) are used as memories of filters and they are internal memories of MUD. Since the filters are based on complex multiplier-accumulators (CMAC), it is, hence, critical to consider an efficient implementation of a complex multiplier on these FPGA components. We used an advanced scheduling based on time multiplexing by modifying the conventional methods, i.e. ASAP (As Soon As Possible) and ALAP (As Late As Possible) to use only two real multipliers. This method exploits the symmetric property of BRAMs and the dedicated multipliers of these components. The execution time of an adder is 1-cycle (C-step) and that of a multiplier is 2-cycle. The total execution time for the complex LMS-FIR of N_T complex taps is $(2N_T+9)T_{clk}$ where T_{clk} is the clock cycle period.

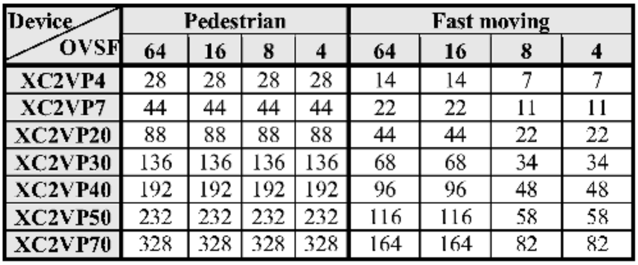
4 SoC Design Flow

Once the algorithm in fixed point is functional, the execution of system is refined into HW and SW tasks. This paper focuses on the HW design flow of the MUD based on a library of the optimized hard IP cores; for instance, complex taps LMS filters used as PE for the adaptive MUD. Different architectures were developed in order to optimize the HW resources and timing performances as a function of several different algorithmic specifications of the MUD such as the OVSF (*Orthogonal Variable Spreading Factor*) used in WCDMA as channelisation codes, the number of users K, the adaptation period, etc. The principal difference between these architectures is the global control in order to exploit the possible strategy of pipelining to minimize HW resources and satisfy performance requirements.

It is necessary to estimate the timing performance and HW resources required by architectures from the architectural specifications stated above. Thus, a program based on nonlinear integer programming model was developed. This tool is used to maximize the time multiplexing (number of users in one PE) and timing performances (number of clock cycles) of system respecting algorithmic constraints and HW resource limitations (number of multipliers and block RAM). It is also necessary to minimize the clock rate for consummation performance. This program helps to choose the type of architecture more corresponding in terms of the strategy of pipeline for the algorithmic specification of MUD. Also, it can be used to estimate the necessary HW resources and timing performances.

The specification and characterization step makes it possible to specify and characterize the PE for a given algorithm. For the algorithm of the MUD to be implemented in this project, the PE are dedicated as adaptive LMS and FIR filters on the complex signals. The advanced scheduling method modified from the conventional techniques (ASAP and ALAP) makes it possible to use only two real multipliers to build the CMAC. The CMAC is 2-level pipeline using the dedicated multipliers available on these FPGAs. The left alignment method is used for coding numbers in normalized fixed format.

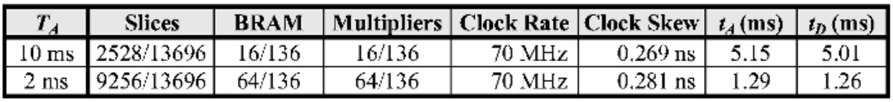
Table 1. Maximum number of users (*K*) in Virtex II Pro devices for different OVSF's (64, 16, 8 and 4) and mobile speeds (pedestrian and fast moving)



Device / OVSF	Pedestrian				Fast moving			
	64	16	8	4	64	16	8	4
XC2VP4	28	28	28	28	14	14	7	7
XC2VP7	44	44	44	44	22	22	11	11
XC2VP20	88	88	88	88	44	44	22	22
XC2VP30	136	136	136	136	68	68	34	34
XC2VP40	192	192	192	192	96	96	48	48
XC2VP50	232	232	232	232	116	116	58	58
XC2VP70	328	328	328	328	164	164	82	82

The remainder of HW design flow relies on conventional design methodology targeted on FPGA: the netlist in EDIF format is used as unified format for Xilinx physical tool, i.e. placement and routing; the timing constraints are applied in a hierarchic manner considering that the verification is critical in the design flow; dynamic verification by simulations is used during all along design flow; results of fixed-point simulations in high-level language (Matlab®) are used as static functional reference for the HW verification of the architecture; and finally the synthesized data are used for the verification of adaptive MMSE FPGA implementation.

Table 2. Post Layout Synthesis Results for Xilinx Virtex II PRO XC2VP30 device with OVSF=16, K=16 and 100 iterations for pedestrian and fast moving conditions



T_A	Slices	BRAM	Multipliers	Clock Rate	Clock Skew	t_A (ms)	t_D (ms)
10 ms	2528/13696	16/136	16/136	70 MHz	0.269 ns	5.15	5.01
2 ms	9256/13696	64/136	64/136	70 MHz	0.281 ns	1.29	1.26

5 Performance Evaluation

The results of this work are functional HW architectures targeted on the Virtex II and Virtex II Pro components satisfying the different algorithmic specifications. FPGA implemented validations are done in comparison with Matlab simulation based on CDMA platform.

Table 1 summarizes the maximum number of users (K) that can be treated on different devices of the Virtex II Pro family in different data throughput based on the 3G standard and fixed by the OVSF parameter such as 64, 16, 8 and 4 corresponding respectively to 12.2kbps (voice rate), 64kbps, 144kbps and 384kbps data rates. We considered two mobile speeds: pedestrian (T_A=10ms) and fast moving (T_A=2ms), where T_A represents the adaptation sequence period considered to adapt all MMSE parameters, $\mathbf{w}_k^{(n)}$, in N_A=100 iterations to adapt each users. Assumption of 100 MHz clock frequency, we can used $\mathrm{T}_{\mathrm{MUX}}$=4 in pedestrian case and $\mathrm{T}_{\mathrm{MUX}}$=2 for OVSF of 64 and 16 and $\mathrm{T}_{\mathrm{MUX}}$=1 for OVSF of 8 and 4 in case of fast moving.
The performance in term of adaptation time (t_A) and detection time (t_D) is defined by t_A=(3N_T+9)($\mathrm{T}_{\mathrm{MUX}}N_A$)(256/OVSF)$\mathrm{T}_{\mathrm{clk}}$ and t_D=(2N_T+5)$\mathrm{T}_{\mathrm{MUX}}$(38400/OVSF)$\mathrm{T}_{\mathrm{clk}}$. With pipeline strategy of architecture, the time processing of this system is max($t_{A,}t_D$) and need to be inferior to one time frame (10 ms) to adapt all parameters and to detect all data. $\mathrm{T}_{\mathrm{MUX}}$ is maximized to respect the time constraint of $t_A \leq T_A$ and $t_D < T_D$ and depend of the clock frequency.

From Table 1, we observed a constant maximum number of users for all throughputs due to the limitation of resources (BRAM to store $\mathbf{w}_k^{(n)}$ and multipliers). Indeed, the OVSF used in WCDMA imposed to store and adapt 256/OVSF vectors $\mathbf{w}_k^{(n)}$ for each user. The post layout synthesis results are included in Table 2 using Leonardo and Foundation tools where $\mathrm{T}_{\mathrm{MUX}}$=4 for pedestrian and $\mathrm{T}_{\mathrm{MUX}}$=1 for fast moving..

6 Final Remarks

The HW architectures of an adaptive multiuser detector based on adaptive MMSE for DS-CDMA systems were developed. These dedicated architectures can be used later as optimized hard cores performing MUD functions. The current HW architectures are purely glue logic. The future work consists in exploiting SW processing in the MUD as a whole respecting the algorithmic specifications of the 3G wireless communications. As for the design methodology level, it will be interesting to use a higher-level language for architectures' HW design flow. The current flow is based on VHDL that is time-consuming in our modeling task for different architectures.

Acknowledgements. The authors wish to thank the NSERC of Canada for their financial support and they are grateful to the Axiocom inc. for appreciate technical and financial contributions.

References

1. S. Verdù, *MultiUser Detection*, Cambridge University Presses, 1998.
2. G. Xu, S. Rajagobal, J.R. Cavallaro, and B. Aazhang, “VLSI Implementation of the Multistage Detector for Next Generation Wideband CDMA Receivers”, Journal of VLSI Signal Processing 30, Kluwer Academic Publishers, march 2002, pp. 21-33.
3. S. Rajagobal, S. Bhashyam, J.R. Cavallaro, and B. Aazhang, “Real-Time Algorithms and Architectures for Multiuser Channel Estimation and Detection in Wireless Base-Station Receivers”, IEEE Trans. Wireless Communications, July 2002, Vol 1, No. 3, pp. 468-479.
4. Y. Guo, G. Xu, D. McCain, and J.R. Cavallaro, “Rapid Scheduling of Efficient VLSI Architectures for Next-Generation HSDPA Wireless System Using Precision C Synthetizer”, IEEE Workshop Rapid Systems Prototyping, San Diego, 2003, pp. 179-185.
5. A.O. Dahmane, D. Massicotte, "DS-CDMA Receivers in Rayleigh Fading Multipath Channels: Direct vs. Indirect Methods", IASTED Int. Conf. on Communications, Internet, and Information Technology (CIIT 2002), St. Thomas, 18-21 Nov., 2002.
6. W. Schlecker, A. Engelhart, W.G. Teich, H.-J. Pfleiderer, “FPGA Hardware Implementation of an Iterative Multiuser Detection Scheme”, 10th Aachen Symposium on Signal Theory (ASST), Aachen, Germany, September 2001 pp. 293-298.

Simulation Platform for Architectural Verification and Performance Analysis in Core-Based SoC Design

Unai Bidarte, Armando Astarloa, José Luis Martín, and Jon Andreu

University of the Basque Country, E.T.S. de Ingeniería de Bilbao,
Urquijo s/n, E-48013 Bilbao - SPAIN
{jtpbipeu, jtpascua, jtpmagoj, jtpanlaj}@bi.ehu.es

Abstract. In complex SoC designs verification consumes more than half of the overall design effort. Design reuse is a critical element in closing the SoC design gap, but it is not enough. A generic core-based architecture for circuits that require high volume data transfer control was designed. After some experience in reusing the architecture, a key improvement has recently been undertaken: a reusable simulation platform. The complexity of architectural verification and performance analysis has been greatly alleviated by means of a monitor module that processes all the events on the SoC and an Architecture Specific Graphical User Interface (ASGUI) that shows all the data transfers while simulation is running. The generation of bitrate and latency statistics is fully automated.

1 Introduction

Several control applications require very high speed data exchange between data source and sink elements [1] [2] [3]: industrial machinery like filling or milling machines, polyphonic audio, three dimensional images, video servers, PC equipment like plotters and printers, etc. After dealing for quite a long time with such applications, it was considered that much work could be reused, and a generic and reusable core-based architecture for circuits that require high bandwidth data transfers was designed in order to reduce the SoC design cycle time as much as possible [4] [5]. An schematic representation of the generic data transfer control architecture is shown in figure 1 inside the rectangle named SoC [6]. It is composed of three buses that are compatible with the Wishbone SoC interconnection Architecture for Portable IP Cores specification [7]:

- Data Transfer Bus (DTB). It is a shared bus interconnection that performs all high speed data transfers. High bandwidth data transfer control can be a very time consuming task. In order to liberate the main control unit a data transfer control specific unit has been used.
- Main Processor Bus (MPB). It is used to perform high level tasks, which are controlled by a high level controller or main processor, usually a microcontroller. It uses its specific bus to read and write on program memory, input and output blocks and any other devices, as well as to communicate with the DTB.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 965–969, 2004.

- Communication Bus (CB). It is used to exchange information with the external system. It can not directly access to the DTB and it exchanges information with the communication processor, which is a module on the DTB.

Architecture reusability was improved by taking into account the usual design-for-reuse guidelines [8]: configurable cores and verified in detail, flexible and scalable architecture, technology and CAD tools independence, good documentation, Register Transfer Level (RTL) synchronous descriptions, etc.

2 Improving Architecture Reuse by Means of a Simulation Environment

Although design complexity was alleviated by reusing the flexible and generic system, architectural verification and analysis were still very time consuming tasks because each time the architecture was reused for a new application, an application specific testbench was developed [9]. At this point, a key improvement has recently been undertaken: a simulation platform valid for any application designed by reusing the generic architecture [10]. These are the goals of the simulation platform:

- Verification. Although each core is a preverified block, it must be ensured that the whole SoC implements intended functionality. Without functional verification, one must trust that the transformation of a specification document into RTL code was performed correctly, without misinterpretation of the specification's intent [11][12].
- Performance analysis. We want to analyze the influence of architectural parameters like the number of simultaneous active channels, the type of data units, and the use of the SDRAM on system features like bitrate and latency.

The complexity of stimuli generation and result analysis has been greatly alleviated by means of using text files. The circuit description and the data exchanged are separated, so simulation data and circuit descriptions are clearly distinguished. In complex SoCs the amount of interconnection signals is so extensive that traditional timing diagrams become nearly unmanageable. An Architecture Specific Graphical User Interface (ASGUI) that shows all the transfers has been developed to simplify the verification task. Tool Command Language/Tool Kit (TCL/Tk) has been used to create a custom window and simulation control features because the command line prompt to the simulator employed (Modelsim) is a Tcl/Tk shell and because it has powerful and flexible capabilities. Scripting languages like Tcl provide an ideal mechanism for integration tasks in component-based designs [13].

3 Simulation Platform

Figure 1 shows the block diagram of the whole simulation platform. All the circuits that compose the SoC under study are represented with a rectangle.

The surrounding circuits that are modelled (a bus functional model) only for simulation porpouses are represented with a circle and all of them are provided with a virtual interface to access one or more text files.

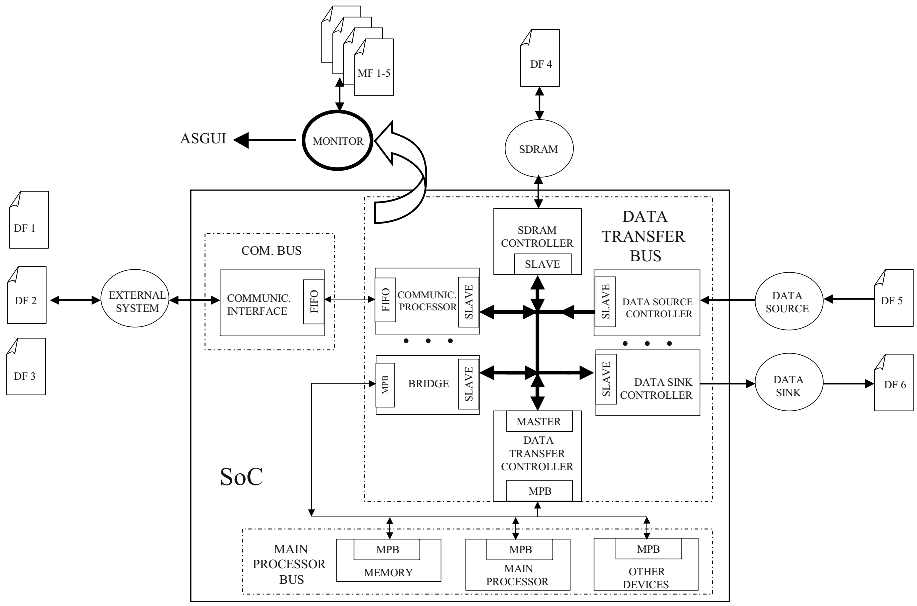


Fig. 1. Block diagram of the simulation platform.

The MONITOR registers all the events in the system and performs two kinds of tasks: it acknowledges the ASGUI on any event that must be shown and it stores performance information in text files. A spreadsheet automatically generates bitrate and latency graphical representations from these monitor files. Table 1 shows the name and description of the most important data and monitor files.

Figure 2 is a screenshot of the Architecture Specific Graphical User Interface (ASGUI). Three main zones can be distinguished:

- Pull-down menus, at the top, to access the VHDL and text files.
- Graphical user interface, at the middle, to create the topology of the simulation platform and to visualize the data transfers performed in the buses. Each simulation clock cycle there is a call to a process that verifies wether a new data transfer has been performed. The simulation is halted whenever a data transfer occurs. Then an arrow shows the origin and destination of the data transfer and the content of the data and address buses is also indicated.
- Simulation control buttons, at the bottom, make possible the interaction with the simulation software.

Table 1. Data and monitor text files.

Name	Description
Data files	
ext_tx_com.txt (DF1)	Commands to be transmitted by the external system
ext_tx_data.txt (DF2)	Data to be transmitted by the external system
ext_rx.txt (DF3)	Commands and data received by the external system
sdram_banki.txt (DF4)	Data in bank i of the SDRAM
dso_data.txt (DF5)	Data to be read by the data source unit
dsi_data.txt (DF6)	Data written by the data sink unit
Monitor files	
transfer_dtb.txt (MF1)	Transfers in the Data Transfer Bus (DTB)
error_dtb.txt (MF2)	Errors in the Data Transfer Bus (DTB)
bit_rate.txt (MF3)	Bitrate in each channel
latency.txt (MF4)	Latency in each channel
sdram_access.txt (MF5)	Read and write accesses to SDRAM

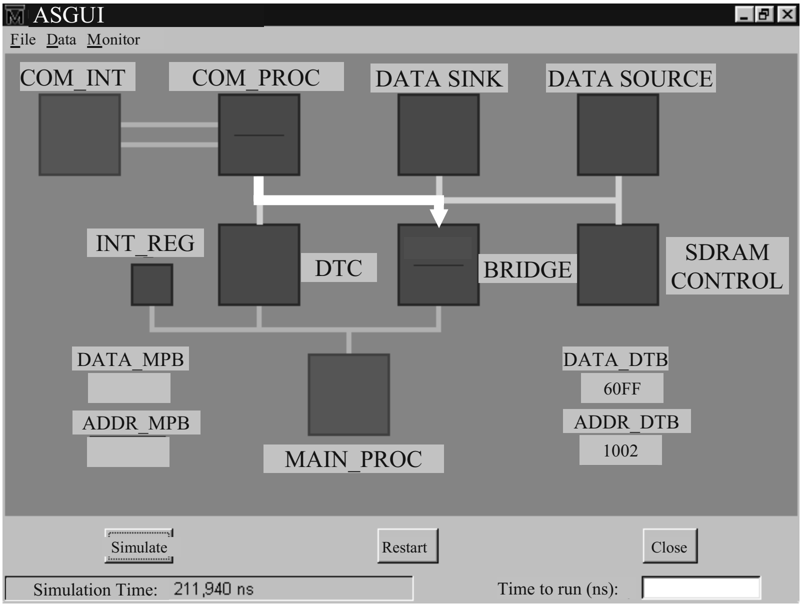


Fig. 2. Architecture Specific Graphical User Interface (ASGUI).

4 Conclusion

Several strategies and concepts are being employed to build chips in the multi-million-gate range in a reasonable amount of time. When reusing predesigned core-based systems, architectural verification and analysis become very time consuming tasks. The simulation platform presented in this paper improves the reusability of a generic architecture for high bandwidth data transfer control. All the events on the system are registered by a monitor block and an Architecture Specific Graphical User Interface (ASGUI) shows all the data transfers while simulation is running. Simulation inputs and outputs are stored in text files, which makes possible an easy procedure to manage simulation stimuli and performance analysis results. A spreadsheet generates automatically bitrate and latency statistics and multiple graphical representations.

References

1. K. Lahiri, A. Raghunathan, and S. Dey. Efficient Exploration of the SoC Communication Architecture Design Space. In *Computer Aided Design*, pages 424 – 430, 2000.
2. R. Clauberg, P. Buchmann, A. Herkersdorf, and D. J. Webb. Design Methodology for a Large Communication Chip. *IEEE Desing and Test of Computers*, pages 86–94, July-September 2000.
3. R. Niemann and P. Marwedel. *Hardware-Software Codesign for Data-Flow Dominated Embedded Systems.* Kluwer Academic Publishers, Boston, 1998.
4. F. Vermeulen. *Reuse of System-Level Design Components in Data-Dominated Digital Systems.* PhD thesis, Katholieke Universiteit Leuven, België, 2002.
5. T. Thomas. Technology for IP Reuse and Portability. *IEEE Design and Test of Computers*, 16(4):7–13, 1999.
6. U. Bidarte, A. Astarloa, A. Zuloaga, J. Jiménez, and I. Mtz. de Alegría. Core-Based Reusable Architecture for Slave Circuits with Extensive Data Exchange Requeriments. *Lecture Notes in Computer Science*, 2778:497–506, 2003.
7. Opencores and Silicore. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision: B.3. http://www.opencores.org/wishbone/, 2002.
8. M. Keating and P. Bricaud. *Reuse Methodology Manual.* Kluwer Academic Publishers, June 2002.
9. K. Lahiri, A. Raghunathan, and S. Dey. System-Level Performance Analysis for Designing On-Chip Communication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.
10. G. D. Peterson and J. C. Willis. High-Performance Hardware Description Language Simulation: Modeling Issues and Recommended Practices. *Transactions of The Society for Computer Simulation International*, 16(1):6–15, 1999.
11. A. Higashi, K. Tamaki, and T. Sasaki. Verification Methodology for a Complex System-on-a-Chip. *Fujitsu Scientific and Technical Journal*, 36(1):24–30, 2000.
12. J. Bergeron. *Writing Testbenches. Functional Verification of HDL Models.* Kluwer Academic Publishers, USA, 2001.
13. J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

A Low Power FPAA for Wide Band Applications

Erik Schüler and Luigi Carro

Universidade Federal do Rio Grande do Sul
Departamento de Engenharia Elétrica
Av. Osvaldo Aranha, 103
Porto Alegre, RS, Brasil
{eschuler, carro}@eletro.ufrgs.br

Abstract. The necessity for programmable analog devices appears when it is necessary to ease the design process in small time and with minimum costs, making the research in FPAAs (Field Programmable Analog Array) more intense. In this work we proposed a new FPAA topology aiming at wide frequency range, while keeping a high degree of reconfigurability. The impact in frequency, programmability and resolution are evaluated, showing that the proposed technique gives a good performance in these aspects.

1 Introduction

The versatility of the FPAA (*Field Programmable Analog Array*) never reached the same level of versatility of the FPGA (*Field Programmable Analog Array*) because of factors such as the need of many reconfiguration and interconnections switches, the huge area occupied for each passive element of the circuit and the granularity involved to create different circuits. Not only versatility suffers with these design aspects. Frequency and linearity limitations due to the same switches that allow programmability are currently a point of interest in FPAA research.

In this paper we present a solution able to cope with the frequency problem. By translating the signal, instead of changing the circuit, we can achieve high performance in a wide range of frequencies, much larger than any commercial FPAA can presently reach. Thanks to the processing by a simple analog function, we do not incur in extra power dissipation.

The paper is organized as following: a brief review about some of the main FPAA architectures is reviewed in section 2. In section 3 the proposed interface is presented. Section 4 shows some practical results that prove the functionality and feasibility of the system.

2 Related Works

Some of the main representatives of the different techniques used to implement FPAAs are switched capacitor, operational amplifier, current conveyor and switched current. The switched capacitor technique favored the development of active filters

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 970–974, 2004.

for precision applications, especially in low frequencies. However, this technique depends on linear capacitors, which occupy a large area in integrated systems, mainly in low frequencies. Moreover, this kind of circuit realization has its operation frequency limited to the Nyquist rate and, since most applications require a huge amount of programmability and interconnection switches, these also limit the achievable bandwidth. On the other hand, structures based on switched capacitors have a high degree of programmability, since some parameters can be changed only through the variation of the switch frequency of the capacitors. Switched capacitor based FPAAs developed are described in [1] and [2], which originated the FPAA currently fabricated by Anadigm, the AN10E40.

FPAAs based on operational amplifiers (opamps) have a band limitation imposed by the characteristics of the opamp itself. Also, an opamp alone does not realize any function but a comparator, and hence the inclusion of passives elements such as capacitors, resistors and diodes becomes necessary, in order to make more complexes functions. Thus, the same problem associated with area of passive components in lower frequencies is now present, and the insertion of switches to program all these components has also impact on the maximum achievable bandwidth and linearity. Commercial devices produced by Lattice Semiconductor Corp. (ispPAC10/20/30/80/81 and ispPAC-POWR604/1208) and by Zetex Ltd. (TRAC) are example of opamp based FPAAs.

The use of current conveyors in the development of FPAAs presents as a great advantage its large bandwidth, which can easily reach hundreds of MHz. Again, its use is limited due to the passive components needed to create some analog function and the switches associated to these components. Current conveyor based FPAAs are described in [3] and [4].

A technique analogous to the switched capacitor is the switched current. This technique allows circuits with lower power consumption. Two problems exist in this topology: there is a lower resolution if one compares it with switched capacitor circuits, and it is necessary to create several copies of the current signal to feed multiple blocks requiring the same signal. Also, bandwidth limitation is restricted, again, to the Nyquist rate. Example of switched current based FPAA is cited in [5].

Others topologies have been used in order to get betters results in one or another characteristic. These are, between others, current mirrors, subthreshold operating CMOS, multiplexer and demultiplexer and others. A generic topology, allowing high performance in a wide frequency range has not yet been developed.

3 Proposed Interface

Trying to cope with the problem of limited range of frequency in which the FPAA can operate, without degrading its level of programmability, we propose the use of frequency shift interface (figure 1), which also have some programmability, both in the mixer (through LO variation) and in the band pass filter (through its parameters variation, such as cut frequency, gain and quality factor).

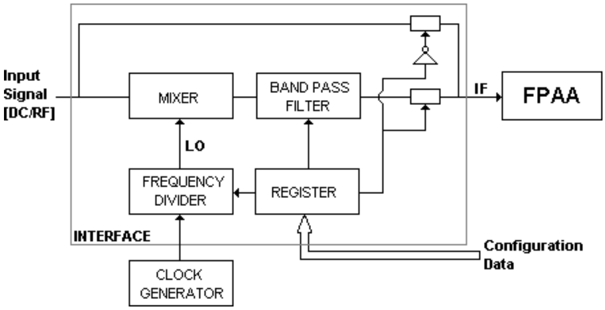


Fig. 1. Proposed structure to signal allocation and selection, with exclusively analog processing of the signal.

With this interface, signals to be processed are now centered in a fixed frequency, allowing the development of the rest of the FPAA optimized to work in this frequency. Even if one thinks in a FPAA that has a certain bandwidth of operation, this interface can be used to allocate any signal out of this bandwidth to a frequency where the device is able to operate. The practical results showed in section 4 use this principle.

4 Practical Results

In order to demonstrate the feasibility of implementation of the proposed interface and its functionality, practical experiments have been developed, where signals with restricted band were translated to a fixed frequency with a passive mixer. A commercial FPAA has been used as the programmable vehicle, in order to show the generality of the method. The used structure is presented in figure 2. The RF input senses de signals in different frequencies. LO is the local oscillator signal and the resulting signals in IF are the difference and the sum of LO and RF. The desired signal is then selected by a band pass filter realized with a Lattice FPAA, the ispPAC10, which consists of two second order sections with quality factor of 7.59. The subsequent block is another Lattice FPAA ispPAC20 that implement an adder.

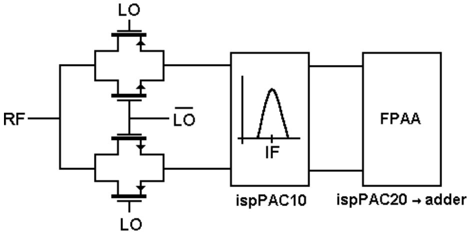


Fig. 2. Structure of mixer and band pass filter followed by a FPAA, used in practical results.

The mixer was implemented with transmission gates to give a better noise margin than one would get with a single transistor. The signals were generated by a

HP33120A and a Tektronix CFG253 function generators and acquired by a HP54645D oscilloscope with a rate of 25 MSa/s.

In the experiment, it is desired to sum two sinusoidal signals of high frequency (5MHz and 5,01MHz), using ispPAC20, whose maximum operating frequency if 550KHz. It is necessary, then, to achieve a down conversion of these signals to a frequency where ispPAC20 is able to work and make their sum. An IF of 60KHz was chosen, resulting in a LO frequency of 5.06MHz. The signal in 5MHz is then translated to this frequency, while the 5,01MHz signal is directly introduced into ispPAC20, already in a frequency of 70KHz. Figure 3 shows the acquired results.

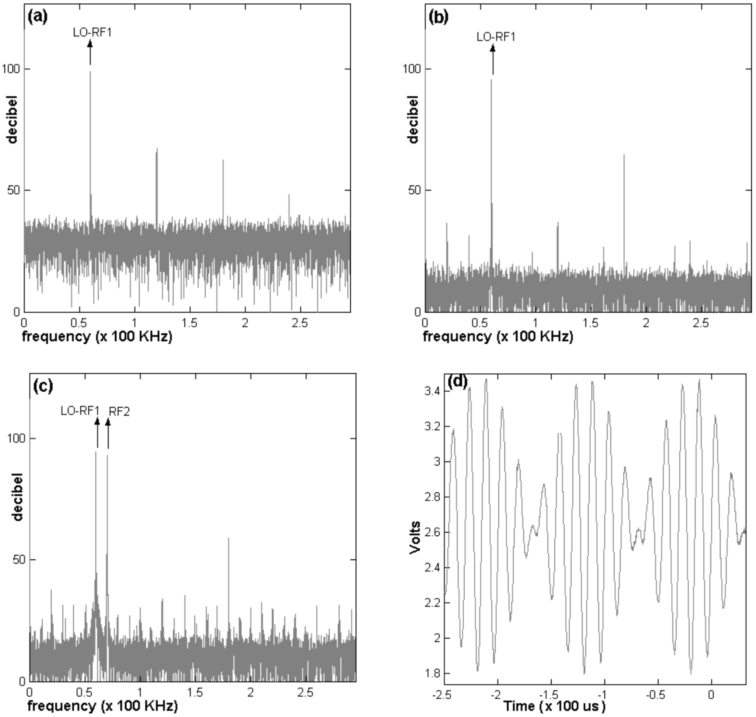


Fig. 3. (a) FFT of the 5MHz signal mixed to 60KHKz; (b) FFT of the band pass filtered signal; (c) FFT of the sum of the 60KHz and 70KHz signals; (d) output signal in time domain.

A mixer based on switches generates, beyond |LO±RF|, higher order harmonics located in |3LO±RF|, |5LO±RF|, and so on [6]. This way, the harmonics that appear in figures 3(a) to 3(c), are not introduced by the proposed interface (mixer + band pass filter), but rather by three other different sources: both the input signals in 5MHz and 70KHz externally generated, the band pass filter implemented with ispPAC10 and the adder, implemented with ispPAC20. This means that the original system (without the interface) produces harmonics that degrade the signal. Since the harmonics introduced by the mixer are in high frequencies, the proposed interface is not contributing with the distortion in the operating frequency band, and all distortion is caused by the reconfigurable commercial device and the input signal. This shows that the actual effect of the mixer at the input is to extend the performance of the target FPAAs.

With the system presented, one can easily deal with signals in frequencies very superiors than those achieved today by the commercial FPAA, for example the AnadigmVortex family (2MHz), Lattice (1.57MHz for the ispPAC30) and Zetex (3MHz for most of the blocks).

5 Conclusions

The proposal presented in this work comes as a solution for the limited range of frequency in which FPAA can operate.

Practical experiment shows results obtained for signals in frequencies much higher than the commercial FPAA can operate, confirming the system efficiency, allowing its use in a large range of applications, from low frequency to high frequency.

As future works, analysis will be done regarding the power dissipation of the system, comparatively with others FPAAs, as well as limits of operation must be determined, taking account problems related to frequency translation using mixers based on switches and the distortions introduced in this process due to mixer's non linearity.

References

1. Lee, E.K.F.; Hui, W.L. *A novel switched-capacitor based field-programmable analog array architecture*. Kluwer Analog Integrated Circuits and Signal Processing - Special Issue on Field Programmable Analog Arrays, v.17, n.1-2, p.35-50, Set. 1998.
2. Bratt, A.; Macbeth, I. *DPAD2 - A field programmable analog array*. Kluwer Analog Integrated Circuits and Signal Processing - Special Issue on Field Programmable Analog Arrays, v.17, v.1-2, p.67-89, Set. 1998.
3. Premont, C. *et al*. *A current conveyor based field programmable analog array*. Kluwer Analog Integrated Circuits and Signal Processing - Special Issue on Field Programmable Analog Arrays, v.17, n.1-2, p.105-124, Set. 1998.
4. Gaudet, V.; Gulak, G. *10 MHz Field Programmable Analog Array Prototype Based on CMOS Current Conveyors*. Micronet Annual Workshop, Otawa, Abr. 1999.
5. Chang, S. T.; Hayes-Gill, B. R.; Paull, C. J. *Multi-Function Block for a Switched Current Field Programmable Analogue Array*. Midwest Symposium on Circuits and Systems, Ago. 1996.
6. Razavi, B. *RF Microeletronics*. Los Angeles : Prentice Hall PTR, 1998. 335p.

Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs

Edson L. Horta[1] and John W. Lockwood[2]

[1] Department of Electronic Engineering, Laboratory of Integrated Systems, EPUSP
edson-horta@ieee.org,
[2] Department of Computer Science, Applied Research Lab, Washington University,
lockwood@arl.wustl.edu,
http://www.arl.wustl.edu/arl/projects/fpx/parbit

Abstract. This paper presents an innovative way to deploy Bitstream Intellectual Property (BIP) cores. By using standard tools to generate bitstreams for Field Programmable Gate Arrays (FPGAs) and a tool called PARBIT, it is possible to extract a partial bitstream containing a modular component developed on one Virtex FPGA that can be placed or relocated inside another Virtex FPGAs. The methodology to obtain the BIP cores is explained, along with details about PARBIT and Virtex devices.

1 Introduction

Field Programmable Gate Arrays (FPGAs) available now contain millions of equivalent logic gates. Large systems are implemented on these FPGAs by integrating multiple intellectual property blocks. There are essentially three types of intellectual property blocks, also called IP cores [1]. The first, called soft cores, are blocks delivered as Verilog or VHDL code. The chip developer is responsible for synthesizing and implementing the logic into FPGA's logic. The second, called firm cores, are circuits described in a netlist format, mainly EDIF, and contain some placement information. These are used by the chip developer to implement the core in the FPGA. The third, called hard cores, are placed and routed blocks ready to be used inside the FPGA. They can not be altered by the chip developer.

This paper presents a new approach for the generation of IP cores: Bitstream Intellectual Property (BIP) cores. The BIP uses a partial bitstream configuration file to deliver the IP core. The full bitstream files can be generated with commercial tools already available from Xilinx. PARBIT, a tool capable of extracting a partial bitstream from full bitstream files, enables the chip developer to generate, develop and test an IP core in isolation on one FPGA then deploy it into a region of another.

The methodology presented in this paper can also be used to develop new reconfigurable modules, which can be loaded into a great variety of reconfigurable platforms, such as [2]. The Field Programmable Port Extender uses one FPGA

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 975–979, 2004.

to reconfigure the bitstream of another [3] [4]. The self-reconfiguring platform presented in [5] implements the reconfiguration control inside the same FPGA with its own logic resources. All of these approaches introduce a new level of flexibility for extensible systems. The configuration bitstream may come from a network or be compressed and stored on a device attached to the FPGA. The time required to generate new modules for such platforms can be reduced significantly using BIP cores.

2 Design Flow

PARBIT [6] processes two full bitstream files called original (containing the reconfigurable module) and target (containing the static logic). The tool has three operation modes: Slice, Block, and BIP. The slice and block modes can be used to perform partial run-time reconfiguration of an FPGA-based platform [7]. The BIP (Bitstream Intellectual Property) mode generates IP cores directly from bitstreams obtained through Xilinx tools [8]. To add a BIP core developed from an original device into a system being implemented on a target device, the source bitstream (containing the BIP core) is reformatted to configure a target device containing a reserved area for the BIP. Logic configuration and routing is specified for an area inside the CLB columns of the chip that exclude the top and bottom frame control bits of the FPGA's Input/Output Block (IOB). The area within the FPGA that defines the BIP core is extracted from the original design an merged into a target bitstream for another device of same family of FPGA. The tool generates the partial bitstream file containing the BIP core area and this file is used to program the core into the target device.

The interface between the reconfigurable module and the static logic is well defined. Modules can be placed into specific locations inside the FPGA. Interfaces between modules, called Gasket in [9], were redefined in [10] to be called a Bus Macro.

Just a few extra steps are needed to generate an original bitstream with the BIP core and a target bitstream that contains static logic with an open area reserved for the BIP core. The original bitstream is generated by the BIP core developer, and the target bitstream is generated by the final user.

The BIP core is defined as the reconfigurable area inside the original device, delimited by the coordinates: Start Row, End Row, Start Column and End Column. This FPGA will not have any logic outside the BIP core area.

Each Bus Macro, used to interface the BIP area with the static logic area of target device, bridges four signals, and occupies 8 CLBs. Four of the CLBs are located inside the reconfigurable area, and four outside. As shown in Figure 1, the connection points outside the BIP area are connected to the routed signals inside of the target device. The location of each one of the bus macro is fixed by a constraint command inside the UCF (User Constraints File) for the project.

A core developed with the BIP methodology consists of two files: the full original bitstream file, which is used by PARBIT to extract the configuration bits for the BIP core; and the Top Level VHDL, which contains the instantiation

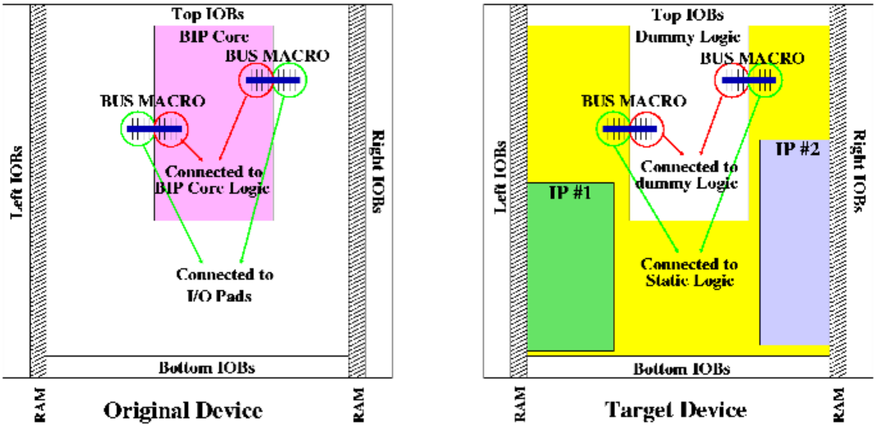


Fig. 1. BIP Original and Target

for the BIP core and the Bus Macro. The BIP width (CLB columns) and height (CLB rows) are added to VHDL in the form of comments. Likewise, the position of each interface signal, relative to the upper left corner of the BIP, is specified.

In order to perform run-time reconfiguration, the target bitstream would be reconfigured within the reserved area that receives the BIP core extracted from the original bitstream. The target bitstream contains static logic around the module that will not change during the downloading of the BIP core. This static logic may include other IPs, previously reconfigured into the FPGA, as shown in Figure 1.

It is assumed that an area was reserved that is the same size of the BIP core, and that the bus macros were placed in the same positions relative to the upper left corner of the BIP. It is not necessary to use the same device employed to generate the BIP bitstream file, provided that the BIP area fits inside the new device.

In order to keep the clock signals active to be used in the BIP core we insert Flip/Flops inside the reserved area to create the needed clock routing.

3 BIP Core Example

To demonstrate the methodology of using a BIP core, a reconfigurable calculator was implemented, based on an application note from Xilinx [10]. This calculator has three modules: adder, lcd_driver and pushbutton. In our example, the lcd_driver and the pushbutton modules are already placed in the target FPGA, and the adder module is used as a BIP core.

First, the size of the BIP core is determined and used to set the position of each bus macro. Six bus macros were used for the original 'calc' design: three of them are used between the lcd_driver and the adder; the other ones to interface the adder within the pushbutton module. For this example, the BIP core (adder) is confined in the region delimited by: Start Column = 5; End Column = 18; Start Row = 1; End Row = 16. Bus macros are placed on rows 5, 10 and 15, horizontally centered on the left and right sides of the BIP.

To implement and test the BIP adder core, an FPGA circuit was implemented with the adder connected through external bus macro signals to the FPGA I/O pins. The FPGA with the adder module was then synthesized, implemented and simulated with the BIP core in isolation.

Figure 2 shows the floorplan for the BIP core. This core is confined in a region containing 14 columns with 16 rows. The figure shows that only the bus macro and clock routing signals cross the boundary region. Because clock routing do not depend on the CLB configuration frames, they are allowed to cross the boundary region.

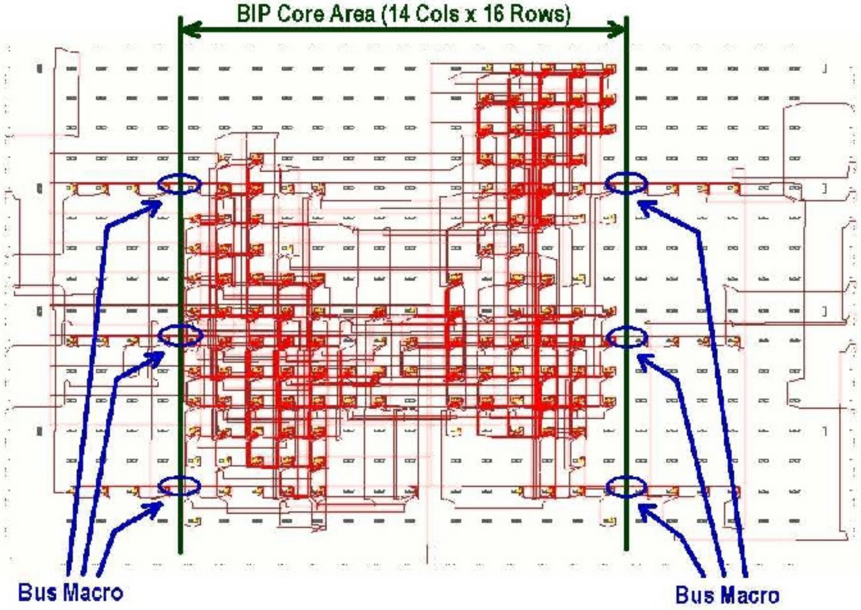


Fig. 2. BIP Core - XCV50

The target device was implemented with the lcd_driver and the pushbutton modules placed on the left and right sides of the chip, respectively. An empty space for the adder module was left by connecting a dummy circuit with Flip/Flops to internal bus macro signals. These flip/flops generate a clock routing template, which makes the clock signals available to any BIP core implemented in the target device. To ensure that all clock signals were routed across the 33 rows of the routed chip, a BIP core with 14 columns and 33 rows was inserted into this region.

4 Conclusions

A new way to generate an IP core was presented, utilizing existing Xilinx tools and PARBIT. The method uses bitstream files generated by commercial tools, and provides a secure method to deliver the IP.

When used with the Xilinx ISE V5.2i tools, the BIP core presented allows the mapper and place and route tools to run 8 times faster than without using a BIP core because the original circuit on the smaller XCV50 routed and placed much faster than when targeting directly the much larger XCV600 FPGA. Thus, the methodology presented in this paper can shorten the time necessary to design new reconfigurable modules for large FPGAs. With a BIP core, a small device can be used to generate and test a module before the implementation onto a larger target device.

References

1. VSIA: VSI Alliance Architecture Document. VSI Alliance, Online `http://www.-vsi.org/resources/techdocs/vsi-or.pdf` (1997)
2. Kalte, H., Porrmann, M., Rückert, U.: A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In: IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC), Hamburg, Germany (2002)
3. Lockwood, J.W.: An open platform for development of network processing modules in reprogrammable hardware. In: IEC DesignCon'01, Santa Clara, CA (2001) WB–19
4. Lockwood, J.W., Turner, J.S., Taylor, D.E.: Field programmable port extender (FPX) for distributed routing and queuing. In: ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000), Monterey, CA, USA (2000) 137–144
5. Blodget, B., James-Roxby, P., Kelle, E., McMillan, S., Sundararajan, P.: A self-reconfiguring platform. In: Field-Programmable Logic and Applications / The Roadmap to Reconfigurable Computing (FPL'2003), Lisbon, Portugal (2003) 565–574
6. Horta, E., Lockwood, J.W.: PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs). Technical Report WUCS-01-13, Washington University in Saint Louis, Department of Computer Science (July 6, 2001)
7. Horta, E.L., Lockwood, J.W., Taylor, D.E., Parlour, D.: Using PARBIT to implement Partial Run-Time Reconfigurable Systems . In: 12th International Conference on Field Programmable Logic and Applications, Montpellier, France (2002)
8. Xilinx, I.: Foundation series user guide. http://toolbox.xilinx.com/docsan /xilinx6/pdf/docs/fsu/fsu.pdf (2004)
9. Horta, E.L., Lockwood, J.W., Taylor, D.E., Parlour, D.: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In: Design Automation Conference (DAC), New Orleans, LA (2002)
10. Lim, D., Peattie, M.: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations. Xilinx XAPP290 (2002)

Real-Time Computation of the Generalized Hough Transform

Tsutomu Maruyama

Institute of Engineering Mechanics and Systems, University of Tsukuba
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 JAPAN
maruyama@darwin.esys.tsukuba.ac.jp

Abstract. In this paper, we describe a circuit for real-time computation of the Generalized Hough Transform (GHT). The GHT is a technique to find out arbitrary shapes in an image. The GHT is very robust to occlusion and noises, but requires large amount of memory and long computation time. In the GHT, a point is chosen as a reference point of a shape, and the point is searched using edge points in an image. In order to implement the GHT on one FPGA, we first look up regions that may include the reference points, and then search the points in those regions. With this two phase search, the circuit on XC2V6000 can find 112 kinds of shapes smaller than 256×256 pixels in an image (640×480 pixels) in real-time (more than 25 frames per second). The 112 kinds of shapes can be used to find an object of arbitrary distance and angle in 3-D space.

1 Introduction

The Generalized Hough Transform (GHT) [1] is a technique to find out an arbitrary shape in an image. The procedure in the GHT is very simple and regular, and suitable for parallel processing by hardwares. The computational complexity is, however, very large, and furthermore, large amount of memory required by the GHT makes it difficult to realize real-time processing on compact circuits. In order to accelerate the performance, many hardware systems have been researched[2][3][4][5]. Their performances were, however, slower than video-rate, or the target image sizes were smaller than standard image sizes. In this paper, we describe an implementation method to realize real-time processing of the GHT on standard image sizes with a compact circuit. In our method, regions that may include the reference points (center of shape) are searched first, and then the points are searched in those regions by a pipelined circuit.

2 The Generalized Hough Transform (GHT)

In the GHT, as shown in Figure 1(a), one pixel is chosen to refer a shape (*center* of the shape), and for each feature point of the shape, relative position of the *center* (r, θ) from the point is registered in a table (called *R-table*) using φ as index in advance. Then, for each edge point in a given image, (1) φ is obtained

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 980–985, 2004.

by some methods, (2) relative positions of the *center* from the point are obtained from *R-table* using φ as index, (3) positions of the *center* are calculated by the relative positions, and (4) values in a memory (called *Accumulator Array*) are incremented using the positions of the *centers* as addresses. After processing all edge points in the given image, peaks in *Accumulator Array* are chosen as *centers* of the shapes.

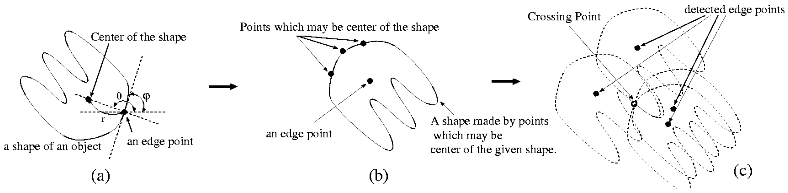


Fig. 1. The Generalized Hough Transform

Under noisy conditions, it it not easy to obtain correct φ, which may cause errors in finding *centers* of shapes. In our approach, all positions which may be the *center* of the shape are considered as candidates of the *center* (Figure 1(b)). Then, all values in *Accumulator Array* which correspond to the candidates are incremented. After processing all edge points, peaks (namely crossing points of the candidates (Figure 1(c))) are chosen as *centers* of the shapes. This method requires much more increment and memory access operations than normal GHT, but much more robust to noises.

In order to increment values in *Accumulator Array* for all the candidates, we prepare *Shape Table* (two-dimensional table) as shown in Figure 2 instead of *R-table*. In *Shape Tables*, values on positions which may be the *center* of the shape are one, and all other values are zero. With one *Shape Table*, we can detect one kind of shape in a image. Therefore, we need N *Shape Tables* and N *Accumulator Arrays* to detect N kinds of shapes of an object in 3-D space. For each edge point in an image, all values in k-th *Shape table* ($k = 1, N$) are shifted according to the address of the edge point, and added to k-th *Accumulator Array.* Then, peaks in k-th *Accumulator Array* are considered as *centers* of k-th shape.

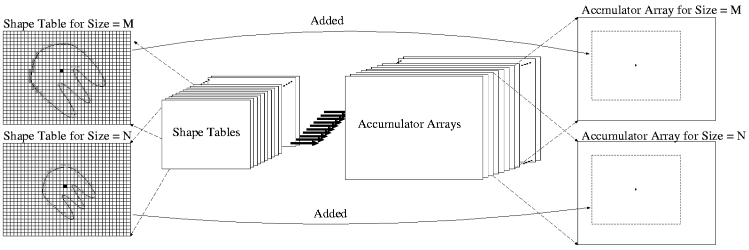


Fig. 2. Shape Tables and Accumulator Arrays

3 Two Phase Search of the GHT

Our method consists of two phases below.

1. In the first phase, M regions that may include *centers* are searched by the GHT using *Reduced Accumulator Arrays* which can be implemented on one FPGA.
2. In the second phase, *centers* are searched in the M regions by the GHT using M accumulator sub-arrays (called *Sub-Arrays*) which have same size with the regions.

Therefore, the GHT is applied twice in our method, but these two phases can be pipelined as described below.

In Figure 3, the size of *Accumulator Array* is reduced to 40×32 from 640×480. Each value in the *Reduced Accumulator Array* are the sum of 16×16 values in the original *Accumulator Array.* The size of *Shape Table* is also reduced to 17×17, and each value in *Reduced Shape Table* are the sum of 16×16 values in the original *Shape Table.*

In the first phase, for each edge point in an image, values in *Reduced Shape Table* are shifted according to the address of the edge point (x, y), and added to *Reduced Accumulator Array.* Then, M entries which give larger values in the *Reduced Accumulator Array* (four gray squares in Figure 3) are chosen as candidates that may include *centers* of the shapes.

In the second phase, for each edge point in the image, values in the original *Shape Table* are shifted according to the address of the edge point (x, y), and added to M *Sub-Arrays*, and peaks in each region are selected as *centers.*

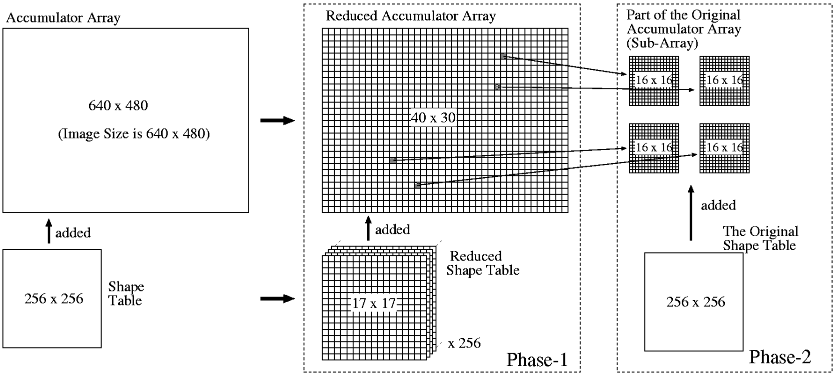


Fig. 3. Reduced Arrays and Tables

In order to add values in *Reduced Shape Table* to *Reduced Accumulator Array* correctly, we need to prepare 16×16 patterns for one *Reduced Shape Table.* Suppose that the address of an edge point is (x, y). Then, values which are added to *Reduced Accumulator Array* have to be

$$RS(k,l) = \sum_{i=-x\%16+k\times16}^{-x\%16+k\times16+15} \sum_{j=-y\%16+l\times j}^{-y\%16+l\times j+15} S(i,j)$$

where $S(i,j)$ are values in original *Shape Table*. This means that $RS(k,l)$ becomes different according to $x\%16$ and $y\%16$. One of these 16×16 patterns is selected according to $x\%16$ and $y\%16$, and added to *Reduced Accumulator Array*. When $x\%16! = 0$ and $y\%16! = 0$, the size of the pattern becomes 17×17.

4 Parallel and Pipeline Processing

In order to find an object in 3-D space, we need to detect many kinds of shapes. Suppose that the maximum size of *Shape Table* is $S \times S$. Then, we need to detect $S/2$ kinds of shapes of an object to find the object in arbitrary distance because the number of shapes which are smaller than $S \times S$ is $S/2$. Therefore, the two phase search by the GHT has to be applied to $S/2$ kinds of shapes.

Figure 4 shows a block diagram of the current implementation of a circuit for the two phase search, and Figure 5 shows the parallel and pipeline processing on the circuit. In Figure 4,

1. An image taken by a camera is sent to one of two external memory banks (*I1* and *I2*) (while an image is sent from the camera to *I1* (*I2*), the previous image in *I2*(*I1*) is accessed by the FPGA),
2. Edge points in an image are detected by *Edge Detection Unit*, and the addresses of the points are stored in a memory bank *E*,
3. The addresses of edge points are read in from *E*, and given to *Accumulator Unit 1* (*Reduced Accumulator Arrays* for Phase-1).
4. According to the addresses, *Reduced Shape Tables* are read out from *R*, and added to the arrays.
5. After all edge points are processed, *Find MAX Unit* reads out values in *Reduce Accumulator Arrays* and finds out *M* regions which have larger values.
6. The addresses of edge points are read in again, and given to *Accumulator Unit 2* (*Sub-Arrays* for Phase-2).
7. According to the addresses, *Shape Tables* are read out from *S1,S2,S3,S4*, and added to *Sub-Arrays.*
8. After all edge points are processed, *Find PEAK Unit* reads out values in *Sub-Arrays*, and finds out peaks (which are considered as the *center* of shapes).

The parallel and pipeline processing becomes as follows (Figure 5).

1. 112 kinds of shapes (not 128 by limitation of the external memory size) are searched in total, and 8 kinds of shapes are searched in parallel. Therefore, two phase search is repeated 14 times on the pipelined circuit.
2. Addresses of edge points are read out 15 times, and given to *Accumulator Unit 1* and *Accumulator Unit 2.*
3. *Accumulator Unit 1* and *Accumulator Unit 2* can not be active with *Find MAX Unit* and *Find Peak Unit* at the same time, because they need dual-port access to same memory banks.

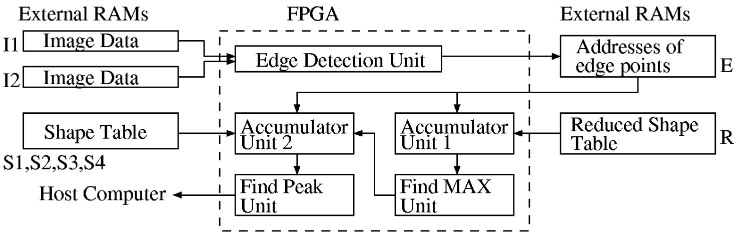


Fig. 4. Block Diagram of the Circuit

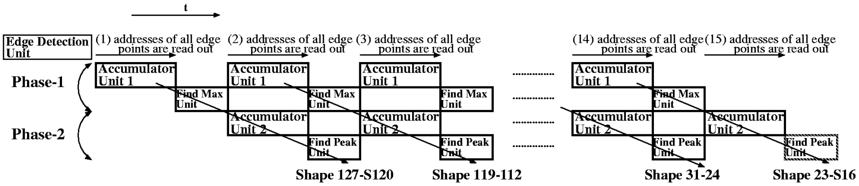

t
Edge Detection Unit
(1) addresses of all edge points are read out
(2) addresses of all edge points are read out
(3) addresses of all edge points are read out
(14) addresses of all edge points are read out
(15) addresses of all edge points are read out
Phase-1
Phase-2
Accumulator Unit 1
Find Max Unit
Accumulator Unit 2
Find Peak Unit
Shape 127-S120
Shape 119-112
Shape 31-24
Shape 23-S16

Fig. 5. Pipeline Processing

5 Performance

The circuit was implemented on ADM-XRC-II by Alpha Data with one Xilinx XC2V6000. This board has eight external memory banks (two banks by an additional SRAM board). This eight memory banks is very important, because all *Tables* are stored in the memory banks, and the number of the memory banks limits the number of *centers* which can be found by the approach.

The circuit runs at 66MHz, and 75% of slices in XC2V6000, 122 Block RAMs are used. Table 1 shows the performance when the rate that a pixel in an image is an edge points is less than 2.84%, 4% and 6% (when the rate is less than 2.84%, the loading time of eight patterns in Phase-1 becomes the bottle-neck). As shown in Table 1, the performance depends on the rate (the number of edge points in an image). The rate can be controlled by a threshold which is used to decide whether a pixel is an edge point in *Edge Detection Unit*. Lower threshold generates more edge points, but many noises are included. In this sense, the circuit can process more than 25 frames in one second under proper range of the threshold.

Table 1. Performance of the Circuit (66MHz)

rate of edge points	clock cycles per image	frames per second
less than 2.84%	2537168	26.3
4%	3451088	19.3
6%	5017808	13.3

6 Conclusions

In this paper, we described a compact circuit for a real-time computation of the Generalized Hough Transform. The major limitation in our method is that only four *centers* for each eight kinds of shapes can be found because of the limited memory band-width. Another major drawback is that we may miss the *center* of a shape which locates just on the boarder of the divided regions. These problems can be improved by doubling the memory access frequency, and overlapping the regions. We are now improving the circuit to achieve faster operational frequency, and to work with other circuits such as line detection and optical-flow to detect objects in 3-D space.

References

1. D.H. Ballard, "Generalizing the Hough transform to detect arbitrary shapes" Pattern Recognition 13 2 (1981), pp. 111-122.
2. Marco Ferretti, "The Generalized Hough Transform on Mesh-Connected Computers", Journal of Parallel and Distributed Computing, Volume 19, 1993, pp. 51-58
3. N. Guil and E. L. Zapata, "A Parallel Pipelined Hough Transform", Euro-Par Vol.II, 1996, pp. 131-138.
4. Meribout, M., Nakanishi, M., Ogura, T., "A parallel algorithm for real-time object recognition", Pattern Recognition 35(9), 2002, pp. 1917-1931.
5. Strzodka, R., Ihrke, I., Magnor, M., "A Graphics Hardware Implementation of the Generalized Hough Transform for Fast Object Recognition, Scale, and 3D Pose Detection", Intl. Conf. on Image Analysis and Processing, 2003 pp. 188-193.

Minimum Sum of Absolute Differences Implementation in a Single FPGA Device

Joaquín Olivares[1], Javier Hormigo[2], Julio Villalba[2], and Ignacio Benavides[1]

[1] Dept. of Electrotechnics and Electronics, University of Córdoba, Spain
[2] Dept. of Computer Architecture, University of Málaga, Spain

Abstract. Most of Block based motion estimation algorithms are based on computing the sum of absolute differences (SAD) between candidate and reference block. In this paper a FPGA design for fast computing of the minimum SAD is proposed. Thanks to the use of the on–line arithmetic (OLA) two goal are achieved: it is possible to implement a full 16×16 macroblock SAD in a single FPGA device and it permits us to speed up the computation by early truncation of the SAD calculation. Reconfigurable devices allows us to change 8×8 or 16×16 pixels per block models. Comparison with other related works are provided.

1 Introduction

Block based motion estimation is one of the critical task in today video compression standards such as H.26x, MPEG-1, -2 and -4 standards. Motion estimation is defined as searching the best similar block in previous frame for the block in current frame. The most commonly used metric to calculate the distortion is the Sum of Absolute Differences (SAD) [1], which adds up the absolute differences between corresponding elements in the candidate and reference block.

In spite of inherent parallelism in SAD, the full parallel implementation requires a large amount of operands for the typical block size (16×16 pixel macroblock needs 512 8–bit operands). Due to the large amount of hardware, the computation of the SAD in only one row of a macroblock (16×1) is implemented on a FPGA device in [1]. In [2] four FPGA chips with 1234 I/O pins each are used for a completely parallel design. On the other hand, the use of on–line arithmetic (OLA) for motion estimation is proposed in [3] to speed up the computation by early truncation of the SAD calculation. In [3], a serial architecture (pixel by pixel) for 4×4 blocks is proposed based on ASIC implementation.

We present a new parallel on–line architecture to carry out the minimum SAD computation for 16×16 macroblocks. OLA works in a digit-serial mode which fits very well the FPGA architecture characteristic. Thus, the hardware requirements are drastically reduced which allows us to implement the complete SAD operation for a macroblock in a single FPGA device.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 986–990, 2004.

2 Online Computation of the Minimum SAD

On-line arithmetic algorithms operate in a digit-serial manner, beginning with the most significant digit (MSD) [4]. The value δ is known as on-line delay, since $j+\delta$ digits of the input operands are needed to generate the j-th digit of the result. Thus, successive operation can execute in an overlapped manner as soon as δ input digits are available. The MSD first mode of computation requires the use of redundant representation system. Signed-digit (SD) representation system is used in this paper. In radix-2 SD representation, the digit set is $\{-1, 0, 1\}$ which are represented by two bits ({10, 11 or 00, 01} respectively).

In our design the SAD computation and comparison operation is performed using on-line arithmetic.The on–line comparator that we design for SAD comparison is described in [5].

The SAD adds up the absolute differences between corresponding elements in the candidate and reference block.

$$SAD = \sum_{i=1}^{N} \sum_{j=1}^{N} |c_{i,j} - r_{i,j}| \tag{1}$$

where $r_{i,j}$ are the elements of the reference block and $c_{i,j}$ the elements of the candidate block. Thus, the computation of the SAD is divided in three steps:

Conversion to SD representation and difference computation: In radix-2 SD representation, each digit is composed by two bits, the first one negative weighted and the second one positive weighted. Thus, a SD number can be interpreted as the difference of two unsigned numbers. This property is used to perform simultaneously the conversion of each pixel value to SD and the difference, with no computational cost. In that way, each digit of the value $d_{i,j} = c_{i,j} - r_{i,j}$ is obtained in SD representation by only taking the corresponding bit of $c_{i,j}$ as the positive weighted and the corresponding bit of $r_{i,j}$ as the negative weighted, since $c_{i,j}$ and $r_{i,j}$ are unsigned numbers.

Absolute value: To compute the absolute value of $d_{i,j}$, the sign of this value have to be changed if $d_{i,j}$ is negative. In SD the negation operation is performed by interchanging both bits of each digit. Since MSD-first mode of computation is used, the sign detection of $d_{i,j}$ is performed on-the-fly by checking if the first non zero digit of $d_{i,j}$ is positive (01) or negative (10).

Sum of absolute differences: The absolute difference of all the pixels corresponding to the current and reference blocks are computed in parallel. Then N^2 absolute difference blocks are required. An on-line adder tree is used to obtain the sum of all $d_{i,j}$ values.

The number of addition steps of the complete adder tree is $\log_2(N^2)$. In radix-2 SD representation the on–line delay of the addition is two. Nevertheless in our case, the carry bit is used as MSD of the results and this digit is obtained one cycle before. Therefore, the on-line delay of the complete adder-tree is $2\log_2(N^2)$, but the first digit of the results is obtained $\log_2(N^2)$ cycles earlier.

3 FPGA Implementation of the SAD Processor

Figure 1 presents the architecture of the design corresponding to the SAD processor. The absolute value of the differences is computed for each pair of pixels ($|c_{i,j} - r_{i,j}|$) and their summation is calculated on the N^2–Operand OLA adder. The result is stored digit by digit on SADc register, and it is simultaneously compared with the corresponding digit of SADr in the comparator (COMP) [5]. If at any cycle the condition SADc > SADr is detected, the computation is stopped and a new candidate block is required. Otherwise, if the condition SADc < SADr is verified, SADc is stored in SADr when the less significant digit of the SAD is calculated.

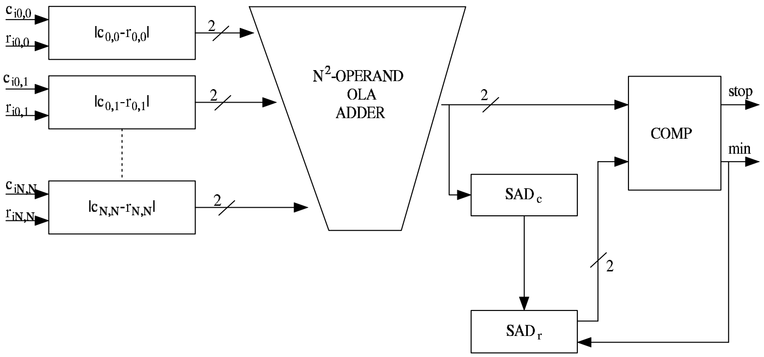


Fig. 1. SAD processor architecture

The timing of the computation for the 4 × 4 SAD processor is shown in figure 2.

The worst case takes place when a new minimum SAD is found, and then 21 cycles are required to the full process, where the last one is to store SADc in SADr. However, as figure 2 shows, a new SAD computation can start after 16 cycles (after the 8 digits and 8 zeroes are introduced) and then, this period of time is the maximum between two consecutive SAD computation. This period is reduced if the candidate SAD is rejected before. In the best case, it happens after the analysis of the MSD of the candidate SAD, that is after 9 cycles. Therefore, the number of cycles for a SAD computation and comparison is between 9 and 16 cycles for a 4×4 SAD processor. This period is in the range between 13 and 20 cycles for 8×8 block size and between 17 and 24 cycles for 16×16 block size.

The design has been implemented on the Xilinx SPARTAN-II and VIRTEX-II FPGA families, using the Xilinx ISE Series 5.2i. The main results of the implementation is shown in table 1. The ratio area/number of pixels is relatively low, due to the serial-digit nature of on-line computation. The maximum clock frequency is independent of the block size.

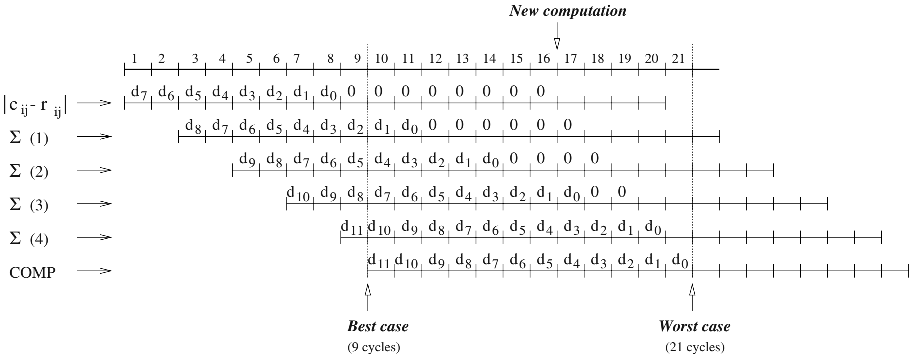

New computation
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
|c ij - r ij| → d7 d6 d5 d4 d3 d2 d1 d0 0 0 0 0 0 0 0 0
Σ (1) → d8 d7 d6 d5 d4 d3 d2 d1 d0 0 0 0 0 0 0
Σ (2) → d9 d8 d7 d6 d5 d4 d3 d2 d1 d0 0 0 0 0
Σ (3) → d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0 0 0
Σ (4) → d11 d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0
COMP → d11 d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0
Best case
(9 cycles)
Worst case
(21 cycles)

Fig. 2. Timing of the 4 × 4 SAD processor

Table 1. Area and clock frequency corresponding to different FPGA implementations

	SPARTAN-II	VIRTEX-II
Block size	Area (4 inputs-LUTs)	
4x4 (16 pixels)	246	241
8x8 (64 pixels)	603	595
16x16 (256 pixels)	1982	1945
	Maximum Frequency(MHz)	
	231.24	424.99

4 Comparison

In [1] the computation of the SAD for 16 pixels (SAD16) is implemented on a FPGA device. The design is based on carry–save adders which perform the computation in parallel over all the digit of the data. According to the authors, the design is synthesized using FPGA Express from Synopsis by targeting the FLEX20KE family from Altera, obtaining an area of 1699 LUTs and a maximum frequency of 197 MHz with a latency of 19 cycles (96ns). The results of our implementation using the VIRTEX-II family is used for comparison, since it has similar performance. The worst case for our equivalent design (4×4) requires 21 cycles to compute SAD16 plus comparison with the previous minimum and store it, which means 49ns at the frequency of 425 MHz. Besides, our design only requires 241 LUTs, that is seven times less area than in [1].

The authors give some notes about how to extend the design to compute a 16×16 SAD in two ways. The first one is based on using 16 SAD16 units (one for each row) and a final adder tree. They estimate that 27 clock cycles are required. Nevertheless, the number of LUTs for the design is close to 30000, which does not seem feasible for the current FPGA devices. Our 16×16 design requires only 1945 LUTs, which is easily implemented on a single FPGA device. The second approach is based on reusing the SAD16 units to compute the SAD of all the 16 rows which are buffered to finally add up them. This involves 42 clock cycles with a larger area size due to buffering and the fact that longer binary data (16 bit instead of 12 bits) must be supported. Moreover, the intrinsic pipeline behavior of the SAD16 units is eliminated. For a similar area, our design computes a SAD each 24 cycles for the worst case.

On the other hand, the solution proposed in [2] involves the use of four Altera STRATIX EP1S80 devices with 1234 I/O pins. This design uses 7765 LCs and requires 29 cycles for a SAD computation at a frequency of 380 MHz. It means that our design obtains better time performance using much less hardware requirement.

We would like to emphasize that the previous comparisons involve our worst case. However, our best case involves only 9 cycles for 4 × 4 SAD and 17 cycles for 16 × 16 SAD (see section 4). On the other hand, our results include the comparison which, for the designs of [1] and [2], involves several clock cycles (due to carry propagation).

5 Conclusion

In this paper a FPGA implementation of the minimum SAD computation has been stated. The computation is carried out by using on–line arithmetic. The different operations involved in the SAD computation have been efficiently adapted to on–line arithmetic. This allows us to implement the design in a single FPGA device and to speed up the computation. Furthermore, the FPGA implementation of the design makes possible to reconfigure the hardware to deal with 8 × 8 and 16 × 16 pixel blocks according to the MPEG-4 standard requirements. We show the delay and area details of the implementation for different block sizes and we provide comparison with other current related works, which shows the benefits of our design.

References

1. S. Wong, S. Vassiliadis, S. Cotofana, “A Sum of Absolute Differences Implementation in FPGA Hardware”, 28th Euromicro Conference, pp.183–188, 2002.
2. S. Wong, B. Stougie, S. Cotofana, “Alternatives in FPGA-based SAD Implementations”, Proc. IEEE International Conf. on Field-Programmable Technology, pp. 449–452, 2002.
3. C. Su and C. Jen, “Motion estimation using MSD-first processing”, IEE Proc. Circuits Devices System, vol. 150, No. 2, pp. 124–133, 2003.
4. M. Ercegovac and T. Lang, “On-Line Arithmetic for DSP Applications”, 32nd Midwest Symposium on Circuits and Systems, pp. 365–368, 1989.
5. Hormigo, J.; Olivares, J.; Villalba, J.; Benavides, I.; “New on-line Comparator with no on-line delay”, 8th World Multiconference on Systemics, Cybernetics and Informatics, 2004 (acepted).

Design and Efficient FPGA Implementation of an RGB to YCrCb Color Space Converter Using Distributed Arithmetic

Faycal Bensaali and Abbes Amira

School of Computer Science, Queen's University of Belfast,
Belfast BT7 1NN
{f.bensaali, a.amira}@qub.ac.uk

Abstract. This paper presents two novel architectures for efficient implementation of a Color Space Converter (CSC) suitable for Field Programmable Gate Array (FPGAs) and VLSI. The proposed architectures are based on Distributed Arithmetic (DA) ROM accumulator principles. The architectures have been implemented and verified using the Celoxica RC1000-PP FPGA development board. In addition, they are platform independent and have a low latency (8 cycles). The first architecture has a throughput of height, while the second one is fully pipelined and has a throughput of one and capable of sustained data rate of over 234 mega-conversions/seconds.

1 Introduction

Color spaces provide a standard method of defining and representing colors. There are many existing color spaces and most of them represent each color as a point in a 3D coordinate system. Each color space is optimized for a well-defined application area [1]. The three most popular color models are RGB (used in computer graphics); YIQ, YUV and YCrCb (used in video systems); and CMYK (used in color printing). All of the color spaces can be derived from the RGB information supplied by devices such as cameras and scanners. Processing an image in the RGB color space, with a set of RGB values for each pixel is not the most efficient method. To speed up some processing steps many broadcast, video and imaging standards use luminance and color difference video signals, such as YCrCb, making a mechanism for converting between formats necessary. Several cores for RGB to YCrCb conversion can be found in the market, which have been designed for FPGA implementation, such as the cores proposed by Amphion Ltd [2], CAST .Inc [3] and ALMA .Tech [4]. This paper proposes the use of FPGA as a low cost accelerator for two RGB to YCrCb Color Space Conversion based architectures using DA ROM accumulator principles. The two proposed architectures are based on serial and parallel manipulation of pixels. The composition of the rest of the paper is as follows. The mathematical backgrounds and the descriptions of the two proposed architectures are given in sections 2 and 3. Section 4 is concerned with the results and analysis for the hardware implementations. Finally concluding remarks are given in section 5.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 991–995, 2004.

2 Architecture Based Serial Manipulation Approach

The CSC core implemented is based on the following mathematical formula to convert data from one space to another:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \end{pmatrix} \times \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ 1 \end{pmatrix} \tag{1}$$

Where C_i $(0 \leq i \leq 2)$ and B_i $(0 \leq i \leq 3)$ represent the input and output color components respectively, A represents the constant conversion matrix. C_i can be computed using the following equation:

$$C_i = \sum_{k=0}^{3} A_{ik} \times B_k \tag{2}$$

Where $\{A_{ik}\}$'s are L-bits constants and $\{B_k\}$'s are written in the unsigned binary representation as shown in equation 3:

$$B_k = \sum_{m=0}^{W-1} b_{k,m} \times 2^m \tag{3}$$

Where $b_{k,m}$ is the m^{th} bit of B_k , which is zero or one, W is the word-length. Since all the components are in the range of 0 to 255, 8 bits are enough to represent them $(W = 8)$. Substituting 3 in 2,

$$C_i = \sum_{k=0}^{3} A_{ik} \times (\sum_{m=0}^{7} b_{k,m} \times 2^m) = \sum_{m=0}^{7} \sum_{k=0}^{3} A_{ik} \times b_{k,m} \times 2^m \tag{4}$$

C_i can be computed as:

$$C_i = \sum_{m=0}^{7} Z_m \times 2^m \quad ; \qquad Z_m = \sum_{k=0}^{3} A_{ik} \times b_{k,m} \tag{5}$$

The idea is that since the term Z_m depends on the $b_{k,m}$ values and has only 2^4 possible values, it is possible to precompute and store them in ROMs. An input set of 3 bits $(b_{0,m},\ b_{1,m}, \ldots b_{3,m})$ is used as an address to retrieve the corresponding Z_m values as illustrated in table 1. Since the last element of the vector B is equal to 1, equation 5 can be rewritten as:

$$C_i = \sum_{m=0}^{7} Z^*_m \times 2^m + A_{i3} \quad ; \qquad Z^*_m = \sum_{k=0}^{2} A_{ik} \times b_{k,m} \tag{6}$$

It is worth mentioning that the size of the ROMs has been reduced to 2^3.

Table 1. Content of the ROM i

$b_{0,m}$	$b_{1,m}$	$b_{2,m}$	**The Content of the ROM i**
0	0	0	0
0	0	1	A_{i2}
0	1	0	A_{i1}
0	1	1	$A_{i1} + A_{i2}$
1	0	0	A_{i0}
1	0	1	$A_{i0} + A_{i2}$
1	1	0	$A_{i0} + A_{i1}$
1	1	1	$A_{i0} + A_{i1} + A_{i2}$

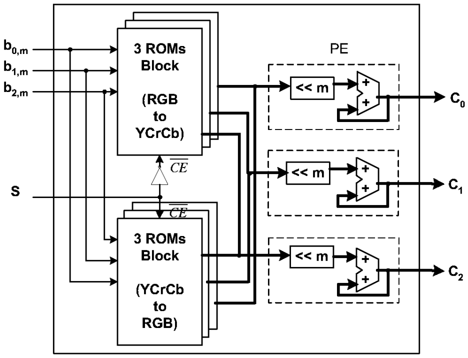


Fig. 1. Serial CSC based DA architecture for RGB ↔ YCrCb conversions

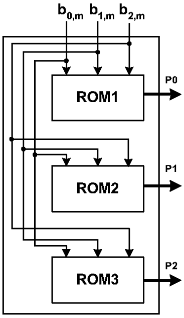


Fig. 2. ROMs block structure

Figures 1 and 2 show the proposed architecture and the ROM blocks structure respectively.

The proposed architecture consists of three identical Processing Elements (PEs) and two ROMs blocks. Each PE comprises a parallel ACCumulator (ACC) and a right shifter and each ROMs block consists of three ROMs with the size of 2^3 each. The ROM's content is different and depends on the matrix A coefficients, which depend on the conversion type. S is used to select the appropriate CSC.

3 Architecture Based Parallel Manipulation Approach

Consider an $N \times M$ image (N: image height, M: image width). Let represent each image pixel by b_{ijk}, $(0 \leq i \leq N-1, 0 \leq j \leq M-1, 0 \leq k \leq 2)$ where:

$$\begin{cases} b_{ij0}= R'_{ij} \text{ the red component of the pixel in row } i \text{ and column } j \\ b_{ij1}=G'_{ij} \text{ the green component of the pixel in row } i \text{ and column } j \\ b_{ij2}= B'_{ij} \text{ the blue component of the pixel in row } i \text{ and column } j \end{cases} \quad (7)$$

The c_{ijk} elements (the output image color space components) can be computed using the following equation:

$$c_{ijk} = \sum_{m=0}^{3} A_{km} \times b_{ijm} \quad (0 \leq i \leq N-1, 0 \leq j \leq M-1, 0 \leq k \leq 2) \tag{8}$$

Where $\{A_{km}\}$'s are L-bits constants and $\{b_{ijm}\}$'s are written in the unsigned binary representation as shown in equation 9:

$$b_{ijm} = \sum_{l=0}^{W-1} b_{ijm,l} \times 2^{l} \quad (0 \leq i \leq N-1, 0 \leq j \leq M-1, 0 \leq m \leq 2) \tag{9}$$

Using the same development in the previous section, equation 8 can be rewritten as:

$$c_{ijk} = \sum_{l=0}^{7} Z_l^{*} \times 2^{l} + A_{k3} \quad ; \qquad Z_l^{*} = \sum_{m=0}^{2} A_{km} \times b_{ijm,l} \tag{10}$$

Equation 10 can be mapped into the proposed architecture as shown in Figure 3.

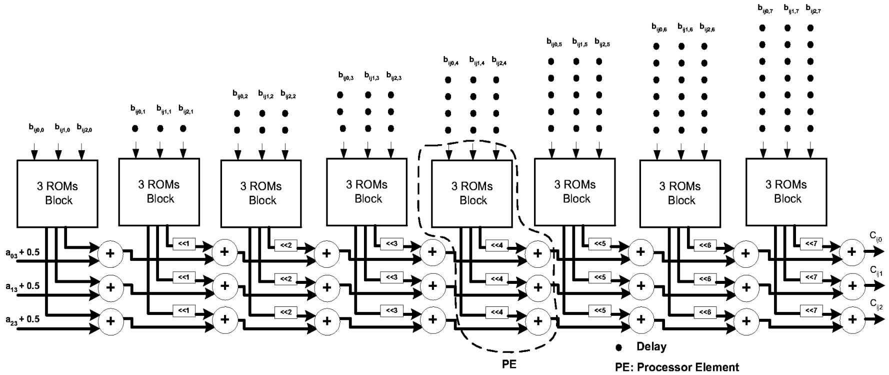

$b_{ij0,0}$ $b_{ij1,0}$ $b_{ij2,0}$
$b_{ij0,1}$ $b_{ij1,1}$ $b_{ij2,1}$
$b_{ij0,2}$ $b_{ij1,2}$ $b_{ij2,2}$
$b_{ij0,3}$ $b_{ij1,3}$ $b_{ij2,3}$
$b_{ij0,4}$ $b_{ij1,4}$ $b_{ij2,4}$
$b_{ij0,5}$ $b_{ij1,5}$ $b_{ij2,5}$
$b_{ij0,6}$ $b_{ij1,6}$ $b_{ij2,6}$
$b_{ij0,7}$ $b_{ij1,7}$ $b_{ij2,7}$
3 ROMs Block
<<1
<<2
<<3
<<4
<<5
<<6
<<7
a_{03} + 0.5
a_{13} + 0.5
a_{23} + 0.5
+
C_{ij0}
C_{ij1}
C_{ij2}
PE
● Delay
PE: Processor Element

Fig. 3. Proposed parallel architecture based on DA principles

The architecture consists of 8 identical $PE_n s$ $(0 \leq n \leq 7)$. Each PE_n comprises three parallel signed integer adders, three n right shifters and one ROMs block, which have the structure as shown in figure 2. It is worth noting that the architecture has a Latency of $W = 8$ and a throughput rate equal to 1.

4 Hardware Implementation

The two proposed architecture have been implemented and verified using the Celoxica RC1000-PP PCI based FPGA development board equipped with a

Xilinx XCV2000E Virtex FPGA [5,6]. In order to make a fair and consistent comparison with the existing FPGA based color space converters, the XCV50E-8 FPGA device has been targeted. Table 2 illustrates the performances obtained for the proposed architecture in terms of area consumed and speed which can be achieved.

Table 2. Performance comparison with existing CSC cores

Design Parameters	Slices	Speed (MHz)	Throughput (vector/clock cycle)
Proposed architecture (1)	70	128	8
Proposed architecture (2)	193	234	1
CAST.Inc [3]	222	112	1
ALMA. Tech [4]	222	105	1
Amphion Ltd [2]	204	90	1

The proposed architecture based serial manipulation approach shows significant improvements in comparison with the existing implementations [2,3,4] in terms of the area consumed and the maximum running clock frequency, while the second architecture outperforms the existing ones in term of the number of conversions per second.

5 Conclusion

RGB ↔ YCrCb conversions require enormous computing power. However, novel, scalable and efficient architectures based on DA principles have been reported in this paper.The implementation result shows the effectiveness of the DA approach. The performance in terms of the area used and the maximum running frequency of the proposed architectures has been assessed and has shown that the proposed systems requires less area and can be run with a higher frequency when compared with existing systems.

References

1. R.C. Gonzalez and R.E. Woods, "Digital Image Processing," Second Edition, Printice Hall Inc, 2002.
2. Datasheet (www.amphion.com), "Color Space Converters," *Amphion semiconductor Ltd,* DS6400 V1.1, April 2002.
3. Application Note (www.cast-inc.com),"CSC Color Space Converter," *CAST Inc,* April 2002.
4. Datasheet (www.alma-tech.com), "High Performance Color Space Converter," *ALMA Technologies,* May 2002.
5. Datasheet (www.celoxica.com), "RC1000 Reconfigigurable Hardware Developement Platform," *Celoxica Ltd,* 2001.
6. Xilinx, Inc, http://www.xilinx.com

High Throughput Serpent Encryption Implementation

Jesús Lázaro, Armando Astarloa, Jagoba Arias, Unai Bidarte, and Carlos Cuadrado

Escuela Superior de Ingenieros,
University of the Basque Country
Alameda Urquijo s/n
48013 Bilbao, Spain
{jtplaarj, jtpascua, jtparpej, jtpbipeu, jtpcuvic}@bi.ehu.es
http://www.ingenierosbilbao.com

Abstract. Very high speed and small area hardware architectures of the Serpent encryption algorithm are presented in this paper. The Serpent algorithm was a submission to the National Institute of Technology (NIST) as a proposal for the Advanced Encryption Standard (FIPS-197). Although it was not finally selected, Serpent was considered very secure and with a high potential in hardware implementations. Among others, a fully pipelined Serpent architecture is described in this paper and when implemented in a Virtex-II X2C2000-6 FPGA device, it runs at a throughput of 40 Gbps.

1 Introduction

In the past few years several calls for Algorithms have been made in the field of encryption and security. Some of these were looking for a new private key block ciphers as with the Advanced Encryption Standard (AES) project [1]. Other projects included also hash algorithms, digital signatures, stream ciphers and public key algorithms, as the New European Schemes for Signatures, Integrity and Encryption (NESSIE) [2]. Cipher Algorithms that have been submitted to any of these calls include MISTY [3], *Camellia* [4], SHACAL [5], MARS [6], RC6 [7], Rijndael [8], Serpent [9] and Twofish [10].

The Rijndael algorithm was selected for the AES contest while NESSIE is still open. In this context, Serpent was rejected from the AES context despite being highly secure -it was reported that “Serpent appears to have a high security margin” in contrast to “Rijndael appears to have an adequate security margin” [1]- and its supposed high throughput in hardware implementations. This article presents a highly efficient hardware implementation. It is faster than any other encryption hardware implementations of the algorithms mentioned above while it uses the smallest silicon area in the integrated circuit leading to an outstanding efficiency.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 996–1000, 2004.

2 Serpent Hardware Implementation

The key points in the hardware implementations are the four input S-Boxes, and the three input linear transformation. Taking into account the internal structure of a Virtex-II CLB (and in general any other Xilinx FPGA), it is easily seen that it is specially suited to perform four input logical operations and latch the result in the Flip-Flop of the CLB. The hardware implementation will take full advantage of this feature to obtain high speed using few resources.

The hardware implementation has been done in two different ways. Fully unrolled loop with key generation and fully unrolled loop without key generation, each of these have been implemented for total reconfiguration or partial reconfiguration. In the reconfigurable version without the Key Schedule Generation hardware the keys are introduced in the encryption pipeline through reconfiguration.

2.1 Key Schedule Generation

The Key Schedule Generation is constructed using a shift register, XORs, a decoder, 33 128 bit registers and 1056 four input LUTs to implement the S-Boxes.

The shift register is 32 bit wide and 8 word depth. The initial key is stored in this register ($8 \cdot 32 = 256$) while the output is the exclusive or of elements 1, 3, 5, 8 and the fractional part of the golden ratio. Every clock edge, the words are shifted except for the 7^{th} element that is XORed with the index and shifted. This way, the functions have less than four inputs and are easly synthesised obtaining the w generating block (except for the shift).

The result of the prior operation is shifted and transported to every 128 bit register. Between the register and w the S-Boxes are generated. The final structure is a four input LUT and a Flip-Flop, matching the internal CLB structure. The initial permutation needed to obtain the final subkeys is performed through correct routing of the signals.

2.2 Encryption Pipeline

The encryption stage is fully pipelined. Every stage of the encryption process is divided into two substages with registers at the end and, as a result, two clock cycles are needed to complete each original round.

The first stage in the round includes the S-Boxes and the end register. The second stage includes the linear transformation and the XOR with the next round's key. This is a modification over the original algorithm, in our case the round algorithm is:

$$\hat{B}_0 = \mathrm{IP}(P) \oplus \hat{K}_0,\ \hat{B}_{i+1} = R_i(\hat{B}_i),\ C = \mathrm{FP}(\hat{B}_{32}) \tag{1}$$

$$\begin{aligned} R_i(X) &= L(\hat{S}_i(X)) \oplus \hat{K}_{i+1} \quad i = 0, \ldots, 30 \\ R_i(X) &= \hat{S}_i(X) \oplus \hat{K}_{32} \quad i = 31 \end{aligned} \tag{2}$$

With this modification we convert the three input linear transformation into a four input one. This modification improves both area and speed since the structure suits better Xilinx devices. The initial and final permutation are no other thing than Flip-Flop reordering using no resources.

The delays in this pipeline are smaller than in the Key Schedule Generation. To achieve a maximum throughput, the encryption pipeline is clocked at double speed than the Key Scheduler. This is done using a DCM (Digital Clock Manager)[11]. This module uses the system clock as input and can output several clocks derived from the original, among others, a double frequency clock can be obtained.

2.3 Reconfigurability

Reconfigurability can be used to improve the efficiency. In our case, the Key Scheduler is implemented on a PC. In the fully unrolled scheme a new linear transformation was generated. This linear transformation includes the keys. Doing so, the area needed is reduced since the Key Schedule Generator is omitted. This is specially interesting since the Key Schedule is very complex and area consuming.

Table 1. Summary of AES, NESSIE and DES Algorithm Hardware Implementations.

Authors	Algorithm	Device Used	Area	Through. (Mbps)	Efficiency (Mbps/slices)
Authors[1]	Serpent	XC2V2000	8,013 slices	42838	5.34
Authors[2]	Serpent	XC2V2000	5,143 slices	45714	8.89
McLoone *et al.* [12]	SHACAL-1	XC2V4000	13,729 slices	17021	1.24
McLoone *et al.* [12]	SHACAL-1	XCV1600E	14,768 slices	10123	0.68
Leong *et al.* [13]	IDEA	XCV1000	11,204 slices	2003	0.18
Standaert *et al.* [14]	Khazad	XCV1000	8,800 slices	9472	1.08
Standaert *et al.* [14]	MISTY1	XCV1000	6,322 slices	10176	1.6
Ichikawa *et al.* [15]	Camellia	XCV1000E	9,692 slices	6750	0.7
Beuchat [16]	RC6	XCV1600E	14,110 slices	9700	0.7
Rouvroy *et al.* [17]	Triple DES	XC2V3000	8,091 slices	13300	1.64

One of the main disadvantage seen by the NIST in the AES contest about serpent was the fact that the hardware needed for encryption/decryption almost doubled that needed for encryption. In other words, encryption and decryption could not share hardware. This obstacle can be overcome through reconfiguration. Encryption and decryption share the same hardware structure, and using

[1] With key generation.

[2] Without key generation.

reconfiguration, the same circuit can be used for encryption and decryption. In the total reconfigurable scheme no special care must be taken.

3 Performance Evaluation

In table 1 we can find several implementations and results of AES and NESSIE candidates as well as a Triple DES implementation.

The table shows that the architecture developed in this paper is faster and more efficient than any other implementation studied by the authors. Apart from the throughput, the area needed to implement the normal design is one of the smallest while the reconfigurable one is the smallest one (leaving apart systems that use Block RAMs and multipliers).

As mentioned in section 2.3, the circuit without Key Generation and total reconfiguration scheme leads to a faster design than the design with Key Generation. This is due to the fact that the decrease in the area makes the place and route process more efficient. The partial reconfiguration scheme has a slightly worse performance due to the area restrictions. These restrictions makes the placement more inefficient, leading slower designs.

The encryption/decryption schemes are treated only with reconfiguration schemes. If we wanted decryption circuits without reconfiguration, the performance and area will be almost the same as the encryption circuits. As it can be seen, the total reconfiguration encryption/decryption and the total reconfiguration encryption share the same performance and area. This is so because both encryption circuit are the same and the decryption one only differs in the order, routing and minor logic changes. In this case, the circuit runs at 333Mhz.

The problem when using reconfiguration is the time needed to reconfigure the system. Using the SelectMAP interface at a throughput of 400 Mbit/s, very fast reconfiguration can be achieved. The bitstream size of the encryption algorithm with key generation needs 1959234 bytes, the difference with the decryption is 652008 bytes leading to 13.0 ms of reconfiguration time.

4 Conclusions

The present paper presents two different architectures for the Serpent encryption algorithm. As it can be seen in table 1, the proposed architectures of the algorithm surpass, by large, both throughput and efficiency of any other implementation of AES and NESSIE candidates. In fact, the Serpent hardware description presented in this paper has an area similar to Triple DES while speed and security are boosted.

Although Serpent is not an standard encryption algorithm, this paper shows that it has outstanding properties in hardware implementations, making it an excellent candidate for very high speed communications encryption. It also shows that using reconfiguration, area very efficient versions can be produced as well as minimize the encryption/decryption area. This approach lightens the heaviest burden of Serpent seen by NIST, that the encryption and decryption doubles the resources needed for encryption.

References

1. US NIST: (Advanced Encryption Standard) `http://csrc.nist.gov/encryption/aes`.
2. IST-1999-12324: (New European Schemes for Signatures, Integrity, and Encryption) `www.cryptonessie.org/`.
3. Matsui, M.: New Block Encryption Algorithm MISTY. Lecture Notes in Computer Science (1997) 54–68
4. Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., Tokita, T.: Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms - Design and Analysis. In: Selected Areas in Cryptography. (2000) 39–56
5. Handschuh, H., Naccache, D.: SHACAL. 1^{st} NESSIE Workshop (2000) `https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/`.
6. (Burwick, C., Coppersmith, D., D'Avignon, E., Gennaro, R., Halevi, S., Jutla, C., Jr., S.M.M., O'Connor, L., Peyravian, M., Safford, D., Zunicof, N.)
7. Rivest, R., Robshaw, M., Sidney, R., Yin, Y.L.: The RC6^{TM} Block Cipher (1998) `http://www.rsasecurity.com/rsalabs/rc6/`.
8. Daemen, J., Rijmen, V.: (Aes proposal: Rijndael) `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.
9. Anderson, R., Biham, E., Knudsen, L.: (Serpent: A Proposal for the Advanced Encryption Standard) `http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/serpent.pdf`.
10. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: (Twofish: A 128-Bit Block Cipher) `http://www.schneier.com/paper-twofish-paper.pdf`.
11. George, M.: The Virtex-II DCM Digital Clock Manager. Xcell Journal Online **7/1/01** (2001) `http://www.xilinx.com/publications/products/v2/v2dcm.htm`.
12. McLoone, M., McCanny, J.: Very High Speed 17 Gbps SHACAL Encryption Architecture. Lecture Notes in Computer Science **2778** (2003) 111–120
13. Leong, M., Cheung, O., Tsoi, K., Leong, P.: A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA. In: IEEE Symposium on FCCMs. (2000)
14. Standaert, F., Rouvroy, G.: Efficient FPGA Implementation of Block Ciphers Khazad and MISTY1. 3^{rd} NESSIE Workshop (2002) `http://www.di.ens.fr/~wwwgrecc/NESSIE3/`.
15. Ichikawa, T., Sorimachi, T., Kasuya, T., Matsui, M.: On the criteria of hardware evaluation of block ciphers(1). Technical report, IEICE (2001) ISEC2001-53.
16. Beuchat, J.: High Throughput Implementations of the RC6 Block Cipher Using Virtex-E and Virtex-II Devices. Technical report, INRIA Research Report (2002) `http://www.ens-lyon.fr/~jlbeucha/publications.html`.
17. Rouvroy, G., Standaert, F.X., Quisquater, J.J., Legat, J.D.: Efficient Uses of FPGAs for Implementations of DES and Its Experimental Linear Cryptanalysis. IEEE Transactions on Computers **52** (2003) 473–482

Implementation of Elliptic Curve Cryptosystems over GF(2^n) in Optimal Normal Basis on a Reconfigurable Computer

Sashisu Bajracharya[1], Chang Shu[1], Kris Gaj[1], and Tarek El-Ghazawi[2]

[1] ECE Department, George Mason University,
4400 University Drive, Fairfax, VA 22030, U.S.A.
{sbajrach, cshu, kgaj}@gmu.edu

[2] ECE Department, The George Washington University,
801 22nd Street NW, Washington, D.C., U.S.A.
tarek@gwu.edu

Abstract. Reconfigurable Computers are general-purpose high-end computers based on a hybrid architecture and close system-level integration of traditional microprocessors and Field Programmable Gate Arrays (FPGAs). In this paper, we present an application of reconfigurable computers to developing a low-latency implementation of Elliptic Curve Cryptosystems, an emerging class of public key cryptosystems used in secure Internet protocols, such as IPSec. An issue of partitioning the description between C and VHDL, and the associated trade-offs are studied in detail. End-to-end speed-ups in the range of 895 to 1300 compared to the pure microprocessor execution time are demonstrated.

1 Introduction

Reconfigurable Computers are high-end computers based on the close system-level integration of traditional microprocessors and Field Programmable Gate Arrays (FPGAs). Cryptography, and in particular public key cryptography, is particularly well suited for implementation on reconfigurable computers because of the need for computationally intensive arithmetic operations with unconventionally long operands sizes.

As a platform for our experiments we have chosen one of the first general-purpose, stand-alone reconfigurable computers available on the market, the SRC-6E [1], shown in Fig. 1. The microprocessor subsystem of SRC-6E is based on commodity PC boards. The reconfigurable subsystem, referred to as MAP, is based on three Xilinx Virtex II FPGAs, XC2V6000.

As shown in Fig. 2, each function executed on the SRC-6E reconfigurable computer can be implemented using three different approaches: 1) as a High Level Language (HLL) function running on a traditional microprocessor, 2) as an HLL function running

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1001–1005, 2004.

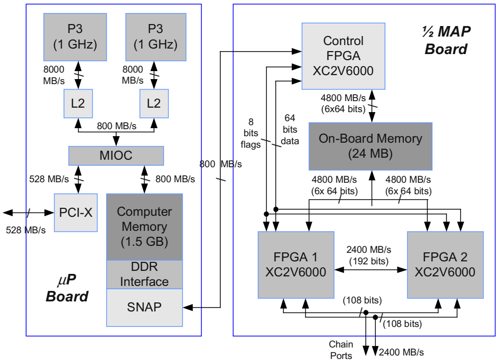


Fig. 1. Hardware architecture of the SRC-6E

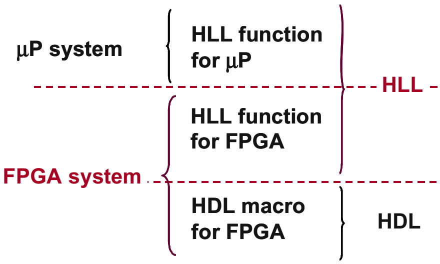


Fig. 2. Three ways of implementing a function on a reconfigurable computer

on an FPGA, and 3) as a Hardware Description Language (HDL) macro running on an FPGA.

As a result, any program developed for execution on the SRC-6E needs to be partitioned taking into account two independent boundaries, the first, between the execution on a microprocessor vs. execution on an FPGA system; and the second between the program entry in HLL vs. program entry in HDL.

2 Elliptic Curve Cryptosystems

Elliptic Curve Cryptosystems (ECCs) are a family of public key cryptosystems. The primary application of ECCs is secure key agreement and digital signature generation and verification [2]. In both of these applications the primary optimization criterion, from the implementation point of view, is the minimum latency for a single set of data (rather then the data throughput for a large set of data). The primary operation of ECCs is elliptic curve scalar multiplication (kP). In our implementation of scalar multiplication we adopted the optimized algorithm for computing scalar multiplication by Lopez and Dahab [3]. Our implementation supports elliptic curve operations over GF(2^n) with optimal basis representation for n=233, which is one of the sizes recommended by NIST [2].

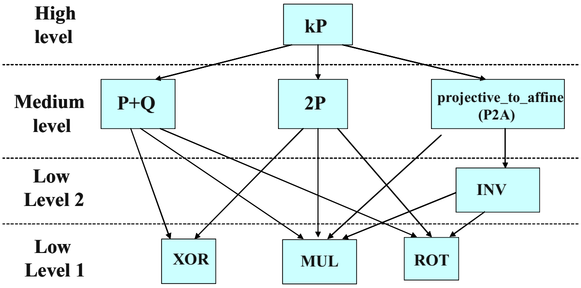


Fig. 3. Hierarchy of the ECC operations

3 Investigated Partitioning Schemes

A hierarchy of operations involved in an elliptic curve scalar multiplication for the case of an elliptic curve over GF(2^n) is given in Fig. 3. Four levels of operations are involved in this hierarchy: scalar multiplication (kP) at the high level (H), point addition (P+Q), point doubling (2P), and projective-to-affine conversion (P2A) at the medium level (M), inversion (INV) at the low level 2 (L2), and the GF(2^n) multiplication (MUL), squaring (rotation) (ROT), and addition (XOR) at the lowest level (L1). Functions belonging to each of these four hierarchy levels (high, medium, low 2 and low 1) can be implemented using three different implementation approaches shown in Fig. 1. In this paper, each of these approaches is characterized by a three-letter codename, such as 0HM. The meaning of these codenames is explained in Fig. 4.

We used as our reference case the complete ECC implementation running in the microprocessor and based on [4]. All other implementations ran entirely on the FPGA and were partitioned between C code that was automatically translated to VHDL and hand-coded VHDL.

4 Results

The results of the timing measurements for all investigated partitioning schemes are summarized in Table 1. The FPGA Computation Time, T_{FPGA}, includes only the time spent performing computations using User FPGAs. The End-to-End time, T_{E2E}, includes the FPGA Computation time and all overheads associated with the data and control transfers between the microprocessor board and the FPGA board.

The Total Overhead, T_{OVH}, is the difference between the End-to-End time and the FPGA Computation Time. Two specific components of the Total Overhead listed in Table 1 are DMA Data In Time, T_{DMA-IN}, and DMA Data Out Time, $T_{DMA-OUT}$. They represent, respectively, the time spent to transfer inputs from the Microprocessor Memory to the On-board Memory, and the time spent to transfer outputs from the On-Board Memory to the Microprocessor Memory.

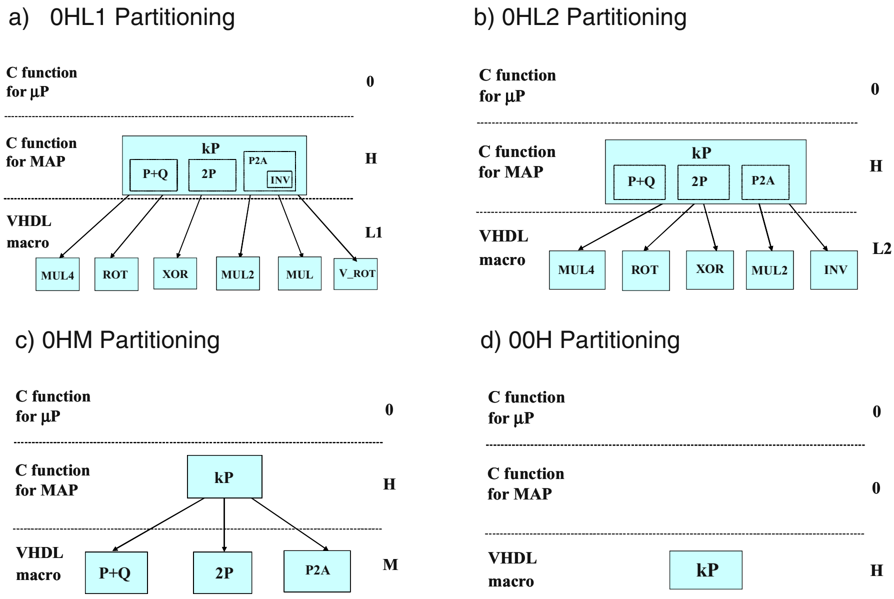


Fig. 4. Four alternative program partitioning schemes

The scheme that requires the smallest amount of hardware expertise and effort, 0HL1, is 893 times faster than software and less than 50% slower than pure VHDL macro. Implementing inversion in VHDL, in the 0HL2 scheme, does not give any significant gain in performance and only small reduction in the resource usage.

The 0HM scheme is more difficult to implement than 0HL1 and 0HL2 schemes, because of the additional operations that need to be expressed in VHDL Nevertheless, using this scheme gives substantial advantages in terms of both performance (about 45% improvement) and resource usage (e.g., reduction in the number of CLB slices by 24% compared to the 0HL1 scheme). The most difficult to implement, the 00H scheme (the entire kP operation described in VHDL) appears to have the same speed as 0HM, but it provides an additional substantial reduction in terms of the amount of required FPGA resources.

Table 1. Results of the timing measurements for several investigated partitioning schemes. Notation: SP_{SW} – speed-up vs. software, SL_{VHDL} – slow-down vs. VHDL macro.

	T_{E2E} (µs)	T_{DMA-IN} (µs)	T_{FPGA} (µs)	$T_{DMA-OUT}$ (µs)	T_{OVH} (µs)	SP_{SW}	SL_{VHDL}
Software	772,519	N/A	N/A	N/A	N/A	**1**	**1,305**
0HL1	866	37	472	14	394	**893**	**1.46**
0HL2	863	37	469	14	394	**895**	**1.45**
0HM	592	37	201	12	391	**1305**	**1.00**
00H	592	39	201	17	391	**1305**	**1.00**

Table 2. Resource utilization for several investigated partitioning schemes

	% of CLB slices	CLB slices vs. 00H	% of LUTs	LUTs vs. 00H	% of FFs	FFs vs. 00H
0HL1	99	**1.68**	57	**1.30**	68	**2.61**
0HL2	92	**1.56**	52	**1.18**	62	**2.38**
0HM	75	**1.27**	48	**1.09**	39	**1.50**
00H	59	**1.00**	44	**1.00**	26	**1.00**

The current version of the MAP compiler (SRC-6E Carte 1.4.1) optimizes performance over resource utilization. As it matures the compiler should be expected to balance high performance, ease of coding, and resource utilization to yield a truly optimized logic.

5 Conclusions

While earlier publications (e.g., [5]) regarding implementations of cryptography on reconfigurable computers have already proven the capability of accomplishing a 1000x speed-up compared to the microprocessor implementations in terms of the data throughput, this is a first publication that shows a comparable speed-up for data latency.

This speed-up is even more remarkable taking into account that the selected operation has only limited amount of intrinsic parallelism, and cannot be easily sped up by multiple instantiations of the same computational unit. In spite of these constraints, a speed-up in the range of 895-1300 has been demonstrated compared to the public domain microprocessor implementation using four different algorithm partitioning approaches.

References

1. SRC Inc. Web Page, http://www.srccomp.com/
2. FIPS 186-2, Digital Signature Standard (DSS), pp. 34-39, 2000, Jan. 27, http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf
3. López, J., and Dahab, R.: Fast Multiplication on Elliptic Curves over GF(2^m) without precomputation. CHES'99, LNCS 1717, (1999)
4. Rosing, M., Implementing Elliptic Curve Cryptography, Manning, 1999
5. Fidanci O. D., Poznanovic D., Gaj K., El-Ghazawi K., and Alexandridis N., "Performance and Overhead in a Hybrid Reconfigurable Computer," Reconfigurable Architecture Workshop, RAW 2003, Nice, France, Apr. 2003

Wavelet-Based Image Compression on the Reconfigurable Computer ACE-V

Hagen Gädke and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 23, D-38106 Braunschweig, Germany
koch@eis.cs.tu-bs.de

Abstract. Wavelet-based image compression has been suggested previously as a means to evaluate and compare both traditional and reconfigurable computers in terms of performance and resource requirements. We present a reconfigurable implementation of such an application that not only achieves a performance comparable to that of recent CPUs, but does so at a fraction of their power consumption.

1 Introduction

Accurately evaluating computer systems, both of the traditional and the reconfigurable kind, is not trivial. Too many characteristics can be measured in too many metrics. Implementations of benchmark applications are sometimes only subtly different, but no longer comparable (e.g., due to different quality of results).

To alleviate this, the Honeywell Benchmark Suite [1] uses so-called stressmarks to evaluate a broad spectrum of system characteristics. Each individual stressmark was developed specifically to test a subset of the interesting properties. All stressmarks are described by usage documents and sample implementations in C and sometimes also in VHDL. Minimum requirements on the quality of results support the comparability of measurements.

This paper examines an implementation of the versatility stressmark of the Honeywell suite. Since the VHDL reference code included in the Honeywell suite is not complete, the results described here can serve as baseline data for comparison with future realizations.

2 Versatility Stressmark

This application from the Honeywell suite [2] aims to evaluate how well the target hardware can perform several different functions using a single architecture. The stressmark is a wavelet-based compression algorithm [3] for square 8b gray scale images. Figure 1 shows the processing flow of the application.

While Honeywell includes a sample C implementation, the stressmark allows deviations. E.g., both true run-length encoding or zero length encoding may be used. For the entropy coding step, acceptable algorithms would include Huffman, Shannon-Fano and arithmetic coding.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1006–1010, 2004.

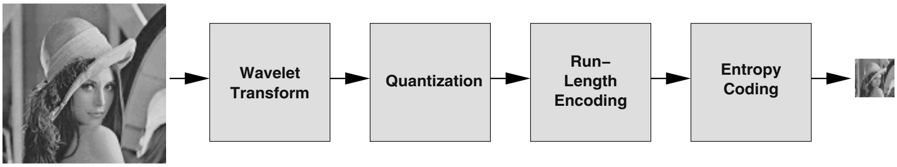


Fig. 1. Versatility structure

The "versatility" aspect of the stressmark considers the different implementation options for the algorithm. The choice we made is a so-called *high-complexity* implementation that executes all steps in hardware using a single configuration. Another angle not considered in the original stressmark definition is the different nature of the various stages: The wavelet transform and the quantization stage perform mostly arithmetic on multi-bit words, while the entropy coding step primarily performs bit-manipulations. With this mix, the capability of the target platform to handle these mixed computation styles can be evaluated.

The quality of results requirements for this stressmark are defined as minimum peak signal-to-noise ratios (specified in dB) at a maximum compressed bit rate (given in bits per pixel).

3 Stressmark Realization and Optimization

As in the sample C implementation, the wavelet transform itself is computed as a 3-step (2,2)-biorthogonal Cohen-Daubechies-Fouveau transform implemented in the Lifting Scheme [4].

The first horizontal filtering pass processes 4 8b pixels simultaneously per clock, subsequent horizontal passes operate on 2 16b values simultaneously per clock (the width of the data expands during processing). All vertical passes process a single 16b value per clock. As in the sample C code, the three highest frequency blocks are assumed to contain only irrelevant (non-visible) details and are dropped entirely from further processing.

The quantization step can be started only *after* the minimum and maximum coefficient values of a block are known. Thus, a block can be quantized only after it has been completely wavelet-transformed. However, the quantization and both the following run-length and entropy encoding steps can be performed in a pipeline-parallel fashion.

While we have adhered to the C sample implementation of the stressmark to a large degree, some obvious inefficiencies were corrected beforehand (both in SW and HW):

- The memory requirements were reduced by sharing the memory for two buffers across all processing phases instead allocating dedicated areas.
- The Wavelet Transform fcdf22 does not explicitly copy the input data into a local buffer before the computation. Instead, the local copy is built and maintained on-the-fly during the calculation.
- The high-frequency coefficient blocks 7, 8, and 9 are discarded as early as possible and are neither stored nor processed further.

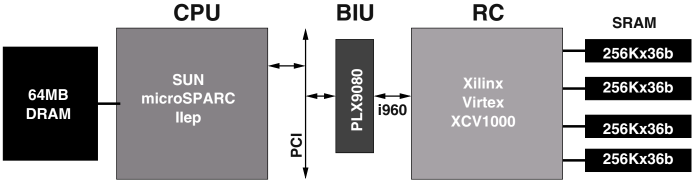


Fig. 2. ACE-V architecture

Table 1. Resource requirements

Module	Slices	BlockRAMs
Wavelet	1075	
Quantization	1251	
Zero-Length Encoding	179	
Huffman Coding	258	3
Global Control & Muxing	1507	
Memory Streaming Engine (MARC) [6]	1971	6
Total	6241	9
Equivalent ASIC	270K Gates	

- Row-order traversal of memory is faster than column-order accesses, since our 32b system data bus can fetch four adjacent 8b pixels or two adjacent 16b wavelet coefficients in a single cycle.

4 Experimental Results

Our implementation of the application for the ACE-V platform (Figure 2, [5]) was formulated in RTL Verilog, synthesized using Synplicity Synplify 7.2.2 and mapped to the Virtex target using Xilinx ISE 5.2.03[1]. Table 1 lists the area requirements of the complete versatility stressmark.

Table 2 gives an overview over the execution times of the stressmark when compressing 256x256 and 512x512 pixel Lena images on various platforms. In all cases, computation and in-memory data transfer operations were timed, but disk I/O was always omitted. Furthermore, for all reconfigurable platforms, configuration times were not included since our implementation does not need to reconfigure between processing phases. The last two lines in the table shows the clock frequencies achievable when targeting our design to recent 90nm commodity (Xilinx Spartan 3 series) and high-performance reconfigurable devices (Xilinx Virtex IIpro series). The hypothetical execution time on these RCUs is estimated by a simple scaling based on the increased clock frequency.

Even more interesting than the absolute performance data is the power consumption of the different processing units for the same task. For the traditional CPUs, the values are quoted from their data sheets. For the ACE-V Virtex 1000 RCU, the number shown is the peak power consumption as determined using the Xilinx XPWR power estimation program on a complete post-layout simulation trace (based on more than 30GB of data).

[1] Later versions of the tools have reproducible errors that lead to non-functional circuits.

Table 2. Performance data

Processor	Clock [MHz]	Execution Time [ms]		Power [W]
		256x256	512x512	
ACE-V RCU	30	6.6	17.5	1.1
Sun UltraSPARC III+	900	6.7	24	52.0
AMD Athlon	1333	6.0	131	63.0 … 70.0
AMD Athlon XP	1666	3.8	91	54.7 … 60.3
Xilinx XC3S1000-4 FPGA	*63*	*3.1*	*8.3*	*-?-*
Xilinx XC2VP20-7 FPGA	*105*	*1.9*	*5.0*	*-?-*

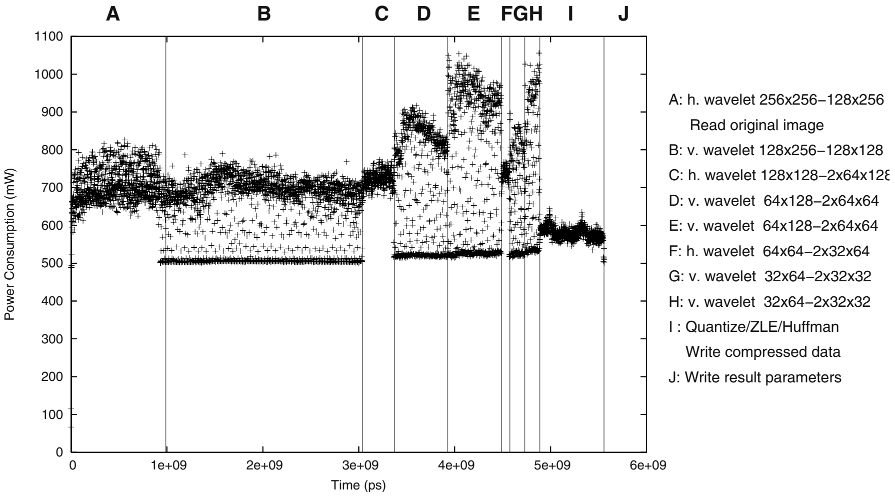


Fig. 3. Power consumption profile for Virtex-based RCU

Figure 3 shows the power consumption profile of the RCU with 1μs resolution over the entire simulated execution. Note the regular drops during the vertical processing phases, occurring when the end of a column has been reached and the read stream has to be reprogrammed and restarted. At those times, the wavelet transform units remain idle.

Table 3 shows the quality-of-results requirements from the original Versatility C implementation and the actual values achieved by our hardware when compressing images with $L = 256^2$ at the stressmark's default quality value of $q = 128$. Our hardware implementation at least matches the software values. For the Goldhill image, an error in the reference software (loss of up to 7b after Huffman coding) that was corrected in the hardware version even improves the hardware-achieved quality.

[2] Since the ACE-V has reliability issues performing memory transfers > 64KB, no QoR data could be obtained on the actual HW for $L = 512$.

Table 3. Quality of results for $L = 256$ at default compression quality

Image	b [bpp]	PSNR [dB]	
		Original	Hardware
Barbara	0.29	26.8	26.8
Goldhill	0.26	27.1	27.6
Lena	0.27	27.6	27.6

5 Discussion and Conclusion

We have implemented the Versatility stressmark of the Honeywell suite on a Virtex-based adaptive computer system (ACS) and evaluated it in terms of resource requirements, performance and power consumption. While the ACE-V ACS with its slow 1998-vintage RCU (250nm process) is no longer competitive with more recent CISC CPUs, current reconfigurable devices will allow the realization of RCUs that reach or exceed CPU performance again. Even more promising is the low power consumption of the reconfigurable solutions. With the move to 90nm devices, higher power savings seem quite achievable.

References

1. Kumar, S., Pires, L., Ponnuswamy S., et al., “A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions”, *Proc. Eighth International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey (USA), 2000
2. Honeywell Technology Center, “Versatility Stressmark”, *Benchmark Specification Document Version 0.8 CDRL A001*, Minneapolis (USA), 1997
3. Antonini, M., Barlaud, M., Mathieu, P., Daubechies, I.,“Image Coding using the Wavelet Transform”, *IEEE Transactions on Image Processing*, No. 1, 1992
4. Uytterhoeven, G., Roose, D., Bultheel, A., “Wavelet Transforms using the Lifting Scheme”, *Technical Report ITA-Wavelets Report WP 1.1, Katholieke Universiteit Leuven, Department of Computer Science*, Belgium, 1997
5. Koch, A., Golze, U., “A Comprehensive Prototyping Platform for Hardware-Software Codesign”, *Proc. Workshop on Rapid Systems Prototyping*, Paris (F), 2000
6. Lange, H., Koch, A. “Memory Access Schemes for Configurable Processors”, *Proc. Intl. Workshop on Field-Programmable Logic and Applications (FPL)*, Villach (AT), 2000

A Reconfigurable Communication Processor Compatible with Different Industrial Fieldbuses

Mª Dolores Valdés, Miguel Ángel Domínguez, Mª José Moure, and Camilo Quintáns

Departamento de Tecnología Electrónica, University of Vigo
36200 Vigo, Spain
{mvaldes, mdgomez, mjmoure, quintans}@uvigo.es

Abstract. Fieldbuses constitute the lower level of communication networks in a flexible manufacturing system. Nowadays there are a lot of proprietary protocols, thus the interconnection of equipment from different manufacturers has become very problematic. Changing equipment supposes the change of the fieldbus too with the consequent economic losses. This paper proposes the implementation of a communication processor based on a reconfigurable device that makes different fieldbuses compatible. In this way, using the appropriate level and impedance matching circuit, the same hardware can be connected to different fieldbuses. In order to verify this proposal a communication processor based on an FPGA supporting two very important fieldbus standards in the area of industrial control such as WorldFIP and Profibus, has been developed. Design constraints and performance of the implemented processor are analyzed in this paper.

1 Introduction

Fieldbuses are digital serial, multidrop data buses used to connect primary elements of automation (sensors and actuators with control and measurement functions) with control devices of a higher level (PLCs, numeric control machines, processors, etc.). Nevertheless, there are a lot of different commercial solutions and protocols, making difficult the compatibility between equipment of different manufacturers. The lack of standardization is a problem in the area of industrial communication networks.

In this paper the authors propose the implementation of fieldbus communication processors using reconfigurable circuits. In this way a communication protocol can be modified without changes in the hardware support that implies an economic saving. In this line the authors have implemented a communication processor based on an FPGA supporting two important fieldbus standards in the area of industrial control such as WorldFIP and Profibus. The chosen FPGA for this project was the FLEX 10K20RC240-4 from the ALTERA company [1].

2 Profibus and WorldFIP

Profibus [2] is a fieldbus protocol proposed by German manufacturers while WorldFIP [3] was driven by French manufacturers. The both have been included by

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1011–1016, 2004.

the CENELEC in the standard EN 50170 (Volume 2 and 3 respectively). The most significant differences between them are:

- *Data rates:* WorldFIP (until 5 Mbps) achieves higher data rates than Profibus (until 500 Kbps).
- *Types of stations:* A Profibus network can have several master stations and also slave stations. WorldFIP networks have a single active bus arbitrator and the rest of nodes have consumer/producer functionality.
- *Timers:* Profibus has a lot of timers making the timing information management and the implementation of timers more complicated than in WorldFIP.
- *Signal level and coding:* Profibus uses a RS-485 communication in the physical level and WorldFIP uses a Manchester coding.
- *Frame format:* In the data link level the protocols are completely different and also the frame format is not the same.

It can be observed that although Profibus and WorldFIP have a same philosophy, the differences between both protocols are important and then it forces to implement different software and hardware for each communication interface. Thus, all this features must be borne in mind in the implementation of a compatible device for these protocols using an FPGA that is proposed and reported in this paper.

3 Design of a Fieldbus Interface Using an FPGA

The implementation of fieldbus communication processors using SRAM FPGAs has a lot of advantages derived from its reconfiguration capability. The most important are:

- The FPGA programming code can be easily improved adding new functions.
- Little changes in bus definitions (frame format, transmission headers, etc.) can be easily solved changing the software description of the interface, avoiding the economic cost associated to the equipment renewal.
- It is possible to design devices for different fieldbuses with similar physical layers making little changes in the programming file.

Taking into account these advantages, the development of reconfigurable fieldbus interfaces results in a very interesting project. Thus, the authors have worked in the implementation of the basic communication functions for Profibus and WorldFIP protocols using a SRAM FPGA. A Profibus slave station and the basic services (writing buffer, reading buffer and transfer buffer) of a WorldFIP consumer/producer station have been implemented over the same hardware (FPGA). This compatible device is able to transmit and receive any frame in right way over any WorldFIP or Profibus network.

There are several factors to be considered in the selection of the FPGA for this application. The most important are the cost, the operation frequency and the available logic resources. The fieldbuses operate at low data rates (in this work 9.6 Kbps for Profibus and 1 Mbps for WorldFIP) and then a high speed FPGA is not required. By other side, fieldbus protocols manage many variables that must be stored. So, in order to simplify the design, it is suitable the use of an FPGA with embedded memory blocks. Taking into account these two aspects and the cost, the

FPGA FLEX 10K20RC240-4 from the ALTERA manufacturer was chosen for implementing the communication processor of this project [4].

The design was developed using the Quartus II CAD tools from ALTERA and the system was specified in VHDL (Very high speed IC Hardware Description Language) to allow the portability of the design to other FPGA devices.

4 Implemented Device

4.1 The Fieldbus Communication Processor

To get a compatible device between WorldFIP and Profibus networks two designs were specified using VHDL. The respective block diagrams are shown in Figure 1.

In the case of WorldFIP the transmitter sends data adding the Frame Starting Sequence (FES) and the Frame End Sequence (FES) established in the WorldFIP protocol. The receiver detects the frames and delivers them to the receiving buffer checking the corresponding Frame Check Sequence (FCS). The receiving buffer is used to temporarily store the last frame received that will be processed by the WorldFIP control system. To implement this circuit a cycle shared double port RAM is used to allow a simultaneous access to this buffer. The WorldFIP control system processes the frames stored in the receiving buffer and runs the required actions according to the protocol. The values of the consumed and produced variables (set in the configuration of the device) are stored in the consumed and produced databases, also implemented with a cycle shared double port RAM. The identifier index contains all the information about the identifiers of the consumed and produced variables. Finally, there is an interface between the external circuits (the corresponding sensors and actuators) and the designed communication processor that is implemented in the variable access protocol block.

A WorldFIP network with a data rate of 1 Mbps was used for this work. The bus uses Manchester coding. Thus, the system requires two clocks, the main one of 10 MHz used by the receiver (take 5 samples per half symbol) and another of 2 MHz used to control the data sending. Both signals are assigned to dedicated clock lines.

In the case of Profibus the receiver circuit manages the reception of the characters conforming the frames transmitted by the bus. When a byte is received, this circuit indicates to the reception application module if it is valid or not. The reception application module takes the valid data bytes and analyzes them to verify the destination address and the frame type. It checks for possible errors in transmission (uses a check field of the frame) and if there are not errors the transmission application receives the information of the received frame. Then, the transmission application module generates the corresponding response frame that is sent to the transmitter circuit so that it conforms and transmits the frame characters. This design includes a sensor buffer to store the last measurement from a specific sensor. This value is read by the transmission application module to generate the adequate response frame when a request frame asking for this data is received in the device.

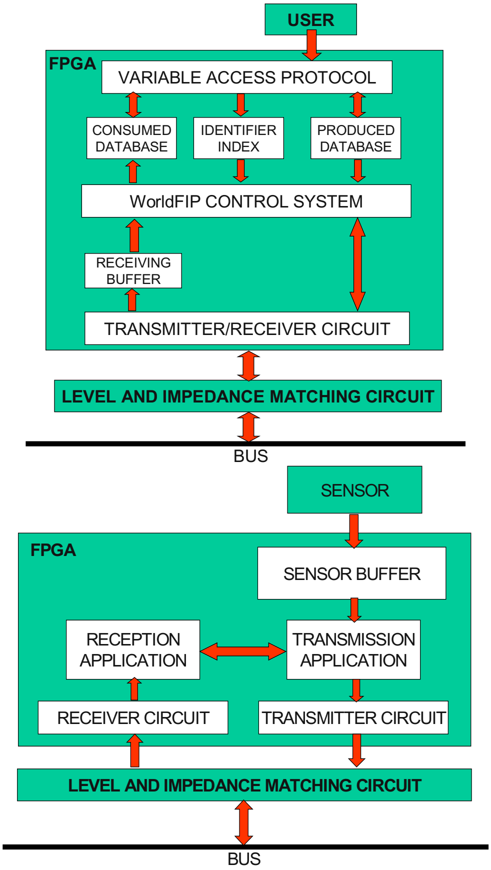


Fig. 1. Schemes of the implemented WorldFIP (a) and Profibus (b) interfaces

The data rate of the Profibus network used in this work is 9.6 Kbps, limited by a protocol analyzer station that uses the serial port to capture the frames circulating by the bus. Then, the clock signal frequency used by the receiver and transmitter circuits is 9.6 KHz too. It is resynchronized with the bus every time a new character is detected.

4.2 Technical Comparison

Table 1 shows a comparative of the FPGA logic resources required to implement the WorldFIP and the Profibus communication processors.

It can be observed that the average of the logic used resources is very similar in both cases. The WorldFIP interface uses more memory blocks and interconnections because it needs to store more variables (several consumed and produced data, identifier index, variables of management to indicate the presence of a station in the bus, etc.) than the Profibus interface (only the data stored in the sensor buffer).

Considering that in both cases the resources usage is very high this FPGA is adequate for the implementation of simple communication interfaces (a Profibus slave station and a WorldFIP consumer/producer station with the basic services). When more functions must be implemented (for example, the design of a Profibus master station, a WorldFIP consumer/producer station with all the services provided by the data link layer or a WorldFIP bus arbitrator station), a bigger FPGA is required.

4.3 Level and Impedance Matching Circuits

Due to the electrical characteristics of Profibus, WorldFIP and FPGA are not the same, it is necessary to design different level and impedance matching circuits that convert the signals present in the bus to TTL/CMOS signals suitable for the FPGA and vice versa. This is the unique change in the hardware required for connecting the designed compatible device based on FPGA to these two fieldbuses.

The WorldFIP matching circuit is based on an integrated circuit from Telemecanique known as CREOL. In the case of Profibus a MAX485CPA integrated circuit is used to convert TTL/CMOS signals into Profibus signals (RS-485).

Table 1. FPGA resources used in this project

WORLDFIP			PROFIBUS		
Resources of FPGA			Resources of FPGA		
Interconnection Resources	Logic Resources		Interconnection Resources	Logic Resources	
	Logic Cells	Memory Blocks		Logic Cells	Memory Blocks
74%	94%	60%	56%	94%	33%
	82%			81%	

5 Conclusions

In this paper the authors propose a solution to design compatible devices for different fieldbuses using reconfigurable circuits. It is demonstrated that it is possible to get a device that can be connected to two of the more important fieldbuses used in Europe like WorldFIP and Profibus without changing the hardware (it is only necessary to change the connector that contains the adequate level and impedance matching circuit). It supposes a great money saving when a user needs to change the protocol of its fieldbus system and also an improvement in the interconnection of commercial fieldbus devices.

The FPGA used in this work is a low cost circuit with limited logic resources and the resources usage is very high (section 4.2). Thus, to implement more complex devices it is necessary to use an FPGA of higher level with more logic resources and interconnection elements and also more expensive.

In this line of research, it would be very interesting for future works to get the implementation of multiple fieldbus protocols in a unique FPGA. Also it would be possible to implement an intelligent reception module able to identify the type of network from the analysis of the received frames to automatically use the adequate protocol. Thus, a total compatibility of the fieldbus devices would be achieved with the advantages that it supposes.

References

1. Lías, G., Valdés, M.D., Domínguez, M.A. and Moure, M.J., 2000. Implementing a fieldbus interface using a FPGA. In FPL'2000 10th International Conference on Field Programmable Logic and Applications. Villach, Austria, August 2000, Lecture Notes in Computer Science 1896, R.W. Hartenstein, and H. Grünbacher (Eds.), ISBN 3-540-67899-9, Springer-Verlag, pp. 175-180.
2. PROFIBUS Nutzeorganisation, 1991 (April). Profibus standard: Translation of the German standard DIN 19245 (part 1 and part 2), PROFIBUS Nutzeorganisation e.V. Wesseling Germany.
3. AFNOR, 1990. Standard NFC 46-601 to 605: Bus FIP pour échange d'information entre transmetteur, actionneur et automate.
4. ALTERA, 1998. ALTERA Data Book, Altera Corporation, San José (CA).

Multithreading in a Hyper-programmable Platform for Networked Systems

Philip James-Roxby and Gordon Brebner

Xilinx Research Labs, Xilinx, Inc., U.S.A.
{phil.james-roxby, gordon.brebner}@xilinx.com

Abstract. Modern programmable logic devices have capabilities that are well suited for them to assume a central role in the implementation of networked systems. We have devised a highly flexible soft platform architecture abstracted from such physical devices, which may be viewed as a particularly configurable and programmable type of network processor. In this paper, we discuss multithreading in the context of this logic-centric soft platform, and describe the programmable mechanisms to support multithreading that we have implemented. Through a design example, we evaluate these mechanisms, and report that the solution obtained had comparable performance to a custom solution written from scratch without the intermediate soft platform.

1 Introduction

Networked systems feature communication and networking as a significant activity, alongside computational activity. Examples include simple sensors and actuators, and more complex clients and servers. In our broad research, we aim to establish that field programmable logic devices can offer an attractive basis for implementing networked systems, through the efficient delivery of "hyper-programmable architectures", where all aspects of the architecture are configurable, not just that the architecture supports a programming model. We have developed an experimental HYPMEP (Hyper-Programmable Message Processing) soft platform, which is currently targeted at Xilinx Virtex-II Pro platform FPGA devices.

Multithreading for *logic-centric systems* (such systems being alternative to conventional processor-centric systems) was introduced at FPL 2002 [1]. The main motivation was to transfer the benefits of multithreading for processors into programmable logic circuitry, thus providing an attractive higher-level method of organizing the fine-grained parallelism offered by programmable logic. The basic ideas were illustrated within a hand-crafted case study – the MIR packet router [2]

In this short paper, we describe the support for logic-centric multithreading that we have implemented in the HYPMEP soft platform, concentrating on practical issues. We first describe the general programmable mechanisms, and how they interact with support for non-thread based computation in a complete system. We then outline a re-implementation of the MIR packet router, programmed on our soft platform, that achieved performance comparable to the original bare-FPGA design.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1017–1021, 2004.

2 Threads in the HYPMEP Soft Platform

A thread is a computational component with a single identifiable control flow. In its logic-centric incarnation, a thread is a finite state machine with programmed structure and programmed operations associated with each state. This has an obvious direct implementation in programmable logic, but an alternative implementation could be via a conventional software thread executed on an embedded processor. Operationally, threads have top-level modes which minimally include *idle*, when no processing is performed, and *active*, when processing is performed through finite state machine execution. The minimal form of interaction between threads is allowing a thread to start or stop another thread, i.e., to move it between idle and active modes.

In the networked systems that we study, threads typically work on an incoming *message* (a generic term used to cover packets, frames, cells, etc.), performing processing in tandem on it until it leaves the system again. Much of the time, particular threads will be idle, waiting for relevant parts of messages to process. An important task in mapping a networked system onto the HYPMEP platform is determining what combination of threads would be best suited to implementing the required functionality and performance. An overall aim is to produce as simple a set as possible, with a clear flow of control interaction between the threads, and with a minimal amount of data exchanged between them. It is also desirable to maximize re-use of threads from earlier designs. From our initial experiments, we have devised various mapping heuristics, which we are continuing to refine in ongoing research.

2.1 Inter-thread Synchronization

We have implemented support for a basic inter-thread synchronization mechanism. It is possible for one thread to cause another thread to move from an idle mode to an active mode by sending it a *start signal*. This also causes the thread to begin execution in an initial state. Symmetrically, it is possible for one thread to cause another thread to move from an active mode back to an idle mode by sending it a *stop signal*. A stop action can optionally cause the thread to pass through a final state before ending execution. As well as these two necessary actions, we have also added support for *suspend* and *resume* signals, which move the thread between its active mode and a *suspended* mode, and back, respectively.

This is a fully programmable mechanism, allowing any thread to control any other thread. A thread sends a signal to some other thread by executing a *signal operation* during its regular execution. Specialization of the support for specific inter-thread signaling is then carried out after programming, when it becomes known which threads affect which other threads. At the top level of abstraction, a thread might be implemented either directly in logic, or on an embedded processor. In our current implementation, for threads implemented in logic, the incoming start and stop control signals are generated just by forming the logical-OR of the signals from every other thread. Then, the specialization is automatically performed by the back-end synthesis tool, which removes all redundant wires, thus creating a set of point-to-point signaling wires. For threads implemented on an embedded processor, a conventional, very lightweight multi-threading kernel can be used to provide inter-thread support.

In our HYPMEP soft platform, there is no single, central, system-wide control flow. Instead, there are per-message control flows, which follow from sequences of start and stop signals. The initial arrival of a message at a system interface causes a thread to start; this thread then conditionally starts other threads; these in turn start further threads, etc. Either through death by natural causes or through stop signals, the threads all finish their efforts on the message, the last thread to stop being the thread responsible for the departure (or perhaps destruction) of the message.

2.2 Inter-thread Data Communication

In our HYPMEP soft platform, the messages that are processed by threads constitute a *de facto* shared memory inter-thread data communication mechanism. One thread's changes to a message can be read by another thread. We naturally support this as an essential feature of the operation of the soft platform, and one of our major research themes concerns novel programmable memory architectures to support our message processing model. In general, however, some threads may wish to communicate data that is not strictly the content of a message being processed by the system. Rather than abuse the external message types as carriers for such data, we also allow internal messages within our system, which are exclusively used for inter-thread data communication. In terms of programming data communication, we currently have two strengths of internal messaging. The lightweight version involves single shared variables between threads, allowing limited communication through normal operations reading from and writing to memory. The heavier weight version involves channels between threads. A thread can send an internal message to some other thread by executing a *send operation* during its regular execution, and the recipient executes a matching *receive operation*.

For all three data communication schemes mentioned, there are issues surrounding synchronization between threads, as in all such parallel systems. In our case, given the nature of the systems being implemented, system behavior is much more predictable than for some general-purpose parallel computing system. Thus, there is a great deal more scope for static scheduling of thread behavior to avoid conflicts or deadlocks, as opposed to adding mechanisms for dynamic scheduling and arbitration of access to shared data.

2.3 Non-threaded Computation

The threading model is well-suited to many aspects of message processing, since the handling of communication protocols normally has the characteristic of following an enclosing finite state machine, with simple, fine-grain operations performed on a message in each stateHowever, there are also cases when rather more complex, coarse-grain operations, not naturally or efficiently expressed in (logic-centric) thread form, are required. It may also be the case that existing IP blocks or software is available for re-use, as a desirable alternative to implementing operations afresh, whether using threads or not. We have made provision for these cases as a fundamental provision in our HYPMEP soft platform, alongside support for threads.

3 Case Study: A Re-implementation of MIR

In 2001, Brebner developed the Multi-version IP Router (MIR) design targeted at the Xilinx Virtex-II Pro platform FPGA [2]. This was a four-port gigabit Ethernet router capable of handling both IPv4 and IPv6 traffic. The MIR design introduced the notion of a logic-centric system architecture targeted at a platform FPGA and, in particular, the notion of multithreading in logic to maximize concurrency and minimize latency. However, the actual implementation of MIR was coded in hand crafted Verilog HDL for the bare programmable logic, plus C language for the bare embedded processor, and so represented a one-off system design that required specialist skills. As a case study for our HYPMEP soft platform, and in particular for the programmable multithreading support, we re-implemented the MIR design over the soft platform. Our aim was to achieve a high-level programmed design, in contrast to the original low-level fixed design, but without major detriment to the MIR system performance. We retained a similar assignment of threads as was used in the original MIR design. Replication of thread clusters at each port maximized concurrency at the port level, to complement the thread-wise concurrency at each port. The only non-thread concurrent processing activity closely integrated with the threads is the lookup of IPv4 packet addresses. For this coarse-grain operation, we investigated various standard algorithms, implemented in logic or on the embedded processor. Our final implementation used a software decelerator implementing a hash table algorithm on the embedded PowerPC in Virtex-II Pro. We present a detailed analysis of its design and implementation in a recent paper [3].

3.1 Inter-thread Synchronization

The required inter-thread synchronization for the original MIR design was covered comprehensively in the earlier paper on multithreading [1]. We do not repeat the details here, since we use the same assignments of threads. The change here is that the start and stop signals sent between threads were programmed using the support for threads provided in our soft platform, as opposed to the original *ad hoc* direct implementation in the original version.

3.2 Inter-thread Data Communication

The amount of direct inter-thread data communication required is very modest. In fact, only a total of 72 bits need to be transferred between threads, and each case is a single source/single destination transfer. This communication was implicit in the original MIR design, being implemented using fixed-purpose registers. Here, the data exchange between the threads was programmed using the lighter weight mechanism for communicating data, as described in Section 2.2. Internally, this feature of the soft platform is mapped to a simple point-to-point connection network. When first received, packet data is communicated via a programmable broadcaster channel that also acts as a data-width converter. Each data word received by an interface is broadcast to all other threads, so that all threads see the same data at the same time.

3.3 Non-threaded Computation

As already mentioned, lookup of IPv4 addresses is performed on behalf of the logic by a software decelerator. In choosing such an implementation, our target was that the time for a lookup (including all of the logic-to-processor-to-logic interface latency) should be within 208 ns, to meet the zero internal latency goal when a minimum-length packet is being processed [3]. Aside from this, the embedded processor is also responsible for handling the less frequent types of packet processing. The final non-threaded computational components are the standard interface blocks used for physical packet input and output. The target block for the MIR system was the standard Xilinx Gigabit MAC (GMAC) IP core.

3.4 Experimental Results

We implemented our system on a Xilinx XC2VP7 Virtex-II Pro platform FPGA, which includes eight multi-gigabit transceivers, and used the standard Xilinx ISE 6.1 tools to produce the bitstream for the FPGA from a VHDL description of the programmed soft platform. Two aspects of performance are important. The first is whether the threads are fast enough to keep up with a gigabit line rate. The GMAC core produces 8-bit words at a rate of 125 MHz, and the broadcaster channel converts this data stream to 32-bit words at 31.25 MHz. Thus, all connected threads must be clocked at 31.25 MHz, which was easily achievable. The second aspect is whether the most frequent packet types can be handled with zero internal latency. The most common packet type is expected to be IPv4, and the latency-determining step in handling such packets is address lookup, here implemented using the embedded processor. We found that at least 98% of lookups could be performed within the desired time of 208 ns by our software decelerator, if the processor is clocked at 350 MHz, and its on-chip memory (OCM) bus is clocked at 175 MHz, both of which were attainable rates [3]. The design required 4140 Virtex-II Pro slices, which is 84% of the small XC2VP7 device used. The number of Block RAMs required for the design was 28 (63% of an XC2VP7 device), a reduction from the original MIR requirements due to use of the PowerPC caches here.

In summary, we were very pleased to find that our programmed re-implementation of MIR was quantitatively as good as the original hand crafted implementation. This case study, along with others, has encouraged our current research directions for the hyper-programmable HYPMEP soft platform for networked systems.

References

1. G. Brebner, "Multithreading for logic-centric systems", *Proc. 12th International Workshop on Field Programmable Logic and Applications*, Springer LNCS 2438, Sep 2002, pp.5-14.
2. G. Brebner, "Single-chip gigabit mixed-version IP router on Virtex-II Pro", *Proc. 10th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, Apr 2002, pp.35-44.
3. P. James-Roxby, G. Brebner and D. Bemmann, "Time-critical software deceleration in an FCCM", *Proc. 12th IEEE Symposium on FPGAs for Custom Computing Machines*, Apr 2004.

An Environment for Exploring Data-Driven Architectures*

Ricardo Ferreira[1], João M.P. Cardoso[2,3], and Horácio C. Neto[3,4]

[1] Departamento de Informática, Universidade Federal de Viçosa,
Viçosa 36570 000, Brazil, cacau@dpi.ufv.br
[2] Universidade do Algarve, Campus de Gambelas, 8000-117, Faro, Portugal
jmpc@acm.org
[3] INESC-ID, 1000-029, Lisboa, Portugal
[4] Instituto Superior Técnico, Lisboa, Portugal, hcn@inesc.pt

Abstract. A wide range of reconfigurable coarse-grain architectures has been proposed in recent years, for an extensive set of applications. These architectures vary widely in the interconnectivity, number, granularity and complexity of the processing elements (PEs). The performance of a specific application usually depends heavily on the adequacy of the PEs to the particular tasks involved, but tools to efficiently experiment architectural features are lacking. This work proposes an environment for exploration and simulation of coarse-grain reconfigurable data-driven architectures. The proposed environment takes advantage of Java and XML technologies to enable a very efficient backend for experiments with different architectural trade-offs, from the array connectivity and topology to the granularity and complexity of each PE. For a proof of concept, we show results on implementing different versions of a FIR filter on a hexagonal data-driven array.

1 Introduction

A large number of coarse-grained reconfigurable architectures have been presented in recent years [1], in order to overcome some of the limitations of the fine-grained structures used in common FPGAs. In fact, coarse-grained architectures are more suitable to implement typical DSP algorithms, require shorter design cycles, and are faster to reconfigure, etc. Coarse-grained architectures, behaving in a static dataflow fashion [2] (e.g., [3], [4]), are of special interest, as they naturally process data streams, and therefore provide a very promising solution for stream-based computations, which are becoming dominant. In addition, the control flow can be distributed and can easily handle data-streams even in presence of irregular latency times. Moreover, a regular design, like an array processor, can reduce the wire lengths, the verification and test costs, the power consumption, and can also improve the circuit reliability and scalability.

* This work is in part supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project. Ricardo Ferreira acknowledges the financial support from Ct-Energia/CNPq, Brazil.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1022–1026, 2004.

Since many design decisions must be taken in order to implement an efficient architecture for a given set of applications, environments to efficiently experiment with different architectural features are fundamental. As far as we know, no environment exists to explore a large extent of architecture trade-offs: topology, granularity, processing element (PE) characteristics, schemes of communication between PEs, etc. Recently, some works (e.g., [5], [6]) have addressed the exploration of a few parameters of specific coarse-grained architectures. Although supporting several features and having the potential to suggest some design decisions, the work in [7] furnishes an exploration environment limited to a family of KressArray architectures.

Our work aims to support a broad range of data-driven based arrays, a significant set of architecture parameters, and then evaluate its trade-offs by implementing representative benchmarks. We present an environment, named as EDA (Environment for exploring Dataflow Architectures). The environment can help the designer to systematically investigate different data-driven array architectures, as well as internal PE parameters (existence of FIFOs in PE input/outputs and their size, number of input/outputs of each PE, pipeline stages in each PE, etc.), and to conduct experiments to evaluate a number of characteristics (e.g., protocol overhead).

2 Environment

A global view of EDA is shown in Fig. 1. The environment uses Java and the XML technology [8] to enable exploration. We use XML since it facilitates the creation of the languages needed in the environment: to specify the dataflow representation (see Fig. 2), to specify the array architecture, to specify the placement and routing, etc. The XSLT transformation language [11], makes easy the generation of descriptions into the appropriate output format.

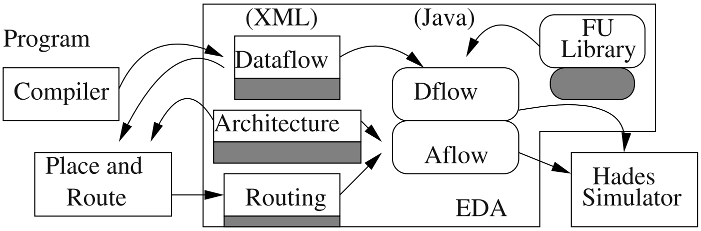
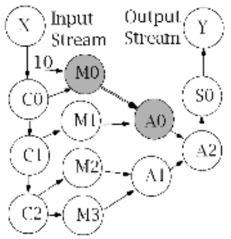


Fig. 1. Global view of the environment

The start point of the exploration flow is the dataflow specification, which can be automatically generated by a compiler from the input program in an imperative programming language (e.g., [9]). Each dataflow operator is directly implemented with a functional unit (FU). The FU behavior is specified in Java. A library of FUs has been developed to assist simulation of the dataflow representation, implemented in a given array architecture, or as an application-specific design (ASIC). A typical FU integrates an ALU, a multiplier or divider, input/output FIFOs, and the control unit to implement the handshake mechanism (see Fig. 3b). The FU is the main component of each PE in the array architecture. A ready/acknowledge based protocol controls the data transfer between FUs or PEs. Parameters such as protocol delay, FIFO size, FU granularity are global in the context of an array architecture but can be local for an application-specific dataflow implementation.

```
for (j=0; j<M; j++) {
  sum = 0;
  for (i=0; i<4; i++) {
    sum+=x[i+j]*h[i];
  }
  y[j]=sum>>15;
}
```

(a)



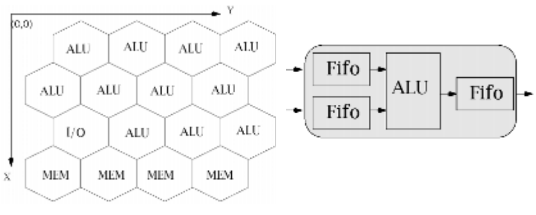
(b)

```
...<COMPONENT unit="ALU" opera-
tion="IMUL"  name="M0">
  <PORT name="A" value="10"/>
</COMPONENT>
<COMPONENT unit="ALU"  opera-
tion="IADD" name="A0" />...
<SIGNAL name="wire3">
  <SOURCE name="M0" port="Y"/>
  <SINK name="AO" port="A"/>
</SIGNAL>...
```

(c)

Fig. 2. 4-tap FIR example: (a) original source code; (b) graph of a static dataflow implementation; (c) XML representation

For simulating either the array architecture or the application-specific design we use the Hades simulation environment [10]. Hades is a pure Java component-based simulator, with a user-friendly interface, a discrete-event based simulation engine, support for hierarchical design, and a flexible waveform viewer. Hades permits to simulate components at different abstract levels, and thus it can also be used to simulate a data-driven array coupled to a standard microprocessor specified in a behavioral level.



(a) (b)

```
<CELL name="ALU" sym-
bol="ALU"/> ....
<LAYOUT length="4" width="4"
             symbol="ALU"/> ....
<!-- override special cells -->
<LAYOUT x="2" y="0" sym-
bol="I/O"/> ...
<!-- all the cells in line 3
    are of type MEM       -->
<LAYOUT x="3" symbol="MEM"/>
```

(c)

Fig. 3. (a) Hexagonal data-driven architecture; (b) Functional unit; (c) XML specification

Our environment supports two simulation flows: Dflow, for application-specific implementations (e.g., ASIC or FPGA), and Aflow, for implementations in a user-defined data-driven array architecture. With Dflow, the dataflow representation is translated to a Hades Design and simulated (see Fig. 1). At this level, the maximum throughput, the buffer size, the operator granularity and type can be explored, as well as other parameters. With Aflow, the implementation in a data-driven array architecture is simulated. The array architecture is described in an XML file. The placement and routing of the implementation is also specified using an XML file. Another XML file is used to specify the properties of each operator, such as the delays, the number of pipeline stages, etc.

An array architecture can be defined as a set of PEs connected with different topologies. To explore different array architectures, we use an XML dialect to specify the properties of each target architecture, such as its array topology, its interconnection network (mesh, hexagonal or other), the PE's placement, and local parameters like the FIFO depth of the FUs. As an example, Fig. 3a shows an architecture, with a hexagonal array topology, composed by three distinct PEs: ALU, Input and Output ports, and internal Memories. An extract of the XML specification of the array architecture is shown in Fig. 3c.

3 Experimental Results

In this section, we provide an example of a data-driven architecture to illustrate the proposed environment. The example consists of a data-driven hexagonal array with bidirectional neighbor connections. Each array cell may contain an FU for operations (arithmetic, logical, merge, copy, and split operations, etc.) or can be used as a routing cell, or both. For this example we use 32-bit width FUs and a 4-phase asynchronous handshake mechanism. Experiments are shown for different versions of the FIR filter (see Fig. 2a). Table 1 shows the number of FUs needed and the results obtained by simulating FIR implementations having different number of taps (2, 4, 8, and 16) and executing data streams of sizes 1 K and 16 K. These simulations are used to evaluate the maximum throughput achieved before mapping the examples in an array architecture. The simulations have been done in a Pentium 4 (at 1.8 GHz, 1 GB of RAM, with Linux).

Table 1. Results when simulating a FIR filter

Example	# filter taps	# FUs	Simulation time (sec)	
			Data size: 1 K	Data size: 16 K
FIR-2	2	7	1.31	5.16
FIR-4	4	13	2.34	9.09
FIR-8	8	26	2.98	17.89
FIR-16	16	49	4.68	38.87

Table 2. Results of FIR-4, using a hexagonal array with size 6 × 6, a data stream of 500 words: (1) operations; (2) routing; (3) operations and routing

Mapping	#PEs			Input FIFOs	Output FIFOs	FIFOs size	Throughput compared to maximum	Simulation time (sec)
	(1)	(2)	(3)					
1	10	0	3	(1)	-	-	50 %	4.26
2	13	4	0	(1),(2)	-	2	100%	4.42
3	13	4	0	(1)-(2)-(3)	-	1	80 %	4.40
4	10	0	3	(1)-(3)	(1)-(3)	1	100%	3.76

Table 2 shows results obtained by different mappings of the FIR-4 (1st col.) on the hexagonal array. Columns 2 to 4 show the number of PEs of the array: only used to perform an FU operation (2nd col.), only used for routing (3rd col.), and used for both routing and operation (4th col.). The simulations also take into account the existence of FIFOs in the inputs, outputs, or in both, for each PE and its size. Columns 5 and 6 illustrate where the FIFOs are used. Column 8 compares the array throughput to the application-specific implementation. The maximum throughput is achieved by mappings 2 and 4. As shown, the simulations are fast enough to allow a very efficient exploration of a significant number of topologies.

4 Conclusions

This paper presents an environment to simulate and explore data-driven array architectures. The environment permits to evaluate architectures with different topologies, as well as different features of a specific architecture. It uses Java and XML technologies, and Hades, a Java simulation engine, to achieve a powerful environment that can hardly be achieved using an HDL. Ongoing work focuses on research algorithms to place and route a given dataflow implementation into the user-defined data-driven architecture.

References

1. R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," In *Int'l Conf. on Design, Automation and Test in Europe (DATE'01)*, Munich, Germany, March 12-15, 2001, pp. 642-649.
2. A. H. Veen, "Dataflow machine architecture," in *ACM Computing Surveys*, Vol. 18, Issue 4, Dec. 1986, pp. 365-396.
3. R. Hartenstein, R. Kress, H. Reinig, "A Dynamically Reconfigurable Wavefront Array Architecture," in *Proc. of the Int'l Conference on Application Specific Array Processors (ASAP'94)*, Aug. 22-24, 1994, pp. 404-414.
4. V. Baumgarte, et al., "PACT-XPP – A Self-reconfigurable Data Processing Architecture," In *Journal of Supercomputing*, Kluwer Academic Publishers, Sep. 2003, vol. 26, iss. 2, pp. 167-184.
5. N. Bansal, et al., "Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures," in *Design, Automation and Test in Europe Conference (DATE '04)*, Paris, France, Feb. 16-20, 2004, pp. 474-479.
6. L. Bossuet, G. Gogniat, and J. Philippe, "Fast design space exploration method for reconfigurable architectures," In *Int'l Conference on Engineering of Reconfigurable Systems and Algorithm (ERSA'03)*, Las Vegas, Nevada, USA, 2003.
7. R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Generation of Design Suggestions for Coarse-Grain Reconfigurable Architectures," in *10th Int'l Workshop on Field Programmable Logic and Applications (FPL'00)*, Villach, Austria, Aug. 27-30, 2000.
8. W3C: Extensible markup language (XML), http://www.w3.org/XML/
9. J. M. P. Cardoso, and M. Weinhardt, "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture," in *12th Int'l Conference on Field Programmable Logic and Applications (FPL'02)*, LNCS 2438, Springer-Verlag, 2002, pp. 864-874.
10. N. Hendrich, "A Java-based Framework for Simulation and Teaching," in *Proceedings of the 3rd European Workshop on Microelectronics Education (EWME'00)*, Aix en Provence, France, 18-19, May 2000, Kluwer Academic Publishers, pp. 285-288.
11. W3C: The extensible stylesheet language family (XSL), http://www.w3.org/Style/XSL

FPGA Implementation of a Novel All Digital PLL Architecture for PCR Related Measurements in DVB-T

Christian Mannino, Hassan Rabah, Camel Tanougast, Yves Berviller, Michael Janiaut, and Serge Weber

U.H.P., Faculté des Sciences
Laboratoire d'Instrumentation Electronique de Nancy
BP239, 54500 Vandoeuvre-les-Nancy, France
christian.mannino@lien.uhp-nancy.fr

Abstract. The MPEG-2 DVB Transport Stream domain carries in addition to audio and video data a Program Clock Reference(PCR). This PCR is used to synchronize the MPEG-2 decoder clock on the receiver side for a given program. The PCR values can be affected by an offset inaccuracy due to encoder imperfection or by the network jitter. The measurement of different PCR parameters like drift, precision and jitter are necessary for evaluating the decodability efficiency. These measurements are generally achieved using a Phase Lock Loop and a set of measurement filters as it is recommended in the DVB-T QoS measurement standard. In this paper, we propose a FPGA implementation of an all digital PLL and its associated measurement filters. We demonstrate how it is possible to process all available programs in a DVB-T transport stream by using an FPGA with an associated embedded processor.

1 Introduction

The MPEG-2 transport stream (TS) multiplex is transmitted through the network in asynchronous mode. The recovery of a 27 MHz clock on the decoder side synchronized to the encoder clock is necessary to regenerate video signals. Samples of this clock called Program Clock Reference (PCR) are transmitted within the stream. These PCRs are affected by the network jitter. Measurement of interval between PCRs arrivals, PCR precision and accumulated PCR jitter are necessary for the evaluation of quality of service. These measurements are generally based on the use of a Phase Lock Loop (PLL) associated to a set of filters as is recommended by the DVB-T standard for the QoS measurement [1].

Several clock recovery methods based on statistical signal processing like Least-square Linear Regressive (LLR) [3] are used instead of a PLL. Their performances are better than those of analog PLLs in terms of time response and jitter reduction. However, the LLR method requires many FPGA resources and an external VCO. Moreover, in our case, it is necessary to reconstruct about ten program clocks within the transport stream at the same time, which require to use many VCOs which is unpractical for a hardware implementation.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1027–1031, 2004.

In order to overcome this problem and aiming FPGA implementation, we propose in this paper an all digital architecture to process all programs in real time. This architecture is based on an ADPLL. Concerning the Digitally Controlled Oscillator (DCO), several techniques exist like the use of a ring oscillator [2]. This type of oscillator is mainly used in full custom design due to the delay time of the inverters that must be well controlled for a good stability [4].

2 System Description

2.1 General Principle

To perform the PCR related measurements, the system has to recover the transmitter clock from time stamps presented at its input. These time stamps are generated by the MPEG-2 encoder and transmitted within the transport stream. The measurement system provides the PCR parameters corresponding to a specific program that are displayed in a monitoring system.

The measurement system is composed of an ADPLL that tracks the transmitter frequency and a set of filters performing parameters extraction. The ADPLL is composed of four main blocks as shown in figure 1. The first block is the comparator that computes the PCR error corresponding to the difference between the received PCRs denoted $PCR_e(n)$ and the locally estimated one, denoted $PCR_r(n)$. The synchronization is achieved when this difference equals zero. The second block corresponds to the loop filter. Its output is used to trigger the signal frequency produced by the DCO, representing the third block. This block delivers a clock signal with an average frequency depending on low variations of the filtered error [5]. The last block represents the counter generating the local $PCR_r(n)$.

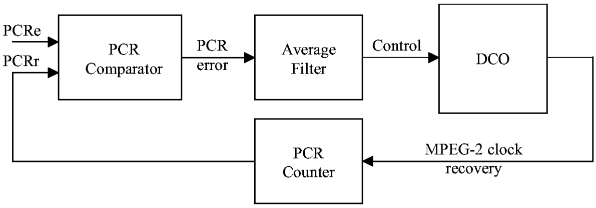


Fig. 1. Bloc diagram of the ADPLL.

Digitally Controlled Oscillator. The Digitally Controlled Oscillator (DCO) [5] delivers a clock signal with an average frequency over a time interval T_{PCR} which represents the arrival duration of two successive $PCRs$. The working frequency range of the system is $[F_{+} = 27MHz + 810Hz$, $F_{-} = 27MHz - 810Hz]$. The step response of the system can have an overshoot, so the maximum frequency is set to $F_{max} = 27MHz + 2\text{x}810Hz$. Those frequencies are obtained from a clock running at 4xF_{max}. The DCO requires a high frequency high precision jitter free clock named $4F_{max}$ for which some rising edges are regularly cancelled. The highly stable clock is divided by four or five in the interval T_{PCR}, according to a decision threshold.

Loop filter. The loop filter structure is determined for a zero steady state error. In fact we aim a zero phase error for a phase step and zero frequency error for a frequency step. This can be obtained by choosing a transfer function $F(z)$ of the loop filter that has a proportional term G_1 and an integral term G_2. Equation 1 gives the transfer function of the filter.

$$F(z) = \frac{S\_filter(z)}{Sub(z)} = G_1 + G_2 . \frac{1}{1 - z^1} \tag{1}$$

3 ADPLL Architecture Implementation Results

3.1 System Verification and Optimization

To verify the functionality of the ADPLL, the overall system is modelled in a hierarchical way with SystemC [6]. This method allows the use of different abstraction levels.

The loop filter used is described in floating point. For harware implementation, a fixed-point format is determined with a heuristic approach: the output of a fixed point filter is compared with the output of a floating point filter. The maximum absolute error ΔS_Filter is stored and we compute then the frequency error ΔF_r that could appear at the DCO output with equation 2 where K_{DCO} is the DCO slope.

$$\Delta F_r = K_{DCO} . \Delta S\_Filter \tag{2}$$

The fractional part size is then increased until obtaining the optimized size which corresponds to a frequency error lower than 10 Hz. The results obtained are presented in figure 2. The fixed point fractional part size is going from 5 bits to 10 bits and the results are given with G1 = 5 and G2 = 0.2.

We can see that between 8 bits and 10 bits, we have the same frequency error which is 7 Hz. So, we can conclude that a good choice for the fixed point fractional part size is 8 bits.

For the ADPLL implementation, we translate the SystemC code into synthetizable VHDL language with the Nepsys Tool [7], whereas the measurement filters algorithms are computed with an embeded Nios processor.

3.2 Experimental Setup

For testing the functionality of our measurement system, we use an Altera EXCALIBUR platform based on an APEX EP20K200EFC4842x FPGA. The board is composed of a 33.33 MHz oscillator and a zero-skew clock distribution circuitry. For generating the $4F_{max}$ frequency for the DCO, we use one of the two

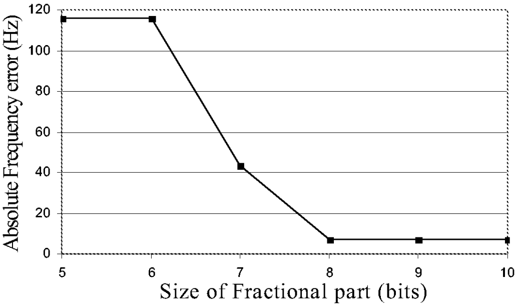


Fig. 2. DCO output error difference between the floating and the fixed point filter.

PLLs available in the APEX circuit. These PLLs make it possible to obtain a 27080625 Hz frequency from the 33.33 MHz clock by applying a 13/4 multiplication factor.

3.3 Synthesis Results

VHDL synthesis results are given in table 1. The information are obtained from the synthesis report of the measurement system implementation. This table shows that the system requires 2823 Logic Cells corresponding to 34 % of the total APEX 200 device resources. Nevertheless, the core processor represents more than a half of the system resources. The ADPLL resource requirements represent only 14 % of total available resources, moreover these resources are mainly allocated to the divider bloc.

Table 1. Synthesis results.

Bloc function		Logic cells (LC)	% of used resources
PCR comparator		182 (6.44 %)	2.2 %
Loop filter		341 (12.1 %)	4.1 %
DCO	divider	507 (18 %)	6.1 %
	FSM	87 (3.1 %)	1.1 %
CPU core (NIOS)		1706 (60.4 %)	20.5 %
Total (20K200; 8320 LC)		2823	34 %

3.4 Experimental Results

The Drift Rate (*DR*) and the Overall Jitter (*OJ*) measurement point is located at the PCR comparator output. The Frequency Offset (*FO*) measurement is achieved by processing the loop filter output. To extract these parameters, measurement filters are used as defined in the DVB standard documentation [1]. Table

2 gives the maximum excecution time for each PCR parameter on a Nios CPU solution. The total time is 1345 μs. Assuming that the minimum PCR arrival time is 20 ms, we can hope to process all the PCR parameters of each program in a DVB-T transport stream that contains about ten programs.

Table 2. Execution time of the PCR parameters.

PCR parameters	PCR_FO	PCR_DR	PCR_OJ	Total
Max. excecution time	365 μs	400 μs	580 μs	1345 μs

4 Conclusion

In this paper we proposed an ADPLL solution implementable in FPGA for the clock recovery in DVB-T. The system verification is achieved with SystemC. Associated to a set of filters, this ADPLL can extract the PCR_FO, the PCR_DR and the PCR_OJ parameter. The originality of our DCO architecture proposal makes it possible to implement as much ADPLL as existing programs. These measurements are necessary for QoS evaluation in DVB-T. The performances and functionality of the ADPLL architecture are verified with the implementation results. The measurement in real time of the PCR parameters for about ten programs is currently in progress.

References

1. ETSI: Digital Video Broadcasting (DVB); Measurement guidelines for DVB systems. Digital Video Broadcasting, http://www.etsi.org (2001)
2. Hwang, I., Song, S., Kim, S.: A Digitally Controlled Phase-Locked Loop With a Digital Phase-Frequency Detector for Fast Acquisition. IEEE Journal of Solid-State Circuits Vol. **36**, No. **10** (2001)
3. Noro, R.: Synchronisation over packet-switching network : theory and applications. Ph. D., EPFL, Lausanne (2000)
4. Olsson, T., Nilsson, P.: An All-Digital PLL Clock Multiplier. Proceeding IEEE Asia-Pacific Conference on ASIC (2002)
5. Mannino, C., Rabah, H., Tanougast, C., Berviller, Y., Weber, S.: Conception d'une architecture dediee à l'instrumentation pour la mesure de gigue en DVB-T. Colloque Interdisciplinaire en Instrumentation, ISBN 2-7462-0828-8, T.1 - pp. 281-288 (2004)
6. SystemC: Version 2, User's Guide. www.systemc.org (2002)
7. Nepsys: Nepsys Version 1.0, User's Guide. Prosilog, http://www.prosilog.com (2002)

A Dynamic NoC Approach for Communication in Reconfigurable Devices[*]

Christophe Bobda, Mateusz Majer, Dirk Koch, Ali Ahmadinia, and Jürgen Teich

Department of Computer Science, University of Erlangen-Nuremberg
Am Weichselgarten 3, D-91058 Erlangen, Germany
{bobda, mateusz, dirk.koch, ahmadinia, teich}@cs.fau.de
www12.informatik.uni-erlangen.de

Abstract. A concept for solving the communication problem among modules dynamically placed on a reconfigurable device is presented. Based on a dynamic network-on-chip (DyNoC) communication infrastructure, components placed at run-time on a device can mutually communicate. A 4x4 dynamic network-on-chip communication infrastructure prototype, implemented in an FPGA occupies only 7% of the device area and can be clocked at 391 MHz.

1 Introduction

On-line algorithms [2] have been developed in the past for temporal placement on reconfigurable device. Almost all those algorithms consider the modules to be rectangular boxes without communication among each other. In [1] a new on-line placement strategy which takes into account the communication information is presented. However, this method helps only to reduce the communication cost among the modules by placing connected modules near to each other on the chip. It does not determine how the communication will be realized. The dynamic placement of components on a reconfigurable device requires a viable communication infrastructure to support the dynamic communication requirements. This paper presents a new network-on-chip-based concept to dynamically handle the communication between modules on a reconfigurable device. A case study is provided as an FPGA implementation of a dynamical network-on-chip arranged in a 4x4 mesh.
The rest of the paper is organized as follow: in Section 2 we present some previous approaches for handling dynamic on-chip communication. The requirements on the architecture are presented in Section 3. Section 4 deals with the dynamic connection of components on the network. In section 5, an FPGA-implementation of a network infrastructure is shown. Section 6 concludes the work.

[*] Supported in part by the German Research Foundation (DFG), SPP 1148 (Rekonfigurierbarer Rechensysteme)

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1032–1036, 2004.

2 Related Work

Recently, Network-on-Chip (NoC) have been shown to be a good solution to support communication on System-on-Chip [3]. Dally et al [3] have proven that NoCs encounter many advantages (performance, structure and modularity) toward global signal wiring. A chip employing a NoC is composed of a set of network clients like DSP, memory, peripheral controller, custom logic, etc. Instead of connecting the modules using dedicated routing wires, they are connected to a network that route packets among them. Maresceaux et al [4] proposed the use of a NoC communication infrastructure as a component of an operating system for reconfigurable systems. However, their NoC approache supports only fixed processing modules defined as tile on the chip at compile time. We seek an approach which allows modules being placed on a reconfigurable device at run-time to communicate with other on-chip modules as well as off-chip modules without restriction on the placement.

3 Communication Infrastructure

To be able to dynamically establish communication between newly placed components, two approaches can be followed: A packet-based communication or an on-line signal routing. On-line signal routing can be used to establish a set of dedicated point to point communications. However the routing of signals is a computational intensive task which can only be efficiently computed off-line. Moreover, if signals are routed on a given area and a new component is placed in that area, then it will affect the signals currently routed. The packet-based approach has the advantage that the alteration of the network will not hinder the communication, since packets can always find their way in a strongly connected network. The packet-based approach can be realized using a *dynamic NoC approach*. In a static NoC, the clients are placed in rectangular tiles on the chip and communicate with other clients via a fixed network. It has been shown in [3] that the area occupied by the network logic is small (about 6 % of the tile's area). This ratio is expected to drastically decrease with the rapid growth in size of reconfigurable devices. In order to have a better network logic/PE ratio and make an optimal use of the resource, we impose the following requirements to the architecture of the communication infrastructure:

- The PEs should be flexible, but coarse grained computing elements. With fined grained PEs, the rapport network logic/PEs becomes big thus wasting the device area.
- Each PE should have access to the network. This condition is very important since it allows any module being placed on the reconfigurable device to always access the network via one of its surrounding PEs.
- PEs should directly communicate with their neighbors. This is helpful because it allows wiring to be done locally within a module boundary.
- The network logic should be flexible enough to be used within the module to which it belongs. Whenever a component is placed on a given region of the

device, the network modules deep inside the module area cannot be used for network operations. Therefore, they should be used as additional resource for the module they belong to. A component will use the resource of its network elements for the time it is running.

Figure 1 shows a communication infrastructure as described below on a reconfigurable device.

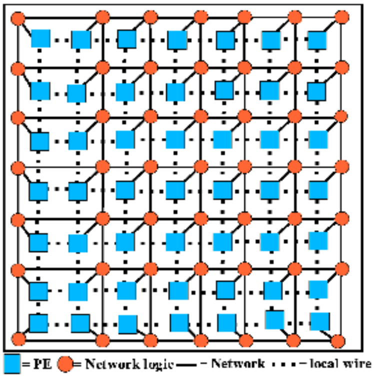


Fig. 1. The dynamic NoC approach for reconfigurable communication

Having defined the communication infrastructure of the device, the main questions are to know how to develop components which can be dynamically connected to the network at run-time. We provide some answers to those questions in the next sections.

4 Network Access

Each task is implemented as a component, represented by a rectangular box and stored in a database. A box encapsulates a circuit implemented with the resource in a given area. Therefore, a component can access the network using one of the network elements on its boundary. Without loss of generality, we select the network element on which the upper right PE of the component is attached. After the placement of a new component on the device, the placer will set the coordinates of the feasible network element as the module address for communication with the module. When placed on the device, components hide part of the network which is restored when they complete their execution. This makes the network dynamic. We call such a network a ***dynamic network-on-chip (DyNoC)***.
During the temporal placement, modules will always be placed such as to main-

tain a strongly connected[1] network. As shown in figure , each placed component is always surrounded by the network elements.

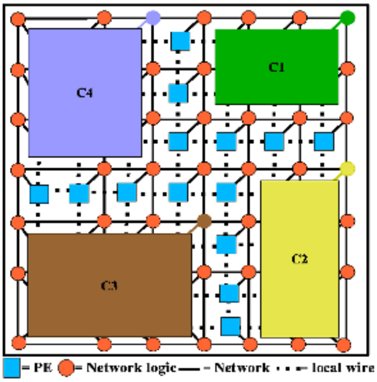


Fig. 2. A temporal placement on the DyNoC

While in a static NoC, each router always has four active neighbor-routers[2], this is not always the case in the DyNoC presented here. Whenever a component is placed on the device, it covers the routers in its area. Since those routers cannot be used, they will be deactivated by setting the corresponding control signals until the component completes its execution. Upon completing its execution, the deactivated routers are set to their default state. With this, a slightly modification of the routing algorithm is required. Before sending the packet in a given direction, a router must check if the router in that direction is activated. If so, then the packet will be routed perpendicular to the direction previously chosen.

5 Case Study

To investigate the feasability of the concept presented here, we have implemented a DyNoC similar to that described in Section 3 on an FPGA Virtex II 6000. The resulting circuit is presented in Figure 3. The complete communication infrastructure is made upon sixteen routers interconnected in a mesh network. The routers are connected by a 32-bit wide bus and 4 control lines and contain six 32-bit wide FIFO buffers with a depth of 4. The complete design occupies 7 % of the device area. If large components are placed on the device, they will cover a large set of routers, thus reducing the total area used by the routers. The router has also been implemented as described in the previous Section. Each

[1] A network is strongly connected, if for each pair of network elements a path exists which connects the two elements

[2] The neighbor-routers of the routers around the chips are assumed to be the package pins through which external modules can access the network

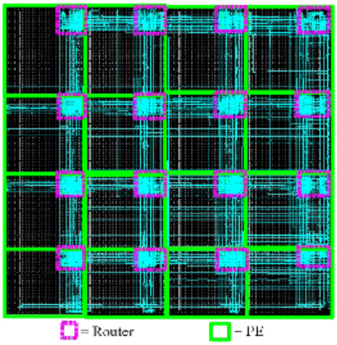


Fig. 3. FPGA implementation of a 4x4 DyNoC

router occupies less than 0.5 % of the device area and has a latency of 2.553 ns corresponding to a frequency of 391 MHz. Because the maximum number of routers to traverse is 6, two components running on the device with a frequency of 50 MHz can send and receive the packets without delay in their execution, if the network is free.

6 Conclusion

In this paper, we have presented a concept for handling the problem of dynamic communication among modules dynamically placed on a reconfigurable device. We have shown how a dynamically changing network-on-chip can be used as a viable communication infrastructure. A prototype of a 4x4 dynamic network-on-chip implemented in an FPGA has been built. The complete network infrastructure occupies only 7% of the device and the network elements can be clocked at 391 MHz.

References

1. Ali Ahmadinia, Christophe Bobda, Marcus Bednara, and Jürgen Teich. A new approach for on-line placement on reconfigurable devices. In *Proc. of IPDPS-2004, Reconfigurable Architectures Workshop (RAW-2004), Santa Fe NM, USA, April 26-27*, 2004.
2. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *In IEEE Design and Test - Special Issue on Reconfigurable Computing*, January-March:68–83, 2000.
3. William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Las Vegas, NV, June 2001.
4. T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins. Networks on Chip as Hardware Components of an OS for Reconfigurable Systems. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, September 2003.

Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems[*]

M. Huebner, M. Ullmann, L. Braun, A. Klausmann, and J. Becker

Universitaet Karlsruhe (TH), Germany
http://www.itiv.uni-karlsruhe.de/
{huebner,ullmann,braun,klausmann,becker}@itiv.uni-karlsruhe.de

Abstract. Current trends show that in future it will be essential that various kinds of applications are running on one chip. These require an efficient and flexible network on chip which is able to adapt to the demands of supported modules. This makes it necessary to think about what kind of network on chip will meet these requirements. This paper describes an approach for a reconfigurable network on chip which allows adapting the performance and topology at run-time to the demand of the application running on Xilinx FPGA.

1 Introduction

Today field programmable gate-arrays (FPGAs) are mainly used for rapid-prototyping purposes. They can be reconfigured many times for different applications. Modern state-of-the-art FPGA devices like Xilinx Virtex FPGAs [1] additionally support a partial dynamic run-time reconfiguration which reveals new aspects for the designer who wants to develop future applications demanding adaptive and flexible hardware. Especially in the domain of mobile computing high-end mobile communication applications will benefit from the capabilities of the new generation of reconfigurable devices. A new approach to create systems which are able to manage configuration are run-time systems. These systems use the flexibility of an FPGA by changing the configuration partially. Only the necessary functions are configured in the chip's memory. By demand a function can be substituted by another while used parts stay operative. To solve the problem of substitution and I/O management the configuration needs a main module controlling the tasks. Additionally this module controls the on chip intercommunication bus to prevent an overhead of bus size. To be able to deal with the amount of needed bus resources for a special application or a scenario of different applications running in parallel on the FPGA, an adaptive bus structure has to be developed. [8] and [9] raise this issue with the main focus on energy saving and basis for new design methodologies. In [7] an approach using a serial connection as communication path is described. The communication structure in this approach is static by using a dedicated bus line for message transfer. Changing demands of

[*] This research work is sponsored by the German special research program "Reconfigurable Computing Systems" - SPP 1148 from the Deutsche Forschungsgemeinschaft DFG

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1037–1041, 2004.

communication require a network on chip which provides this feature. This paper describes one approach for such a network on chip. Section 2 describes the target system including all components. In section 3 the method of communication is shown. Finally section 4 contains conclusion and an outlook on further projects.

2 Target System

The target system and its reconfigurable network are realized on a partial dynamic reconfigurable FPGA. For a big variety of user tasks five dynamically reconfigurable areas are defined. More information for dynamic reconfiguration can be found in [2]. Figure 1 shows a schematic view of the complete system.

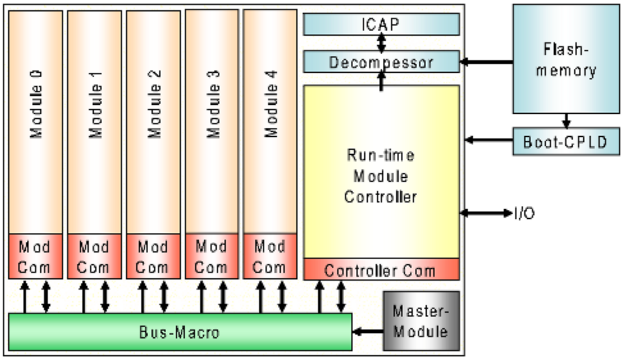


Fig. 1. Structural scheme of the target system with five reconfigurable modules

The reconfigurable areas are described as modules 0 to 4. All these modules can contain different user-defined functions running in parallel. The reconfigurable network offers the possibility to adapt the network resources to the different demands of the modules. The access to the network is given by the “Master-Module”. A “Module Communication Unit” (Mod Com) is used to connect the module to the network. Similar to the Mod Com Unit the “Run-time Module Controller” is linked via a “Controller Communication Unit” (Controller Com). In contrast to the Controller Com the Mod Com is configured into a reconfigurable module area and therefore together with the user-function a part of the partial reconfiguration bit stream. The hardware platform of the system is the Xilinx Virtex-II FPGA XC2V3000. Xilinx also offers a microprocessor IP-core called Microblaze. The “Run-time Module Controller” applies this IP-Core. This Controller manages the external communication and the partial dynamic reconfiguration which is executed by using the Decompressor- and ICAP-Unit inside as well as Flash-memory and Boot-CPLD outside the FPGA. A more detailed description of the run-time system can be found in [3]. In this implementation the Module Controller additionally controls the communication topology.

2.1 Network on Chip

The network on chip is based on a serial communication protocol (section 3). Figure 2 shows an example of a Bus-Macro using four bi-directional bus lines. The connection to a reconfigurable module is shown on the left side and to the Module Controller on the right. Clock, clock-enable and reset signal are left out in Fig. 2. During reconfiguration a dynamic reconfigured slot is disabled via an enable slot signal.

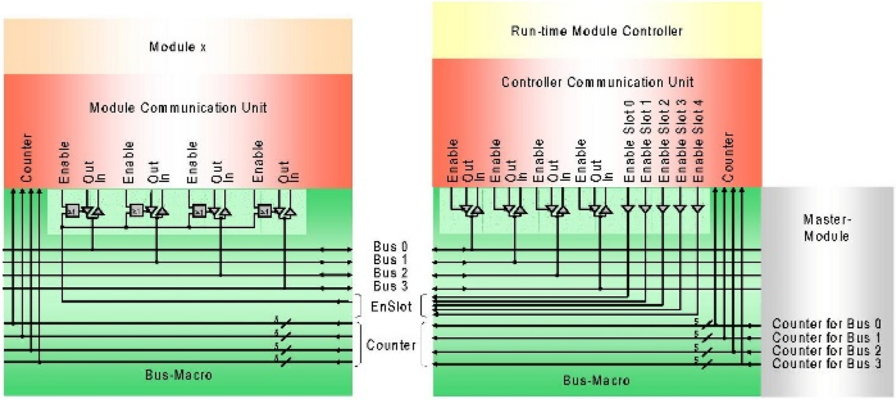


Fig. 2. Interfaces of the Bus-Macro

The Master-Module provides a special five bit counter for each bus line which divides the bus-access time into timeslots. The counters have the same structure but don't necessarily have the same state simultaneously. Every slot is provided with the four 5 bit counter signals. The bus-arbitration is based on time-slots. The Mod Com Units and the Controller Com Unit have fixed amounts of time-slots corresponding to the counter values for access to one or more bus lines within a counter period. These timeslots can be released or locked. So the distribution of network-resources and the network structure can be adapted to the required bus-performance of the modules during run-time. At the moment the Module Com Unit gets the release signals for its timeslots from the Controller Com Unit. It is planned to use valid-timeslot bits which are contained in the Module Com Unit instead. Then the ICAP Module will control these bits in order to change the communication topology.

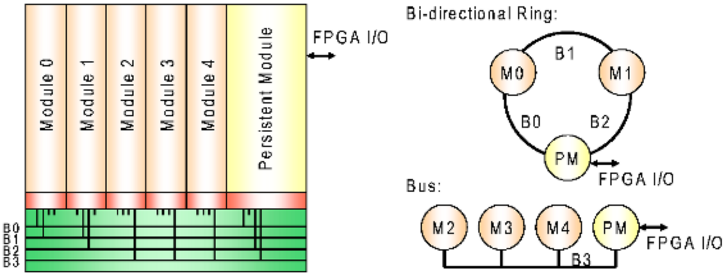
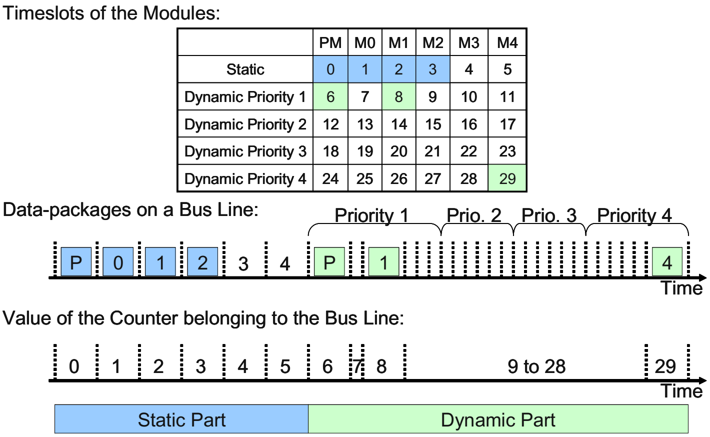


Fig. 3. Example of a communication topology



Timeslots of the Modules:

	PM	M0	M1	M2	M3	M4
Static	0	1	2	3	4	5
Dynamic Priority 1	6	7	8	9	10	11
Dynamic Priority 2	12	13	14	15	16	17
Dynamic Priority 3	18	19	20	21	22	23
Dynamic Priority 4	24	25	26	27	28	29

Fig. 4. Access Scenario of a Bus Line

Besides the bus structure the reconfigurable NoC can be configured to different net-structures. A bus line can be disconnected from a module by locking all belonging time-slots. Thereby a star or ring communication topology of the reconfigurable network can be realized during run-time (Fig.3). Also mixed topologies are possible.

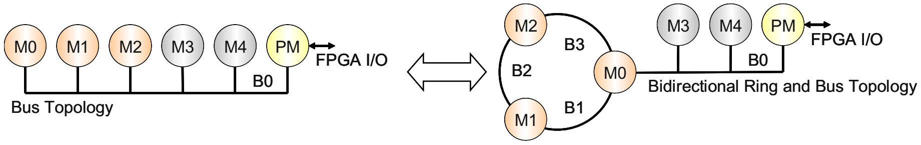


Fig. 5. Communication Topology Change between Bus and Ring

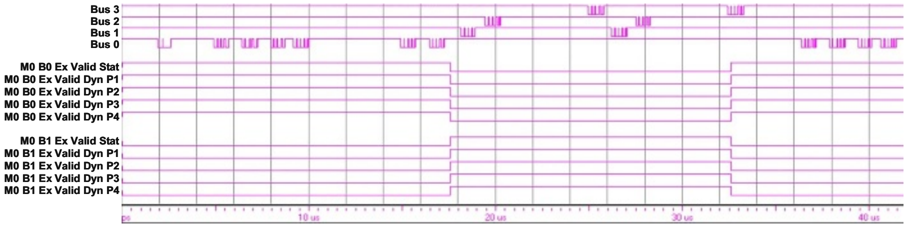


Fig. 6. Post PAR Simulation of a Communication Topology Change

3 Module Intercommunication

The used serial communication protocol is similar to the FlexRay [5] protocol. Both are divided into a static and a dynamic part (Fig. 4). The static part keeps the predictability of data transmission. The dynamic part increases bandwidth if necessary. A counter controls the transmission. Every module has a list of numbers which represents the time-slot numbers (table Fig.4). The dynamic part is subdivided into priority levels. If the counter of a bus line reaches a number from this list, the module is able to begin its data transmission. Since every module has its own unique list data-package collisions are avoided. Besides a data block the data-packages contain a receiver and a sender address. Whereas a static data-package has a fixed length, the length of a dynamic data-package is flexible.

4 Conclusions and Further Work

This approach of a network on chip makes a dynamic communication topology change with the next rising edge of the clock signal possible (keep setup-time). Data-packages which are transmitted during switching are not influenced (avoidance of data loss). So far the maximum real dataflow is 4.89 MB/s per bus line. The overhead is 2.2 %. The maximum duration from a module send request to the bus line release is 261.6 µs (using all timeslots). These calculations are based on a static data block length of 8 byte and a dynamic of 256 byte. The system clock has 40 MHz. Figure 5 and 6 show a communication topology change from bus to ring and back. This test-design is running at 40 MHz. Fig. 6 shows beside the four bus lines some timeslot-release signals which make the topology change possible.

This network on chip will be used in the future for multimedia applications but also for modules with control functions. The possibility for energy saving with run-time optimisation of the bus will also be a part for further investigations.

References

1. www.xilinx.com
2. Two Flows for Partial Reconfiguration: Module Based or Difference Based, Xilinx XAPP290, November 25, 2003
3. M. Ullmann, B. Grimm, M. Huebner, J. Becker: „An FPGA Run-Time System for Dynamical On-Demand Reconfiguration", RAW04, Santa Fé, USA
4. M. Huebner, M. Ullmann, F. Weissel, J. Becker: „Real-time Configuration Code Decompression for Dynamic FPGA Self-Reconfiguration", RAW04, Santa Fé, USA
5. http://www.flexray.de/
6. B. Blodget, S. McMillan: "A lightweight approach for embedded reconfiguration of FPGAs", Date 03, Munich Germany
7. J.C. Palma, A. Vieira de Melo, F. G. Moraes, N. Calazans, "Core Communication Interface for FPGAs", Proceedings of 15th Symposium on Integrated Circuits and Systems Design (SBCCI), 2002,Porto Alegre BRAZIL, Page(s): 183 –188
8. L. Benini, G. De Micheli: "Networks on Chip: A New Paradigm for Systems on Chip Design", Date 02, March 3~7, Paris France
9. F. Worm, P. Ienne, P. Thiran, G. De Micheli: "An Adaptive Low Power Transmission Scheme for On-chip Networks", ISSS 02, October 2002, Kyoto Japan

FiPRe: An Implementation Model to Enable Self-Reconfigurable Applications

Leandro Möller, Ney Calazans, Fernando Moraes, Eduardo Brião,
Ewerson Carvalho, and Daniel Camozzato

Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS),
Av. Ipiranga, 6681 - P 30/Bl 4 - 90619-900 - Porto Alegre – RS– BRASIL
{moller, calazans, moraes, briao, ecarvalho,
camozzato}@inf.pucrs.br

Abstract. ASIPs and reconfigurable processors are architectural choices to extend the capabilities of a given processor. ASIPs suffer from fixed hardware after design, while ASIPs and reconfigurable processors suffer from the lack of a pre-established instruction set, making them difficult to program. As intermediate choice, reconfigurable coprocessors systems (RCSs) contain dedicated hardware (coprocessors) coupled to a standard processor core to accelerate specific tasks, allowing inserting or substituting hardware functionalities at execution time. This paper proposes a generic model for RCSs, targeted to reconfigurable devices with self-reconfiguration capabilities. A proof-of-concept case study is presented as well.

1 Introduction

A single processor may meet the requirements of several embedded system scenarios if it is somehow parameterizable. Application-specific instruction set processors (ASIPs) and reconfigurable instruction set processors (RISPs) [1] are two opposite forms of implementing processors with regard to *design* versus *runtime* parameterization trade-offs. *ASIPs* provide flexibility and performance at the cost of extra silicon area for each new function directly supported in hardware. If the application requires a new specific functionality, the ASIP is redesigned. *RISPs* are processors where some or all instructions are implemented as dedicated hardware and loaded on demand, according to the software execution flow. Here, the highest degree of flexibility is achieved. However, the lack of a pre-established fixed instruction set makes it harder to generate the object code for new applications. This occurs because each new function must be supported at the same time by the dedicated hardware *and* the compiler.

An intermediate solution, named *reconfigurable coprocessors systems* (RCSs) is addressed in this paper. As ASIPs and RISPs, RCSs contain dedicated hardware (coprocessors) to accelerate specific tasks. However, these are not fixed at design time as in ASIPs. It is possible to insert or substitute hardware functionalities at execution time in RISPs and RCSs without having to redesign the processor. Contrary to what happens in RISPs, RCSs contain a standard processor core with a fixed instruction set, enabling the use of standard compilers.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1042–1046, 2004.

The processor and the parameterizable parts are *loosely coupled* in RCSs and *tightly coupled* in RISPs and ASIPs. Additionally, ASIPs and RISPs are inherently sequential approaches, while RCSs may benefit from the parallel execution of the processor software and dedicated computations in each coprocessor. Communication between the processor and the coprocessors can be achieved in this case through the use of e.g. interrupts.

A potential performance bottleneck faced by embedded applications in RISPs and RCSs is the latency to perform hardware reconfiguration, which can be orders of magnitude longer than the time to perform application atomic operations. To reduce or eliminate this problem, RISPs and RCSs assume the existence of an infrastructure to control the storage and the dynamic loading of hardware configurations, usually called a *configuration controller* [2].

Consider the current trend to increase the number of embedded processors in SoCs, leading to the concept of "sea of processors" systems [3], and add to this the above discussion on implementation alternatives for parameterizable embedded processors. From these, it is possible to justify the objective of this paper, which comprises proposing a generic implementation model for RCSs called FiPRe, and introducing a case study used to evaluate the ideas behind the model.

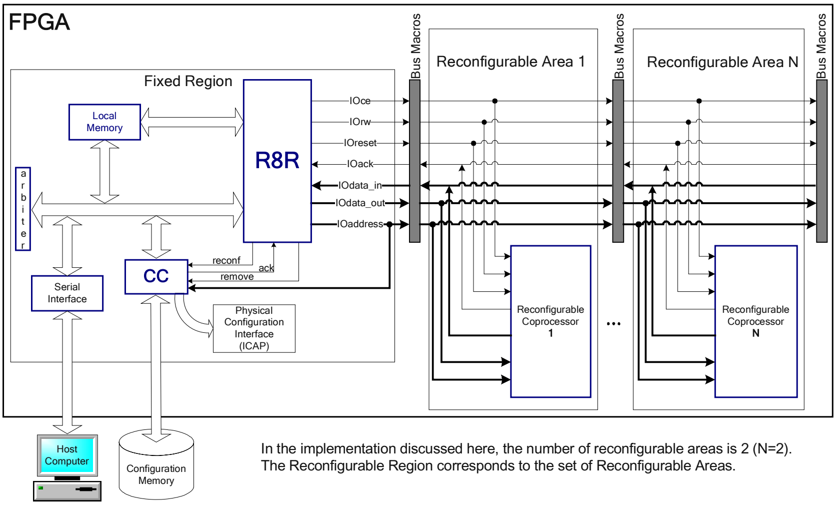
2 The FiPRe Implementation Model and the R82R Case Study

The FiPRe (**Fi**xed core **P**rocessor connected to **Re**configurable Coprocessors) model is conceived to allow self-reconfigurable applications implemented as RCSs and can be understood from the case study example in Fig. 1. First, there is a Fixed Region, which has an embedded processor to execute applications and to trigger reconfiguration actions. This region also contains a configuration controller (CC), to manage the details of the reconfiguration process. The existence of external devices intended to provide input/output capabilities for the embedded system is also part of the model. Besides, a memory is needed to store coprocessor bitstreams, a block named Configuration Memory. Finally, the model assumes the existence of a Reconfigurable Region that contains a subset of configured coprocessors. This region presents data exchange and configuration interfaces to the rest of the system.

A fixed instruction set processor provides advantage in terms of code and hardware reuse, because neither the processor nor the compiler needs to be changed in the process of developing coprocessors to achieve performance and functionality goals.

The CC handles coprocessor selection and dismiss procedures produced by the processor. When selection is executed, the configuration memory is accessed and a coprocessor bitstream is sent to the configuration interface.

In order to evaluate the FiPRe model to implement RCSs embedded systems, an example case study, named R82R was designed and implemented. The system was implemented in a single VirtexII device, with the exception of the Configuration Memory. The case study employed a soft core processor customized for the FiPRe model. The changes made in the original processor were to add specific instructions to support reconfiguration (Table 1), and a specific external interface to the reconfigurable region and to the configuration controller.



In the implementation discussed here, the number of reconfigurable areas is 2 (N=2).
The Reconfigurable Region corresponds to the set of Reconfigurable Areas.

Fig. 1. General structure of the R8NR system.

Fig. 1 displays the organization of the R82R system. The system is composed by three main modules: a host computer, providing an interface with the system user; a configuration memory, containing all partial bitstreams used during system execution; the FPGA, containing fixed and reconfigurable regions of the R82R system. The fixed part in the FPGA comprises the R8R processor [4], its local memory, containing instructions and data, a system bus controlled by an arbiter, and peripherals (serial interface and the configuration controller). The serial interface peripheral provides capabilities for communication with the host computer (an RS232 serial interface). The CC is a specialized hardware, acting as a slave of the R8R and of the host computer, which fills the configuration memory before system execution starts.

The R8R processor was wrapped to provide communication with *(i)* the local memory; *(ii)*the system bus; *(iii)* the CC; *(iv)* the reconfigurable region. The interface to the reconfigurable areas comprises three identical sets of signals interconnected through special components furnished by the FPGA vendor, called *bus macros*.

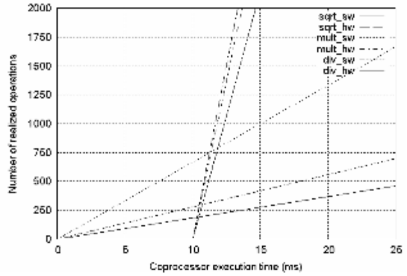
Table 1. Instructions added to the R8 processor in order to produce the R8R processor.

Reconfigurable instruction	Semantics description
SELR *address*	Selects the coprocessor identified by *address* for communication with the processor, using the *reconf* signal (see Fig. 1). If the coprocessor is not currently loaded into the FPGA, the CC automatically reconfigures some area of the FPGA with it.
DISR *address*	Informs the CC, using the *remove* signal (see Fig. 1), that the coprocessor specified by *address* is dismissed and can be removed if needed.
INITR *address*	Resets the coprocessor specified by *address*, using the *Ioreset* signal. The coprocessor must have been previously configured.
WRR *RS1 RS2*	Sends the data stored in *RS1* and *RS2* to the coprocessor selected by the last SELR instruction. *RS1* can be used for passing a command while *RS2* passes data.
RDR *RS RT*	Sends the data stored in *RS* to the coprocessor (typically a command or an address). Next, reads data from the coprocessor, storing it into the *RT* register.

3 Results

The R8NR system has been prototyped and is operational in two versions, with one and two reconfigurable areas, respectively (R81R and R82R). A V2MB1000 board from Insight-Memec was employed in the prototyping process.

Fig. 2 shows the comparison between the number of operations executed and the time to execute hardware and software versions of three 16/32-bit arithmetic nodules: multiplication, division and square root. Note that for each module, the execution time grows linearly but with different slopes for software and hardware implementations. Also, the reconfiguration time adds a fixed latency (~10 ms) to the hardware implementations. The fixed latency is an approximation of the time measured to configure one FPGA area dedicated to hold one coprocessor.



The hardware reconfiguration latency, 10ms, is dependent on the size of the reconfigurable area partial bitstream and on the performance of the CC module. This graph was plotted for a CC working at 24MHz frequency and for reconfigurable bitstreams of 46Kbytes, corresponding to a reconfigurable coprocessor with an area of roughly one tenth of the employed million-gate device.

Fig. 2. Execution time versus the number of operations for three arithmetic modules, multiplication, division and square root, implemented in HW (hw suffix) and SW (sw suffix).

The break even point for each functionality determines when a given hardware implementation starts to be advantageous with regard to a plain software implementation, based on the number of times this hardware is employed before it is reconfigured. From the graph, it can be seen that the multiplier, division and square root coprocessors are advantageous starting from 750, 260 and 200 executions without an intervening reconfiguration step. Consider the application of a filter (e.g. edge or smooth) over an image with 800*x*600 pixels. If only one operation is applied per pixel 480000 operations are executed, easily justifying the use of a hardware coprocessor. This simple experiment highlights how in practice it is possible to take advantage of RCSs, achieving performance gains, flexibility and system area savings.

The R82R case study was synthesized using Leonardo Spectrum. The area report data for the fixed modules is presented in Table 2.

Table 2. Area report data for a XC2V1000 FPGA and for 0.35 μm ASIC CMOS tecnology.

Module	ASIC Gates	LUTs	FFs	%LUTs
R82R	6331	1020	555	9.96
Memory	3139	307	366	2.99
Serial Interface	5430	616	607	6.01
CC	2790	493	218	4.81
Arbiter	157	27	15	0.26
Total	17847	2443	1761	23.85

Configuration controllers (CC) found in the literature are mostly software implementations. The CC proposed here was implemented in hardware, having a small area footprint (around 3,000 gates) and is expected to present superior performance over software versions in terms of reconfiguration speed. Another important advantage to implement the CC in hardware is that the embedded processor is free to execute tasks in parallel during the reconfiguration process.

The Modular Design method [5] employed for generating partial bitstreams, limits the size of a reconfigurable area to a minimum of 4 CLB FPGA columns. One minimum size reconfigurable area contains 1280 LUTs (each column contains 320 LUTs). Nevertheless, the implemented coprocessors use in average 140 LUTs. Therefore, it is possible to implement much larger coprocessors in these areas. Examples are simple dedicated processors, FFT operators, and image filters.

4 Conclusions

The major contribution of the present work is the FiPRe model for RCSs. An advantage of the model is the parallelism between processor and coprocessors, enabling the use of non-blocking operations. Also, the compiler does not need to be changed when a new coprocessor is added. On the other hand, an increased latency in communication may be observed, since the system parts are loosely coupled.

Also, since RCSs are reconfigurable systems, they potentially reduce the final system cost, as the user can employ smaller configurable devices, downloading partial bitstreams on demand. In addition, partial system reconfiguration makes it possible to benefit from a virtual hardware approach, in the same manner that present day computers benefit from the use of virtual memory.

Application areas for RCSs are another crucial point. Unless sound applications are developed to show real advantage of using RCSs over conventional solutions, such as RISPs and ASIPs, RCSs will remain no more than an academic exercise on an interesting subject area. Ongoing work includes performance measurement of benchmarks and improvements on the configuration controller to reduce the time wasted during partial reconfiguration.

References

[1] F. Barat; R. Lauwereins. Reconfigurable instruction set processors: a survey. In: Rapid System Prototyping (RSP´00), pp.168-173, 2000.

[2] D. Robinson; P. Lysaght. Modeling and Synthesis of Configuration Controllers for Dynamically Reconfigurable Logic Systems using the DCS CAD Framework. In: 9th Field-Programmable Logic and Applications (FPL'99), 1999.

[3] J. Henkel. Closing the SoC Design Gap. IEEE Computer, vol 36(9), pp. 119-121, 2003.

[4] F. Moraes and N. Calazans. R8 Processor Architecture and Organization Specification and Design Guidelines. 2003.
http://www.inf.pucrs.br/~gaph/Projects/ R8/public/R8_arq_spec_eng.pdf

[5] Xilinx, Inc. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Application Note XAPP290, Version 1.1. 2003.

A Structured Methodology for System-on-an-FPGA Design

P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, and W. Luk

Imperial College, London SW7 2BT, UK

Abstract. Increasing logic resources coupled with a proliferation of integrated performance enhancing primitives in high-end FPGAs results in an increased design complexity which requires new methodologies to overcome. This paper presents a structured system based design methodology, centred around the concept of architecture reuse, which aims to increase productivity and exploit the reconfigurability of high-end FPGAs. The methodology is exemplified by the Sonic-on-a-Chip architecture. Preliminary experimental investigations reveal that while the proposed methodology is able to achieve the desired aims, its success would be enhanced if changes were made to existing FPGA fabrics in order to make them better suited to modular design.

1 Introduction

Field Programmable Gate Arrays are an increasingly attractive choice for highly integrated digital systems due to their significantly lower design costs when compared to semi-custom integrated circuit design. The trade-off in FPGA-based systems is an increase in unit cost and a degradation of performance (power, speed, size), which vendors mitigate by embedding into the FPGA silicon performance enhancing primitives such as memories and multipliers. Inevitably, the increasing transistor density and heterogeneity of FPGAs leads to a complexity challenge necessitating new design methodologies to increase productivity.

This paper presents a *structured system* design methodology for FPGA based system-on-a-chip development, based on a paradigm of architecture reuse; instead of designing a monolithic FPGA configuration by connecting together blocks of predesigned intellectual property (IP) the system is constructed by defining the its logical and physical structure first. Increased productivity is achieved through *(a) modularity*, *(b) abstraction*, and *(c) orthogonalisation of concerns*, such as the separation of communication and computation. Significantly, system-level timing is pre-determined avoiding iterative design and verification cycles normally necessary to achieve timing closure.

Since the methodology is based on architectural reuse, the choice of architecture is critical; the architecture must be suited to the applications which will be implemented on it as well as being well matched to the FPGA structure at which it is targeted. In this paper the methodology is exemplified with a single architectural instance described in [1] called *Sonic-on-a-Chip*.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1047–1051, 2004.

2 Related Work

A popular area of research for improving designer productivity is the investigation of direct synthesis from existing software languages, e.g. [2], in an attempt to increase the level of abstraction. While this approach has had some measure of success, it is limited by the mismatch between logic and languages designed to abstract the execution of code for a sequential processor. An alternative is to use novel algorithm description formats, in which the algorithm designer explicitly identifies task-level parallelism [3,4]. Often the representation uses a dataflow graph format; invariably some form of modularity is used. These techniques suffer from a lack of a clear macro-architecture within the target FPGAs to which the modules may be mapped. This problem can be circumvented by using a custom modular or coarse-grained reconfigurable fabric instead of a traditional FPGA; the SCORE system [5] is one such example. The work in this paper focuses on an alternative, platform based approach, in which a hardware infrastructure is designed within an FPGA to be reused by many application instances. An FPGA based platform that employs an AMBA bus has been reported by Kalte *et al.* [6]. Rather than use a general purpose bus, our work assumes an infrastructure optimised for a specific domain. Moreover, the architecture is part of a larger methodology which includes precepts for hardware, software and operating system interactions.

3 Structured System Design

In a structured system design, a complex system is built by defining an architecture and then filling it with IP. The architecture can be envisaged as an interface between the form of the application algorithm and the unstructured resources of the FPGA. The architecture includes a modular logical layer which can be customised to the application, and a physical infrastructure layer, which describes how the logical modules are implemented and connected in the FPGA fabric.

The architecture design is exemplified by Sonic-on-a-Chip [1], an embedded video processing platform targeted at the Virtex II Pro family of FPGAs from Xilinx. The logical architecture is shown in Figure 1. In terms of the design methodology, the salient and necessary features of the logical model are that it is highly modular, extensible, and application implementation is achieved through customisation of modules. Moreover, the computation and communication are separated (via routers in this case) such that the topology of the application is independent of the intermodular communication infrastructure. This is important, since the infrastructure must remain fixed between applications.

The physical layer of the architecture must define how the logical system structure is implemented on the FPGA, including the location of inter-modular interconnect and the positioning of the modules. Existing design flows (including the Modular Design flow from Xilinx) integrate the modules and infrastructure together at design time, which we term *early integration.* In our design methodology the integration point is at run-time. This *late integration* affords the

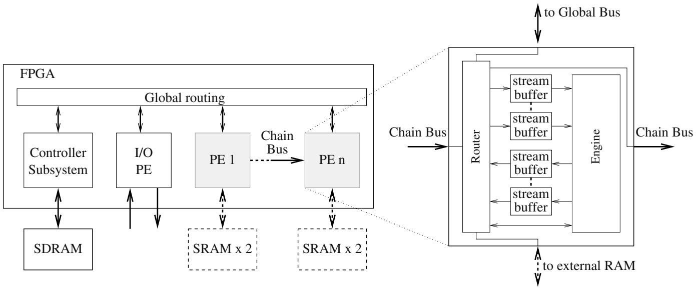


Fig. 1. General logical structure and processing element internals of Sonic-on-a-Chip.

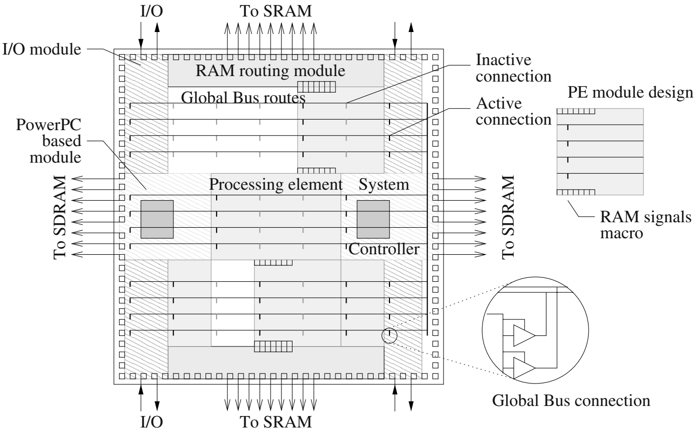


Fig. 2. Physical implementation of Sonic-on-a-Chip on a XC2VP100. Note that processing element designs must incorporate the Global Bus wiring.

maximum design productivity advantage from modularity by separating module development down to the bitstream level, since design, implementation and verification of different modules can be executed independently. In addition, different functionality can be obtained at run-time by combining fully developed modules in different permutations and combinations, allowing applications to be customised from instance to instance without re-synthesis, re-implementation and re-verification each time.

The example Sonic-on-a-Chip physical infrastructure for a Xilinx Virtex II Pro target is illustrated in Figure 2. Note that the modularity of the logical design is preserved at the physical level. In late integration the position of each module is not fixed until run-time, implying modules must be bitstream-relocatable and that both the FPGA and the infrastructure exhibit translational symmetry.

Importantly, the location of the Global Bus interconnect is fully predetermined. This is necessary for module relocatability; moreover, the overall bus timing is constant and can be characterised, enabling modules to be physically implemented independently. The interconnect and relocatability is inspired by the DISC system [7].

Application development for Sonic-on-a-Chip begins by specifying the processing algorithm to be implemented as a number of parallel tasks, connected in a dataflow graph format. Each task-node of the dataflow graph needs to be implemented in a processing element. Overall system control is handled in application software, which loads processing element module bitstreams into the configuration memory of the FPGA fabric and programs the routers within each processing element to direct dataflow appropriately.

In a stand-alone software environment, it is left to the application developer to manage the allocation of space and the positioning of modules within the fabric. In systems where two or more applications execute concurrently an operating system is usually employed to manage shared resources. In the case of Sonic-on-a-Chip, physically adjacent modules can communicate directly, so the relative positioning of modules must be considered by the operating system.

In order to represent the parallel processing and the topology of the hardware modules, in our methodology the application spawns a new software process for each hardware module. These processes are termed *ghost processes*, since they do not perform any processing but are a representation of the hardware. The communication between hardware modules is represented by redirecting the inputs and outputs of the ghost processes to named pipes (FIFOs). This provides a means to encode the logical topology of the algorithm dataflow as well as a mechanism for software to interact seamlessly with hardware entities.

4 Design Evaluation

A functional simulation model has been constructed and a prototype is under development based on the Xilinx ML300 evaluation board. The characteristics of two computational element designs is given in Table 1. The approximately 8x difference in resource usage is a simple but clear justification for the necessity of variable-sized PEs.

Table 1. Processing element characteristics (designs run at 50MHz).

Type	Slices	Block	MOPS	Throughput	Estimated no. per device	
		RAMs		Msamples/s	XC2VP30	XC2VP100
Image difference	345	3	50	50 peak	39	127
3x3 convolution	2450	32	528	59.3 peak	5	18

The physical implementation of processing element modules and RAM routing modules has been investigated using the partial reconfigurable design flow [8], which uses hard macros to ensure signals entering and exiting a reconfigurable module are always routed on the same wires. The standard bus macro in this flow is unsuitable for our implementation since it is only designed to pass signals across vertical module boundaries. New hard macro objects have been developed, including one to constrain Global Bus routing to specific tristate lines.

These investigations exposed routing issues with both the FPGA fabric and the routing tool (Xilinx PAR F.31). The FPGA interconnect is optimised for use as a global resource and therefore includes features such as global clock trees and longlines, which are not well suited to partial modular reconfiguration. In particular, horizontal longlines are not used when implementing a reconfigurable module. In addition, the place and route tool currently recognises vertical module boundaries only, and routing violations were observed for horizontal boundaries.

5 Conclusion

This paper introduced an architecture reuse based methodology for FPGA SoC design, which uses modularity and abstractions to enhance productivity. The architecture, which includes physical and logical constructs, is exemplified by the Sonic-on-a-Chip instance. Applications comprise hardware IP modules and control-level software; in an OS-based software environment the hardware processing is represented by spawned software processes connected via IPC FIFOs.

Investigations into the implementation of the methodology have exposed a dissonance between the globally optimised FPGA interconnect design and the requirements for modular reconfigurable routing; this is an area we plan to address in future work.

Acknowledgements. The authors thank the Commonwealth Scholarship Commission in the UK, the New Zealand Vice Chancellors' Committee and Xilinx Inc. for financial support.

References

1. Sedcole, N.P., Cheung, P.Y.K., Constantinides, G.A., Luk, W.: A reconfigurable platform for real-time embedded video image processing. In: Proc. Field–Programmable Logic and Applications. (2003)
2. Babb, J., Rinard, M., Moritz, C.A., Lee, W., Frank, M., Barua, R., Amarasinghe, S.: Parallelizing applications into silicon. In: Proc. IEEE Symposium on Field–Programmable Custom Computing Machines. (1999)
3. Diessel, O., Milne, G.: Hardware compiler realising concurrent processes in reconfigurable logic. IEE Proc. Computers and Digital Techniques **148** (2001) 152–162
4. Weinhardt, M., Luk, W.: Task parallel programming of reconfigurable systems. In: Proc. Field–Programmable Logic and Applications. (2001)
5. Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., DeHon, A.: Stream computations organized for reconfigurable execution (SCORE). In: Proc. Field–Programmable Logic and Applications. (2000)
6. Kalte, H., Langen, D., Vonnahme, E., Brinkmann, A., Rückert, U.: Dynamically reconfigurable system-on-programmable-chip. In: Proc. Euromicro Workshop on Parallel, Distributed and Network-based Processing. (2002)
7. Wirthlin, M.J., Hutchings, B.L.: A dynamic instruction set computer. In: Proc. IEEE Symposium on FPGAs for Custom Computing Machines. (1995)
8. Lim, D., Peattie, M.: Two flows for partial reconfiguration: module based or small bit manipulation. Application Note 290, Xilinx (2002)

Secure Logic Synthesis*

Kris Tiri[1] and Ingrid Verbauwhede[1,2]

[1] UC Los Angeles,
{tiri,ingrid}@ee.ucla.edu
[2] K.U.Leuven

Abstract. This paper describes the synthesis of dynamic differential logic to increase the resistance of FPGAs against Differential Power Analysis. Compared with an existing technique, it saves more than a factor 2 in slice utilization. Experimental results indicate that a secure version of the AES algorithm can now be implemented with a mere doubling of the slice utilization when compared with a normal non-secure single ended implementation.

1 Introduction

Side-channel attacks (SCAs) have been identified as an important open issue related to the general security of cryptographic applications on FPGAs [1]. These attacks find the secret key with information associated with the physical implementation of the device, such as time delay and power consumption. Much effort has already gone into setting up the Differential Power Analysis (DPA) on FPGAs [2].

We have previously presented a logic level design methodology to implement a secure DPA resistant crypto processor on FPGA [3]. In this manuscript, we study the synthesis aspects in order to reduce area consumption and time delay. The next section briefly introduces Wave Dynamic Differential Logic (WDDL), the cornerstone of the logic level design methodology. Section 3 discusses a technique to combine several WDDL gates into 1 slice. This reduces the area consumption and time delay. Section 4 describes the clustering procedure of the synthesis methodology. In section 5, the performance is evaluated. Finally, a conclusion is formulated.

2 Wave Dynamic Differential Logic

To address power attacks, we have introduced a family of secure compound standard cells, referred to as Wave Dynamic Differential Logic [3]. WDDL can be constructed from regular standard cells and is applicable to FPGA. WDDL achieves its resistance by charging in every cycle a constant load capacitance. It is dual rail with precharge logic in which a pre-discharge wave travels over the circuit. In the precharge phase, the inputs to the WDDL gate are set at 0. This puts the output of the gate at 0 and the precharge wave travels over to the next gate.

The set of logic gates is restricted to the WDDL AND- and OR-gates in order to assure that every compound standard cell has exactly 1 output transition per cycle [3]. In addition, it is essential for input independent power consumption that the gate

* This work was supported in part by NSF grant CCR-0098361.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1052–1056, 2004.

always charges ideally the same load capacitance. The capacitances at the differential in- and output signals are alike [4]. There is however a difference in the interconnect capacitance due to routing variations. Placing the 2 LUTs of a compound standard cell adjacent and in the same slice minimizes this effect. Then, the differential signals need to travel the same distance.

The basic building block of a Virtex-II FPGA is known as a slice and consists of two 4-input, 1-output look up tables (LUTs), some multiplexers and registers. A WDDL AND-gate (OR-gate) occupies 1 slice, in which the G-LUT functions as an and-operator (or-operator) on the true inputs, while the F-LUT functions as an or-operator (and-operator) on the false inputs.

3 Slice Compaction

Currently, 2 LUTs are used to build a compound WDDL gate. Each LUT is generates one output of the differential output pair. It is however possible to add more functionality into each LUT. A cluster formed by an arbitrary collection of G-LUTs and the cluster formed by the corresponding F-LUTs behave as a compound WDDL-gate. The 2 clusters (1) are differential; (2) transmit the precharge value; and (3) have a 100% switching factor. A LUT on the Virtex-II platform has 4 inputs and 1 output. In this case, the clusters can have at most 4 inputs and 1 output.

Fig. 1 shows an example. Fig. 1A depicts the single ended logic function to be implemented. The WDDL implementation that results from our original methodology is shown in Fig. 1B. Each gate is replaced by its corresponding WDDL gate. In total, 6 slices are occupied. The logic depth is 3. Fig. 1C shows the implementation after clustering. This implementation occupies only 2 slices and has a logical depth of 2. The clustering algorithm to obtain such compact, side-channel resistant implementations of WDDL based circuits is the topic of this paper.

4 Logic Synthesis

The kernel of DPA-proof logic synthesis is a clustering algorithm. Given a DPA-proof implementation consisting solely out of secure compound WDDL AND- and OR-gates, it partitions the design into groups of LUTs with 4 or less distinct inputs and 1 output. Each group will form together with their corresponding dual group a secure compound gate and will be mapped onto adjacent LUTs within the same slice. A group of LUTs can be divided into many partitions. Various factors, such as graph traversal order and redundancy introduction, influence the compaction. The remainder of this section describes an alternative, yet efficient clustering procedure.

4.1 Clustering Through Transformation

Fig. 1C could also have been obtained through a transformation of the synthesized single ended design, shown in Fig. 1D. It is a parallel combination of this design and its dual. To implement an arbitrary logic function however, several inversions may be present. Inversions prohibit a direct transformation.

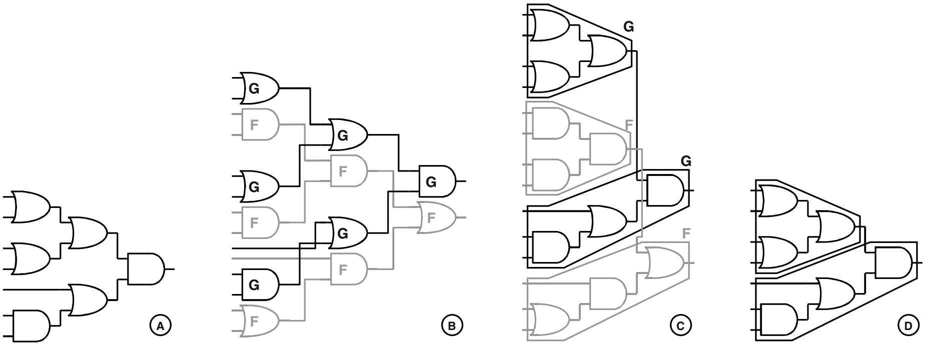


Fig. 1. Original single ended logic function (A); WDDL implementation (B); clustered WDDL implementation (C); and synthesized single ended implementation (D).

This is best seen with an example. Fig. 2A shows a logic function implemented with and2, or2, and inverter gates. The synthesized single ended implementation is shown in Fig. 2B. Note that inside one LUT, there is an inversion. This is not a good partitioning. The precharge 0 at the input of the inverter is propagated as a 1 and consequently at least 1 of the 2 dual LUTs will have a 1 at the output during the precharge phase. Hence, the precharge 0-wave is halted. The WDDL implementation obtained through to the original design methodology is shown in Fig. 2C. Here the inverters have been removed. The outputs of the secure compound gate that precedes the inverter have been exchanged and as a result there is no inverter anymore to halt the precharge wave. This procedure however, interconnects the G- and F-LUTs.

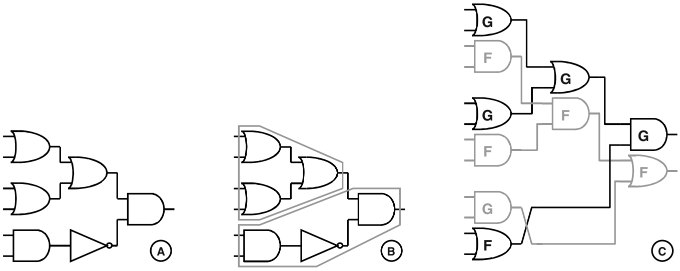


Fig. 2. Inversion mixes G-LUTs and F-LUTs: arbitrary logic function with inversion (A); synthesized single ended design (B); and original WDDL implementation (C).

4.2 Practical Design Flow

The examples of above, lead to a first design flow:

1. The design is synthesized with a limited standard cell library (and2, or2, inverter).
2. The inverters are removed from the result of step 1. The input of each inverter becomes a global output; the output of each inverter a global input.
3. The result of step 2 is synthesized for FPGA implementation.

4. Each LUT of step 3 is implemented in a G-LUT, its dual in the adjacent F-LUT. The in- and outputs created in step 2 are reconnected. The inversions are established by switching the differential connections.

A detailed discussion of this design flow is available [5].

Performing 2 synthesis procedures (in step 1 and 3) is inconvenient and seems redundant. Furthermore, the methodology is only suitable for area optimization. In step 4, the disconnected paths, which have been created in step 2 through stripping of the inverters, are connected. As a result, the delays are summed and may be larger than the critical path of step 3. In the next section, a compressed design flow is presented that ignores steps 2 and 3 and that can minimize the critical path.

4.3 Compressed Design Flow

Since a cluster formed by an arbitrary collection of and2 and or2 gates and its dual will behave as a WDDL gate, the synthesis library can be expanded to include all functions in which 4 or fewer inputs are combined with the and2 and or2 operator. Additionally, since all signals will eventually be differential, the input signals may be inverted and the output signals may be inverted. Or in other words instead of having a secure AND and OR gate, we synthesize directly with the complete selection of secure gates that can be implemented in a slice. We can now skip step 2 and 3 of the practical design flow. The resulting secure digital design flow to implement DPA resistant FPGAs is shown in Fig. 3. The gray colored blocks are the stages of the previous design flow that have been expanded or excluded.

The script transforms the single ended gates in their WDDL counterparts. Each gate declaration is replaced with a primitive module of the FPGA and the dual of this primitive module. Mapping directives are added to implement both in adjacent LUTs.

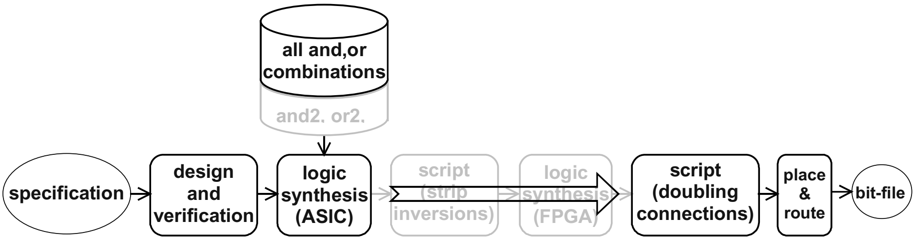


Fig. 3. Secure digital design flow for FPGAs.

5 Experimental Results

We implemented substitution-boxes of Kasumi, DES and AES. For each substitution-box, 4 designs have been implemented: (1) *original WDDL*, the result from the original AND-OR design methodology (Fig. 1B and 2C); (2) *differential*, the result from a regular insecure synthesis of the differential netlist of the original WDDL description; (3) *compacted WDDL*, the result from the compressed design flow of section 4.3 (Fig. 1C); and (4) *single ended*, the result from a synthesis of a normal insecure single ended design (Fig. 1D and 2B). The differential implementation serves as benchmark because the synthesis tool only has behavioral information and is free to map the functionality onto the LUTs. We have used DesignAnalyzer for the original and the compacted WDDL implementation and the synthesis tool in the XST Verilog design flow for the differential and the single ended implementation. The programming files have been generated for a Virtex2 xc2v1000-6bg575 with the same pin locations for each implementation. Synthesis and Place & Route have been done with the default settings of the tools.

Table 1 presents the slice utilization. On average, there is a factor 2.25 reduction between the original and the compacted WDDL implementations. There is also an important difference, up to 37%, between the benchmark implementation and the compacted WDDL implementation. The compacted WDDL designs of DES and Kasumi are on average a factor 4.42 larger than the single ended designs. The secure AES design however, only requires 1.95 times the slices of the single ended design.

Table 1. Slice utilization.

	DES								Kasumi		AES
	S1	S2	S3	S4	S5	S6	S7	S8	S7	S9	Sbox
original WDDL	138	139	137	143	136	142	137	128	249	303	797
differential	73	81	87	78	86	85	76	73	142	157	357
compacted WDDL	67	65	64	57	70	68	59	64	110	123	340
single ended	14	15	14	8	13	15	15	13	30	32	174

6 Conclusions

We have presented a design methodology to synthesize secure DPA resistant logic. Compared with the original WDDL, slice compaction offers more than a factor 2 reduction in slice utilization. The methodology seems perfect for the AES algorithm. Compared with a single ended design, the overhead in slice utilization is restricted to a factor 2. The experiments have also shown that the synthesis methodology achieves smaller utilization factor than a conventional FPGA synthesis tool.

References

1. T. Wollinger and C. Paar, "How Secure Are FPGAs in Cryptographic Applications?" in Proc. of FPL 2003, LNCS 2778, pp. 91–100, Sept. 2003.
2. B. Örs, E. Oswald and B. Preneel, "Power-Analysis Attacks on an FPGA – First Experimental Results," in Proc. of CHES 2003, LNCS 2779, pp. 35–50, Sept. 2003.
3. K. Tiri and I. Verbauwhede, "A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation" in Proc. of DATE 2004, pp. 246–251, Feb. 2004.
4. L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic Power Consumption in Virtex-II FPGA Family," in Proc. of FPGA 2002, pp. 157–164, Feb. 2002.
5. K. Tiri and I. Verbauwhede, "Synthesis of Secure FPGA Implementations," UCLA Internal report, available as report 2004/068 from the IACR Cryptology ePrint Archive, Feb. 2004.

Hardware and Software Debugging of FPGA Based Microprocessor Systems Through Debug Logic Insertion

M.G. Valderas[2], E. de la Torre[1], F. Ariza[1], and T. Riesgo[1]

[1] Universidad Politécnica de Madrid, ETSII, División de Ingeniería Electrónica
{eduardo, teresa}@upmdie.upm.es, felix.a@alum.etsii.upm.es
[2] Universidad Carlos III de Madrid. Departamento de Tecnología Electrónica
mgvalder@ing.uc3m.es

Abstract. In this paper, we present a technique and a tool to debug microprocessor systems implemented in FPGAs. We propose a method based on debug logic insertion and a set of debug modules to provide soft core microprocessors with In-Circuit Emulation capabilities.

1 Introduction

Designers have used several approaches for embedded microprocessor system design, including several combinations of logic simulators, instruction simulators, hardware emulators and in-circuit emulators [1]. Nowadays, implementing microprocessor cores in FPGAs for rapid prototyping and system implementation is becoming quite common. Methods and tools to debug designs implemented in FPGAs are needed.

FPGA vendors offer general purpose embedded logic analyzers, like Chipscope ILA from Xilinx, SignalTAP from Altera or CLAM from Actel. They also offer microprocessor soft cores especially designed to be implemented in their devices, like Xilinx MicroBlaze or Altera Nios. Modern platform FPGAs integrate embedded processors, like Xilinx Virtex-II Pro devices, that offer several PowerPC processors, or Altera Excalibur devices, with an ARM processor. All these microprocessors and cores have reasonable built in debugging capabilities.

In microprocessor systems design, debug resources are needed both for hardware and software. In this paper, we show the techniques and tools that allow system debugging in a general way, offering highly configurable event detection and tracing capabilities, and the debugging of microprocessor systems, providing microprocessors with in-circuit emulation features. The features offered range from simple event detectors and signal monitors to the most powerful and resource consuming features, like tracing, in-circuit emulation and complex event and sequence detection.

The most important properties of the presented debug features are their high configurability, so that the user can adjust to available logic resources, remote control of debug logic and expandability by means of user customized debug blocks.

The paper is structured as follows: section 2 explains the used techniques and the architecture proposed for debug logic, section 3 deals with debug modules aimed for embedded microprocessor systems debug, section 4 shows the tool for microprocessor debug, section 5 shows some experimental results and section 6 states some conclusions and future lines.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1057–1061, 2004.

2 Debug Logic Implementation: Debug Modules

Debug logic is inserted into a design in blocks called debug modules. Debug modules are logic blocks that monitor some signals from the design and process their values to produce some results. They are operated from a host computer, requiring mechanisms to send parameters and commands and to extract data from monitoring and tracing. It is important to use non intrusive mechanisms in order to be able to debug systems in real time. It is also desirable to use mechanisms already available in FPGAs. Nowadays, the most popular interface for debug is JTAG [2].

The techniques that have been considered for debugging are Boundary Register sampling, and custom JTAG user registers. Pin sampling is a mandatory feature of the JTAG standard. Signals can be monitored by routing them to unused pins and modules can use this feature to extract monitoring data. Custom JTAG user registers are shift registers built using FPGA logic. Xilinx and some Actel FPGAs allow the connection of the FPGA logic to the built-in JTAG infrastructure to build these registers. Debug modules can use these registers to capture monitoring data and to store working parameters and instructions.

Other techniques, like Readback for signal access and Partial Reconfiguration for faster module insertion and module control might improve the proposed methods [3], although they would constrain the applicability to those devices where these techniques are available.

An architecture for debug modules has been developed (Fig. 1). In general, they have some processing logic to analyze and extract information from circuit signals. It can have additional logic to control the module operation. Two shift registers are used to communicate with the host: a data register (DR) is used to capture data and send it to the host, and a control register (CR) is used to receive commands and parameters from the host. Modules can also have outputs, to allow module interconnection.

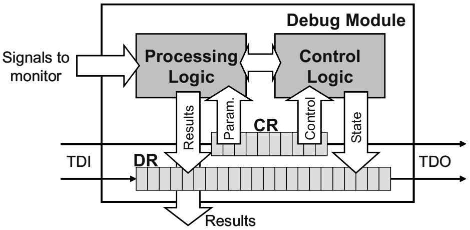


Fig. 1. Debug module general architecture

Although the JTAG standard suggests the implementation of a different shift chain for every register, we have decided to use two shift chains, because when using Xilinx devices, only two scan chains can be implemented by using the built-in JTAG infrastructure. In Actel devices there is no restriction.

Debug modules must be highly configurable through the intensive use of generics. Modules can be adjusted to fit the particular dimensions of the design, by modifying parameters such as bus widths or counter lengths. Some parameters can also be changed to restrict resource usage; the user can specify the inclusion or deactivation of additional features or he can establish the specific resources to be allocated (like the amount of tracing memory to be used, for example).

3 Microprocessor Oriented Debug Modules

Although some general purpose debug modules have been developed, only microprocessor related modules will be discussed. We want to provide any embedded microprocessor with features like event detection and logging, breakpoints and watchpoints, tracing and code performance analysis. To implement these features, several debug modules have been developed: a Clock Controller, which allows to stop the system and perform single-step and multi-step execution, a Logic Analyzer, and an In-Circuit Emulator macrocell.

The Logic Analyzer is a data acquisition module intended to capture address and data bus values at system clock speed and store them in FPGA embedded memory. Users can choose the amount of memory allocated. Trace memory can be dynamically reorganized to trace both buses, or only one of them with a higher number of samples.

The In-Circuit Emulator macrocell is a very complex and highly configurable event detector for buses and single signals. The ICE is connected to the microprocessor address and data buses, and to some control signals (Fig. 2).

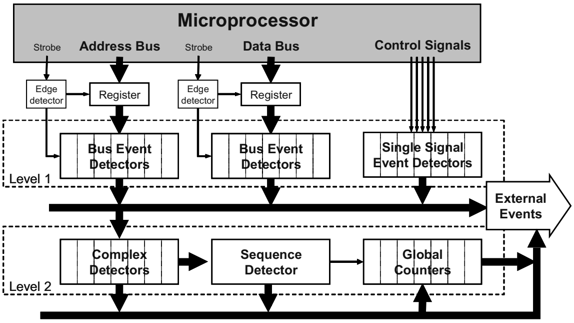


Fig. 2. ICE architecture

The ICE consists of a user-configurable set of event detector blocks. The blocks are organized in two levels: first level detectors monitor microprocessor signals or buses directly and second level detectors monitor other detectors to make more complex processing.

At the first level, there are blocks that can detect events on the microprocessor signals. These detectors can be divided into single signal detectors and bus detectors. Single signal event detectors are activated with given signal logic values or edges.

Bus event detectors check conditions in buses. There are three different detectors for buses: simple detectors, detectors with mask and ranged detectors. *Simple Detectors* check a condition between the bus value and a given value (equal, different, greater or lower). *Detectors with Mask* check the bus value against a given value and mask. *Ranged Detectors* check the bus value position in relation to a given range (inside, outside, above or below the range).

Detectors have three different versions: direct detector, detector with counter and delayed detector. The direct detector is the detector in the basic form. The event detector with counter has a counter to record the number of times the event has taken place. In the delayed event detector, the basic event has to take place a selectable number of times to be considered an event.

Second level blocks use event information generated by other blocks to make more flexible event detectors or to count events. There are three possible blocks of this kind. *Complex Event Detectors* check the occurrence of a logical condition (AND, OR) of a selectable set of level 1 detectors. A *Sequence Detector* checks the occurrence of some complex events in a sequence. *Global Counters* allow counting the occurrence of complex events and sequences.

With this set of detectors, debug microprocessor features can be implemented: breakpoints and watchpoints are implemented combining the clock controller and bus detectors in the address and data bus; performance analysis is performed using ranged bus detectors in the address bus; tracing is performed using the logic analyzer module triggered by a complex event detector or a sequence detector.

4 Tools

Some tools have been developed to aid the user in debug related tasks. The CHDT (Configuration Handling and Debug Tool) [4] has been designed to help in debug tasks: module selection, configuration and interconnection, automatic VHDL modification, synthesis and FPGA programming, and debug logic operation.

The debug methodology allows the design of *module controllers*, software tools designed to operate debug modules. Module controllers can remotely control debug modules by sending commands and receiving monitoring information trough the CHDT in a client-server TCP/IP architecture. The CHDT is attached to the hardware using the JTAG interface and acts as a server for debug module controllers.

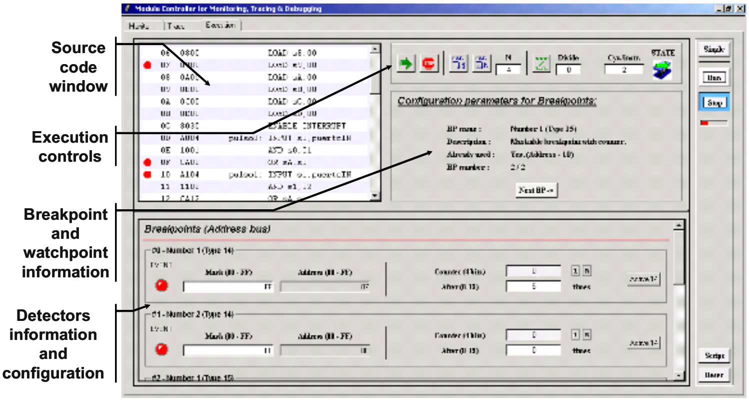

Source code window
Execution controls
Breakpoint and watchpoint information
Detectors information and configuration
Module Controller for Monitoring, Tracing & Debugging
Configuration parameters for Breakpoints:
Next BP -->
Breakpoints (Address bus)
STATE
Single
Run
Stop
Script

Fig. 3. ICE tool: execution control window

A module controller has been designed for microprocessor debugging (Fig. 3). Among other features, it can arrange the different detectors of the ICE, the Clock Controller and the Logic Analyzer to implement software debugging features at assembler code level: breakpoints, watchpoints, profiling and tracing.

5 Experimental Results

The user can choose the amount of debug logic to be inserted in the system. The possibilities range from simple Signal Monitors, which do not use additional logic, to the debug features presented in this paper. As an example, a microprocessor system has been built and implemented in a board with a Virtex-200E. Debug features have been added to a PicoBlaze, an 8-bit soft core microprocessor provided by Xilinx [5].

The ICE macrocell has been attached to the program memory, and has been configured to implement 6 single signal detectors, 6 bus detectors with mask for the address bus, 4 simple event bus detectors for the data bus, and 2 complex event detectors. With this configuration, 6 breakpoints and 2 watchpoints can be implemented. The Logic Analyzer has been attached to trace the address and data bus, and has been configured to use all available RAM in the device. It can store 4K samples if both buses are traced, or 12K samples if only the address bus is traced.

This debug configuration has supposed an increment in FPGA resource usage of 845 flip-flops, 1418 LUTs and 24 RAM blocks. In this case, as the FPGA still has unused resources, more debug features could yet be implemented. Performance could also be affected by debug logic insertion. In this case, critical path delay has just increased from 23.5ns to 24.7ns, which is not a significant change.

6 Conclusions and Future Work

In this paper, we have presented some techniques, modules and tools that allow the debugging of microprocessor systems implemented in FPGAs. Debugging capabilities are provided by the insertion of debug modules into circuit descriptions, and taking advantage of unused FPGA logic resources.

The presented approach offers some advantages over existing solutions. The methodology allows the usage of a wide range of debugging features, including complex event detection, tracing and in-circuit emulation for microprocessors. The debug methodology is expandable, allowing an easy integration of new debug features. The presented debug modules also offer a highly configurable to allow the user to establish an appropriate trade-off between the needed features and the available logic resources.

The presented ICE can be used as a standard source code debugger, implementing breakpoints and watchpoints, and also as a non-intrusive debugging tool, being able to detect and take account of complex event occurrence.

Among the future lines, there is an ongoing work to add support for readback and partial reconfiguration. It might result into an area overhead reduction and an increase of the overall observability.

References

1. R. Turner, “System-level verification - a comparison of approaches”, RSP, Florida, 1999.
2. IEEE Standard 1149.1-1993, “IEEE Standard Test Access Port and Boundary Scan Architecture”, IEEE Standards Board, 1993.
3. M. A. Aguirre, J. N. Tombs, A. J. Torralba, L. García Franquelo, “UNSHADES-1: An Advanced Tool for In-System Run-Time Hardware Debugging”. FPL, Lisbon, 2003.
4. E. de la Torre, T. Riesgo, J. Uceda, E. Macip, M. Rizzi, "Highly configurable control boards: a tool and a design experience", RSP, Paris, 2000
5. Ken Chapman, “PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices”, Xilinx Application Note, XAPP213 (v2.1) 2003.

The Implementation of a FPGA Hardware Debugger System with Minimal System Overhead

J. Tombs[1], M.A. Aguirre Echanóve[1], F. Muñoz[1], V. Baena[1], A. Torralba[1],
A. Fernandez-León[2] and F. Tortosa[2]

[1] Escuela de Ingenieros, University of Seville, Camino de los descubrimientos s/n. Spain.
jon@gte.esi.us.es
[2] ESA/ESTEC Noordwijk ZH, The Netherlands

Abstract. FPGAs provide powerful hardware emulation platforms for rapid prototyping of digital designs. This potential is usually restricted to overall system level emulation, with interactive debugging possibilities limited to the real-time observation of external signals. This article describes the most recent advances made in the UNSHADES[1] system, where unlike most commercial packages, signals need not be previously selected, nor are limited in number or size by the internal memory available. This system, which adds a small debug controller to the design to be inspected, provides many new design debugging features such as single stepping, state modification or register inspection over the entire design. The debug controller provides these powerful debugging operations without the need for large design modification whilst occupying itself very little FPGA resources. A minimal debug controller implemented in a virtex-II FPGA requires the occupation of just 3 IO pins and 43 logic slices; over half of these logic slices are dedicated to an optional 32-bit cycle counter.

1 Introduction

As digital designs grow larger, and FPGA prototyping become more common, current design debugging concepts often become ineffective. Digital design debugging is heavily based on user created test benches and software logic simulators. Any error, misconception or idealization of the external stimulation will often be found in both the design under test and the user written simulation test bench. For these reasons, the FPGA foundries have sponsored the creation of in-circuit debuggers. An example of which is the Xilinx Inc ChipScope™[2] or Altera Inc SignalTap ™[3] packages. These packages with their associated software allow the insertion of a simple logic analyzer IP inside the FPGA. This IP uses the internal FPGA memory to record internal signals, and later transfer the results to the user. It has the advantage that the user design does not need to be manually modified, but competes for the limited FPGA resources. The presence of the signal probes in the final design netlist can reduce the design routability, lower allowable clock speeds, and possibly mask or create subtle design timing problems that are only present when the analyzer is not used. A full study of these effects can be found in [4].

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1062–1066, 2004.

The UNSHADES system [1] provides a complementary debugging tool. By using the built-in Xilinx Capture macros and the dedicated configuration port it does not require that the design is modified in order to monitor internal system state. Full FPGA state recording is performed in-situ and only an insignificant amount of FPGA resources are required. The disadvantage of the UNSHADES-1 approach is that in order to provide cycle by cycle analysis, the design must be "frozen" every time the state is captured or modified and only "released" when the necessary state information has been successfully transferred to the external debugger software. Although individual clock cycles are run at full clock rate, this stop-start nature of the design clock may cause the design to loose synchronization with the external stimulus.

The system consist of a slightly modified version of the design to be inspected, and the custom UNSHADES debug controller. The debug controller [5], consists of a very small finite state machine and the necessary logic to communicate the debugger state and activate the Xilinx Capture macro when an inspection is to be made.

2 The UNSHADES System

The basic functioning of the UNSHADES system is based on the concept of debugging events. Debugging event conditions are added by the user to the HDL source code to be debugged. The event conditions are concurrent VHDL expressions that are true when the condition is true. Together the user design and controller constitute the complete UNSHADES debuggable design. When an event is activated, the design under inspection is frozen, the CAPTURE macro is stimulated and the entire FPGA register state can then be read from the SelectMap configuration port. Partial reconfiguration and reading allows subsets of the state information to be read if full state information is not requested. While the design is frozen a read-modify-write-write sequence of partial configurations enables the free modification of the value of any FPGA Flip-Flop. A 3-pin IO interface provides progress indication, single cycle stepping, asynchronous state capture and clock resumption functions.

For the system to work, all synchronous elements of both the controller and design must use the same clock, and the design to be inspected must be modified to use a single, asynchronous clock enable signal. This global clock enable signal is controlled by the debug controller. In many cases this global signal was the limiting factor in design operation speed, and has the inconvenience that all IPs used within the design must either come ready with a clock enable input or provide RTL source so that the signal can be added, this is often not the case.

3 Improving the UNSHADES System: Clock Control

In principal the easiest method for freezing the synchronous elements of the design under inspection is to use a gated clock. Such techniques have been historically unsuitable for FPGA due to various problems. The FPGAs provide special low skew buffers and routing for the clock signal, any attempt to perform the logical AND with the clock and the gate control would be incompatible with this dedicated resource and produce unusable high skew clock networks. Additional care must also be taken to guarantee that runt pulses or glitches clocks are never generated. These problems have been completely removed in the newer Xilinx FPGAs. In the Virtex-II and later devices, a special clock buffer macro has been added. This cell, the BUFGMUX contains all the necessary circuitry to guarantee glitch free switching between two independent clock sources. If one of these clocks is a constant logic value, then this cell can be used as a low skew global clock buffer with dedicated clock enable.

Figure 1(a) shows the implementation of a gated clock that can be used in the newer Xilinx FPGAs. The input I0 of the buffer is tied low thus when the select signal CE selects input I0, the buffer will maintain the output clock signal low as soon as the clock input is low. In earlier devices, this buffer did not exist, and the alternative structure of figure 1(b) can be used. The explicit instancing of a global clock buffer (BUFG) produces a low skew clock network, whilst a flip-flop is used to avoid glitches. An obvious consequence of this is that the resultant gated clock is at half the debug controller clock frequency.

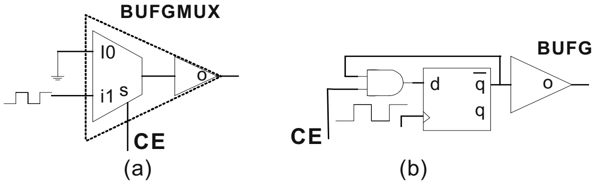


Fig. 1. Xilinx FPGA gated clock buffer configurations: (a) The Virtex2 BUFGMUX implementation, (b) Half speed implementation for other Xilinx devices.

It should be realized, that although both techniques produce correctly controlled clocks, the gated-clock will have an unknown phase relationship with the debug controller clock. Clock compensators (DDLs or DCMs) can not be used, as they will lose synchronization when the clock is gated. This adds a serious complication to the debugger control. The debug controller must halt the clock whenever an event is true before the next clock edge, but it can not make decisions using its own clock due to the unknown phase difference. The existence of debug event is possibly just a temporary glitch produced by timing delays that will have passed before the next design clock edge. Therefore while the inspected design is running the debugging events themselves control the gated clock, the debug controller passes to the stopped state once it detects that the design clock has stopped. With this configuration, except in the unlikely case that the debug event generator causes the critical delay, the design under inspection will be able to operate at its normal design frequency. With the implementation of figure 1(b) the case is somewhat worse. The clock signal is generated from a debug controller flip-flop within an FPGA logic block and the clock signal must use general routing resources to arrive at the BUFG clock buffer tree. In this case the input clock frequency must be chosen such that the time between the delayed gated clock edge and the next debug controller clock edge is equal to the desired working clock period. This may not be possible, and timing related debugging will be somewhat more limited.

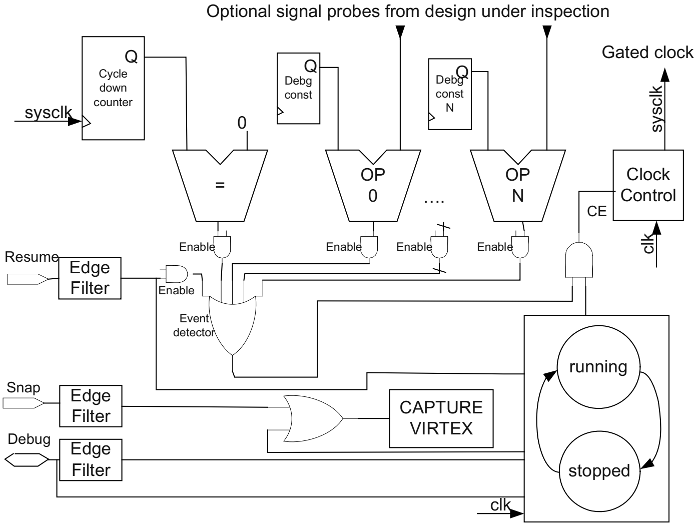


Fig. 2. Simplified structure of the debug controller. All inputs to the event detector also have an associated enable/disable memory bit not shown in the figure.

4 Improving the UNSHADES System: Event Definition

Figure 2 show the basic event capture structure of the debug controller. The debug controller consists of two states (stopped and running), input filters, clock control and event detectors. Two basic event detectors are provided that are independent of the design under inspection. These are the “cycle counter event”, and the “resume event”. The cycle counter event is a 32 bit counter using the gated clock. When the counter reaches zero a “counter event” is produced. A “resume event” is produced by a rising edge of a dedicated input pin. The resume event stops the clock while the design is running or resumes the clock when stopped. These two events are all that is required for the creation of a minimal debug controller. A completely unmodified design can be placed under UNSAHDES inspection by simply including it with the controller. Certain areas have been identified where such implementation is advantageous. The design is completely unmodified and critical timings or input related problems are not modified by the debugger. The design can be used "in-system" and in the moment that a problem is observed, the debugger can be invoked by a “resume event”. Once invoked, the user has complete single stepping and signal observation/modification access to the design, and thus the area where the fault lies can be quickly identified. The cycle counter allows cycle accurate periodic state inspection. It can be loaded with a cycle number where inspection will begin or resume.

For more powerful debugging, the user must provide a list of event conditions. The user must specify internal signal conditions that flag a moment that could be of interesting for the debugging of the design. Rather than using static conditions, each debug event “dbg_eventN” is implemented as:

<<dbg_eventN <= '1' WHEN (UserSignal OP dbg_constantN)   ELSE '0'; >>

Where dbg_constantN is an internal registe, and OP a user selected operation. All debugging constant registers are held within the debug controller, and no internal logic can modify their value. As the "constants" are stored in flip-flops, the debugger can now freely modify the event condition register value using the general UNSHADES register modification option.

5 Experimental Results and Conclusions

The debug controller has been implemented and tested in SpartanII, VirtexE devices using the flip-flop based gated clock, and in the Virtex-II devices using the BUFGMUX cell. The debugger has allowed the detection of register overflow and state machine errors in a variety of situations [6].

The ability of the debugger to stop a design in an arbitrary cycle and modify registers contents is currently being exploited for the simulation of radiation produced bit upsets (Single Event Upsets) of digital designs for space applications, under the ESA/ESTEC sponsored project FT-UNSHADES.

The controller is very small; only 20 to 43 FPGA slices (depending on the cycle counter size). In Virtex-II devices with the BUFGMUX cell available, the debugger has negligible effect on the speed and performance of the design under inspection. The use of flip-flop registers to hold the constants that form part of debugging events conditions enable "run time" modification of the all conditions via partial reconfiguration.

Future work will study this problem so that the debugger insertion can be made transparent to the design to be debugged and allow debugging of FPGA internal RAMs.

References

1. Aguirre, M.A, Tombs, J., Torralba, A., Franquelo, L.G. "UNSHADES-1: An advanced tool for In-System Run-Time Hardware Debugging" LNCS, Proc. FPL 2003, Lisboa, Portugal, September 2003
2. ChipScope Pro Software and Cores user manual. Xilinx. UG029. May 2003.
3. Design Verification Using the SignalTap II Embedded Logic Analyzer. Application note 280. Altera. June 2003.
4. Graham P "Logical Hardware Debuggers for FPGA-Based Systems", PHD Dissertation. Bringham Young University, 2001.
5. Aguirre, M.A, Tombs, J., Torralba, A., Franquelo, L.G "Method for the functional test of large digital circuits using hardware emulators", International Patent W ES-02/00571 pending. 2002.
6. Aguirre, M.A, Tombs, J., Baena-Lecuyer, V. Mora, J.L., Carrasco, J.M., Torralba, A., Franquelo, L.G "Microprocessor and FPGA interfaces for in-system co-debugging in field programmable hybrid systems", accepted for publication in "Microprocessors and Microsystems". Elsevier 2004.

Optimization of Testability of Sequential Circuits Implemented in FPGAs with Embedded Memory[*]

Andrzej Krasniewski

Warsaw University of Technology, Institute of Telecommunications
Nowowiejska 15/19, 00-665 Warsaw, Poland
andrzej@tele.pw.edu.pl

Abstract. A method for testability-oriented optimization of sequential circuits implemented using FPGAs with embedded memory is presented. It specifies the content of those memory words which are not defined by the conventional FSM synthesis. The experimental results confirm its effectiveness; for the largest examined circuit, the self-test session required to achieve an acceptable level of fault escapes for the optimized design, obtained using the proposed procedure, is almost 10^6 times shorter than for the non-optimized design. The proposed method does not involve any extra circuitry or speed degradation. Also, it does not require any extra reconfiguration during self-testing.

1 Introduction

Modern FPGAs contain embedded memory. Embedded memory, in its ROM-like mode of operation (with its content determined at the time of programming) can be used to implement complex combinational or sequential circuits. In this paper, we focus on implementations of finite state machines (FSMs).

A simple implementation of an FSM, in which a memory module contains the entire combinational part of the circuit, is shown in Fig. 1(a). To reduce the size of the memory block, an alternative implementation, shown in Fig. 1(b), that includes an extra module, called *address modifier*, has been proposed [1]. Dedicated synthesis methods have been developed for the structure of Fig. 1(b) [2-4].

When the structures of Fig. 1 are implemented using FPGAs, the memory module is normally mapped to embedded memory blocks, whereas the address modifier is implemented with LUTs included in programmable logic blocks. The address register can be implemented either with flip-flops included in programmable logic blocks or as an internal register of the embedded memory array located at its input.

To thoroughly exercise the FPGA configuration corresponding to the specific user-defined application, application-dependent testing [5], also referred to as application-oriented test [6] or configuration-dependent testing [7], is performed. A number of techniques have been proposed for application-dependent testing (usually, self-testing) of sequential circuits implemented with LUT-based FPGAs [5, 7-10]. The

[*] This work was supported by the State Committee for Scientific Research of Poland under grant no. 4 T11D 014 24

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1067–1072, 2004.

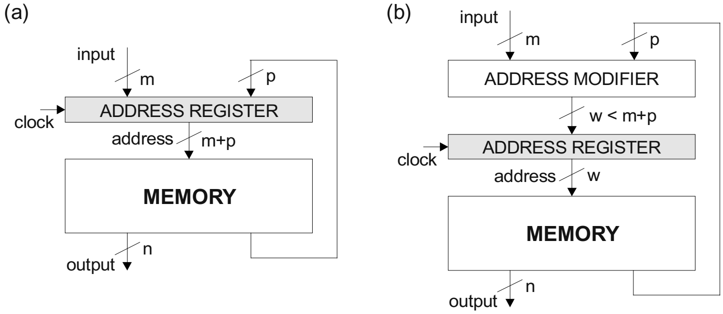


Fig. 1. Memory-based implementation of a finite state machine (FSM)

methods presented in [7-10] rely on a modification of the user-defined functions of LUTs (without changing the LUT interconnection structure) and, therefore, are not suitable for the memory-based FSM structures of Fig. 1. An analysis of the applicability of the self-test technique presented in [5] to FSMs implemented using embedded memory of FPGAs shows that a modified structure, in which an internal register of the embedded memory array located at its output serves as the FSM state/output register, is more suitable for testing than alternative FSM structures [11]. Therefore, in our study, we assume the BIST scheme shown in Fig. 2.

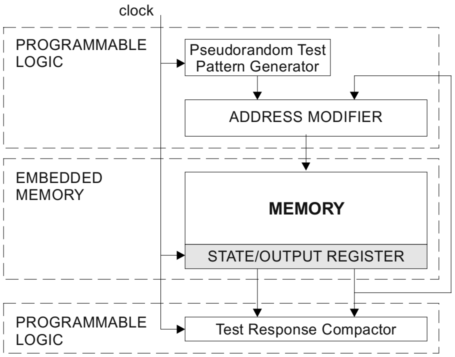


Fig. 2. Self-testing of an FSM implemented using an FPGA with embedded memory

2 Extension of Memory Specification

The method presented in this paper is intended to optimize the testability of an FSM defined by an incompletely specified next state function, that is implemented using an FPGA with embedded memory and is tested using the scheme of Fig. 2. The optimization relies on an appropriate specification of the content of those memory words which are left undefined by the conventional FSM synthesis procedure. This process is referred to as *extension of memory specification* or, simply, *memory* *extension*. As the proposed optimization does not involve the FSM output function, we assume that the output of the FSM is represented by its current state.

We assume the *single transition fault model* [12]. In this model, a fault means an incorrect transition to the next state. Thus, each fault f is defined by a triple (s, v, s*), where s is the current state and v is the input vector that causes a faulty transition to state s* (s* denotes any state different from that defined by the next state function). Thus, the number of faults is equal to the number of pairs (s, v) for which the next state function is specified. With the assumed trivial output function, pair (s, v) represents a test (the only one) for fault f.

In a self-testing environment, such as that of Fig. 2, assuming that a random test sequence of a given length is applied, the circuit testability can be measured by the *expected fault coverage* or by the *escape probability*, i.e. the probability that at least one fault from the assumed set of faults will be left undetected [13]. Alternatively, the circuit testability can be measured by the length of test (random test sequence) required to obtain a specified level of fault coverage or escape probability.

As in our analysis we focus on the improvement of the susceptibility of the circuit to random testing, the impact of test response compaction is disregarded. Then, the values of the testability measures are determined by *detection probabilities* of faults included in the fault model, especially faults that are most difficult to detect [12, 13].

In the FSM implementations of Fig. 1, with the trivial output function, the memory contains state codes - the values of the next state function. Thus, any extension of the memory specification corresponds to some *extension of the next state function*.

To optimize the circuit testability (to maximize the detection probability of the most-difficult-to-detect faults), the next state function should be extended so its values are as uniformly distributed over the set of states as possible.

For the simple memory-based implementation of the FSM (without an address modifier), the procedure that extends the next state function NSF, so that to satisfy this objective is quite trivial. For an FSM implementation with an address modifier, the procedure is more complex because, in general, several state-input combinations (for which NSF has the same value) may correspond to a single memory word. This correspondence is defined by the *mapping function* mapf: (s,v) → addr, which is determined by the function of the address modifier. The proposed procedure is

```
procedure NSF_extension(NSF,memory)

derive the mapping function mapf;
until all values of NSF are defined do
   find a memory word whose content is not defined and such that its
   address, addr, occurs most frequently as the value of mapf;
   find state s* that occurs least frequently as a value of NSF;
   for each (s,v) such that NSF(s,v) is not defined and mapf(s,v) =
   addr, assign NSF(s,v) = s*;
```

3 Experimental Results

To demonstrate the effectiveness of the proposed method, we examine several benchmark FSMs with an incompletely specified next state function. The output functions of the benchmarks are ignored. As all the examined FSMs fit into an FPGA with embedded memory blocks of 4K bits (available in Altera APEX II devices and Xilinx Virtex, Virtex-E and Spartan II devices), we consider the simple memory-based implementation (without an address modifier).

The results of our experiments are shown in Table 1. For each FSM, we compare an optimized design, with memory extension (in the first subrow), and a non-optimized design, with all the memory words whose content is not specified in the original FSM description filled with the same value (in the second subrow). The testability measures are calculated for a random test sequence applied to the circuit input, using a procedure developed based on theoretical results presented in [12, 13].

The following explanations should be made regarding the data shown in Table 1:

- the *incompletion factor* is the fraction of state-input combinations for which the value of the next state function (in the original FSM specification) is not defined;
- DP_{wf} is the detection probability of the worst fault (most-difficult-to-detect fault), i.e. the probability that such a fault is detected at a given clock cycle;
- L_{wf} is the length of a random test sequence (length of test) required to detect the worst fault with probability no lower than δ (required to assure an escape probability no larger than ε, $\varepsilon = 1 - \delta$), calculated for $\delta = 0.999$;
- FC is the expected fault coverage, calculated for L_{wf} - the length of test required to detect the worst fault in the optimal design with probability no lower than 0.999;
- L_L and L_U are the lower and upper bound on the length of test required to achieve an escape probability not larger than $\varepsilon = 0.001$ (for circuit *beecount*, L_L and L_U cannot be effectively calculated using formulas given in [12] because assumptions underlying the derivation of these formulas are not satisfied); for all the examined circuits $L_L = L_U$;
- L_{nopt}/L_{opt} shows how many times the length of test required to achieve the escape probability no larger than $\varepsilon = 0.001$ (given by L_L or L_U) is lower in the optimized design than in the non-optimized design.

Table 1. Testability measures for optimal and non-optimal memory extension

circuit	no. of inputs	no. of states	incomp. factor	DP_{wf}	faults with DP_{wf}	L_{wf} for δ=0.999	FC(L_{wf})	$L_L$and L_U for ε=0.001	L_{nopt}/L_{opt} for ε=0.001
train4	2	4	0.125	$6.25 \cdot 10^{-2}$	14	108	99.91%	148	2.1
				$2.50 \cdot 10^{-2}$	3	273	98.58%	317	
train11	2	11	0.432	$1.47 \cdot 10^{-2}$	11	467	99.96%	629	3.5
				$4.10 \cdot 10^{-3}$	8	1682	95.20%	2189	
lion9	2	9	0.306	$2.78 \cdot 10^{-2}$	25	246	99.90%	360	353.0
				$5.98 \cdot 10^{-5}$	2	115495	54.54%	127084	
beecount	3	7	0.089	$2.42 \cdot 10^{-3}$	7	2849	99.95%	?	?
				$2.83 \cdot 10^{-4}$	14	24368	87.66%	?	
s8	4	5	0.750	$1.25 \cdot 10^{-2}$	20	550	99.90%	788	1998.1
				$5.27 \cdot 10^{-6}$	4	1311310	41.49%	1574473	
tma	7	20	0.730	$3.91 \cdot 10^{-4}$	692	17681	99.90%	26551	$9.6 \cdot 10^{5}$
				$4.08 \cdot 10^{-10}$	32	$1.7 \cdot 10^{10}$	< 5.00%	$2.6 \cdot 10^{10}$	

The results in Table 1 show significant differences between the optimized and non-optimized designs with regard to testability characteristics. For example:

- for circuit *tma*, the test session required to achieve the assumed level of test quality for the optimal design is almost 10^6 times shorter than for the non-optimal design;
- for the optimal design of circuit *s8*, a sequence of 550 random patterns is sufficient to obtain the fault coverage of 99.9%, whereas for the non-optimal design only 41.5% of faults are detected by the same test sequence.

It can be seen that the presented approach is very effective even for FSMs with a small number of undefined values of the next state function. An optimal extension of the next state function of circuit *beecount*, which is undefined for only 5 state-input combinations, results in an 8-fold reduction of the test length required to achieve an assumed level of test quality.

The presented experimental results demonstrate both the need for an appropriate memory extension (for some circuits, e.g. circuit *tma*, without such an extension no acceptable level of test quality can be achieved) and the effectiveness of the proposed procedure.

The proposed optimization of the circuit testability does not involve any extra circuitry or speed degradation. Unlike other techniques for application-dependent FPGA testing, that rely on multiple reconfigurations of the user-defined circuit in the course of the test procedure [7-10], the method presented in this paper does not require any reconfiguration of this type. The only cost associated with the proposed method is an extra design effort which - because of the computational simplicity of the proposed memory extension procedure - is negligible.

References

1. Luba, T., Gorski, K., Wronski, L.B.: ROM-based Finite State Machines with PLA address modifiers. In: Proc. European Design Automation Conf. (1992) 272-277
2. Cong, J., Yan, K.: Synthesis for FPGAs with Embedded Memory Blocks. In: Proc. Int. Symp. on FPGAs (2000) 75-82
3. Wilton, S.J.E.: Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays. IEEE Trans. on CAD, **19** (2000) 56-68
4. Rawski, M., Luba, T.: FSM Implementation in Embedded Memory Blocks Using Concept of Decomposition. In: Ciazynski W. et al. (eds.), Programmable Devices and Systems, Pergamon - Elsevier Science (2002) 291-296
5. Krasniewski, A.: Design for Application-Dependent Testability of FPGAs. In: Proc. Int'l Workshop on Logic and Architecture Synthesis (1997) 245-254
6. Renovell, M.: Some Aspects of the Test Generation Problem for an Application-Oriented Test of SRAM-Based FPGAs. Journal of Circuits, Systems, and Computers, **12** (2003) 143-158
7. Quddus, W., Jas, A., Touba, N.A.: Configuration Self-Test in FPGA-Based Reconfigurable Systems. In: Proc. ISCAS'99 (1999) 97-100
8. Harris, I.G., Menon, P.R., Tessier, R.: BIST-Based Delay Path Testing in FPGA Architectures. In: Proc. IEEE Int. Test Conf. (2001) 932-938
9. Krasniewski, A.: Exploiting Reconfigurability for Effective Testing of Delay Faults in Sequential Subcircuits of LUT-Based FPGAs. In: Glesner, M., Zipf, P., Renovell, M. (eds.), Proc. FPL 2002, LNCS, vol. 2438, Springer Verlag (2002) 616-626

10. Tahoori, M.B., McCluskey, E.J., Renovell, M., Faure, P.: A Multi-Configuration Strategy for Application Dependent Testing of FPGAs. In: Proc. VLSI Test Symp. (2004)
11. Krasniewski, A.: Self-Testing of Sequential Circuits Designed for Implementation in FPGAs with Embedded Memory Blocks. In: Proc. IEEE Design and Diagnostics of Electronics Circuits and Systems Workshop (2004) 75-82
12. David, R.: Random Testing of Digital Circuits: Theory and Applications. Marcel Dekker, Inc. (1998) 135-191
13. Bardell, P.H., McAnney, W.H., Savir, J.: Built-In Test for VLSI: Pseudorandom Techniques. J. Wiley & Sons, Inc. (1987) 177-217

FPGA Implementation of a Neuromimetic Cochlea for a Bionic Bat Head

Chris Clarke[1], Lin Qiang[1], Herbert Peremans[2], and Álvaro Hernández[3]

[1] University of Bath, Department of Electronic and Electrical Engineering,
Bath, BA2 7AY, United Kingdom
{C.T.Clarke, L.Qiang}@bath.ac.uk
[2] University Antwerpen, Department of Environment and Technology
Management, Prinsstraat 13, B-2000 Antwerpen 1, Belgium
herbert.peremans@ua.ac.be
[3] University of Alcalá, Department of Electronics,
28806 Alcalá of Henares (Madrid), Spain
alvaro@depeca.uah.es

Abstract. The implementation of a neuromimetic bat model is presented in this paper. In particular the use of an FPGA to perform high performance multi-channel frequency filtering is described. Since many applications in auditory modelling require implementation of filter banks that model the characteristics of psychophysical signals, the paper focuses on an FPGA implementation of the digital filters in the neuromimetic cochlear. The paper presents an efficient hardware realization of a number of IIR Butterworth bandpass filters. The pipelined parallel filter architecture is implemented on a single Xilinx Virtex II FPGA chip running in real time.

1 Introduction

Bats exhibit navigation and prey-capture skills that when duplicated in a robot would be the envy of any robotics engineer. This is all the more impressive if one realizes that all the neural computations needed to sustain this behaviour occur within a brain the size of a large pearl [1]. In order to apply insights gleaned from the study of bat biosonar to the improvement of manmade sonar systems, it is necessary to investigate more closely the biosonar tasks, e.g., prey capture and navigation, routinely executed by bats. To facilitate the study of these tasks, we are building the CIRCE bat head [2], a robotic system which reproduces, at a functional level, the echolocation system of bats. This bionic bat head makes it possible to systematically investigate how the world is not just perceived but actively explored by bats. In this paper a model and implementation of one component of the CIRCE head are concentrated on: the neuromimetic cochlea as implemented with FPGA technology.

A simplified model of the bat cochlear as shown in Fig.1, is a bandpass filterbank with subsequent demodulation in each frequency channel by a combination of half-wave rectification and lowpass filtering. This is followed by an optional automatic gain control step implemented as a normalization prior to spike generation. The analog signals are converted into a spike code by thresholding them. This

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1073–1075, 2004.

quantitatively correct representation of the auditory nerve, allows us to study code properties at a neural population level. The following section presents an efficient filter bank implementation of the neuromimetic cochlea on FPGA.

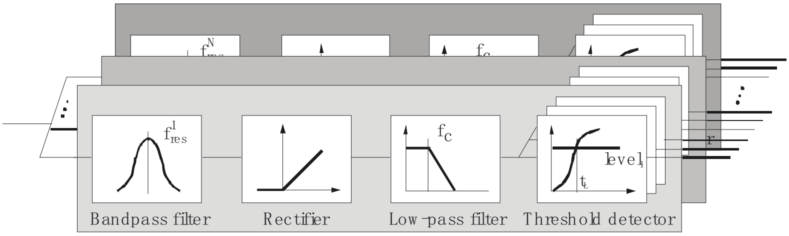


Fig. 1. A simplified model of the cochlear processing

2 Hardware Implementation of the Filters

As shown Fig.1, the cochlear mechanical filtering of the basilar membrane forms the first stage in the model. We used 2nd order IIR Butterworth bandpasss filters to undertake the cochlear filtering. The equation is given by

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) - a_1 y(n-1) - a_2 y(n-2) \tag{1}$$

As shown the equation (1), a single butterworth filter can be simplified into 3 multiplications and 3 subtractions, due to unique coefficient properties. The pipelined operation is illustrated in Fig.2(a).Each column represents one clock cycle.X memory

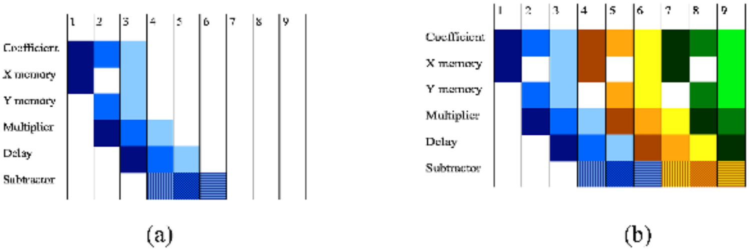


Fig. 2. Operational process of digital filter processing

represents input variables stored and Y memory represents output variables stored. As shown in Fig.2a, the input sample x_0 in X memory and the coefficient b_0 are read in the first cycle, and then x_0 is multiplied by b_0 in the second cycle. y_1 and a_1 are also read from the memories during cycle 2. In the third cycle, four operations take place. (1) y_2 and a_2 are read from memory.(2) The product b_0x_0 is stored in the x memory at the same time that the double delayed version b_0x_2 is read out of memory. (3) y_1 and a_1 are multiplied together. (4) The product b_0x_0 is delayed for one cycle. In the fourth clock cycle, three operations are performed. (1) y_2 and a_2 are multiplied together. (2) a_1y_1 is delayed one cycle. (3) The subtractor performs b_0x_0 - b_0x_2. Due to the inversion equivalence of b_0 and b_2. This is equivalent to $b_0x_0 + b_0x_2$. In the fifth and sixth cycles, the delayed a_1y_1 and a_2y_2 are subtracted from this result. The output appears in the seventh cycle and is written back to the Y memory during a spare cycle in the later filter operation. Fig. 2b shows how the resource usage of time domain multiplexed filter implementations on the same physical hardware tile perfectly to make optimal use of the multiplier resources [3]. In 32 filter implementation running at 100 MHz with a 1 MHz sample rate, multiplier utilisation is 96%.

In our neuromimetic cochlea implementation, the input samples are broadcast into 22 bandpass filters blocks. Each of these 22 filter blocks is time domain multiplexed to create 32 separate filters. The outputs appear on the 22 output busses at 3 cycle intervals (after an initial latency of 7 cycles). Total 704 filtering output is in pipelined parallel within 1µs.

In the design, all coefficients are stored in 16 RAM128X1Ss distributed RAMs in advance. The intermediate values are stored in synchronous block RAM (RAM16_S18_S18). The 22 parallel filter blocks take 22 multipliers and 22 RAMs. The 704 frequency filters are implemented on a single Xilinx Virtex II XC2V6000-4 running in real time, the frequency span of the input signals is 20 kHz to 200 kHz. To improve the precision of the design, the intermediate values of the algorithm may be extended to 32bit, so that double the multipliers and block RAMs are required for the bandpass filters' implementation.

3 Conclusions

The pipelined parallel architecture of Butterworth bandpass filters has been shown to be implemented efficiently at a high speed. The realization of the entire neuromimetic cochlea on the chosen chip is achievable. The precision of the filters can be further improved by extending the precision of intermediate values within the filter. The current implementation is a scalable design and the filter coefficients are programmable.

Acknowledgment. The CIRCE project is funded by the European Commission, FET-LPS Initiative (IST-2001-35144).

References

1. Nobuo Suga : Cortical Computational Maps for Auditory Imaging. Neural Networks, Vol.3, (1990) 3-21
2. H. Peremans and R. Müller : A comprehensive robotic model for neural & acoustic signal processing in bats. Proc. of the 1st Int. IEEE EMBS Conf. on Neural Engineering, Capri. March,(2003) 458-461.
3. Virtex-II Complete Data Sheet, Xilinx Inc., October 14, 2003.

FPGA-Based Computation for Maximum Likelihood Phylogenetic Tree Evaluation

Terrence S.T. Mak and K.P. Lam

Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong,
Hong Kong, China
{stmak, kplam}@se.cuhk.edu.hk

Abstract. Phylogenetic tree is a meaningful representation for the evolutionary history of different organisms. Due to the exponentially increasing search space for the optimal Maximum Likelihood (ML) criterion, the phylogeny inference is classified as NP-hard. Heuristic search makes use of the likelihood evaluation function extensively to give score for the candidate solutions. This tree evaluation becomes a critical but computationally demanding task. In this paper, we address the computational issue for the evaluation of a phylogenetic tree under ML criterion, for a given set *D* of *n* properly aligned DNA sequences each with *l* nucleotide sites. We present a high performance field programmable gate arrays (FPGA) implementation for tackling the tree evaluation process in order to speed up the tree reconstruction. An efficient fine-grained parallel design based on the idea of partial likelihood is proposed. It has been shown to be 100 times faster than solely using the software.

1 Introduction

Molecular phylogeny is the study of the evolutionary processes of different organisms using molecular data (i.e. DNA and protein), which provides significant information for molecular biologists to understand bacteria and virus behaviors. Evolutionary relationships are often encoded as a bifurcating unrooted[1] tree. Although there are a variety of methods for reconstructing the phylogenetic tree from molecular data, Maximum Likelihood (*ML*) has become one of the most popular approaches [1]. However, it is a difficult task to find the optimal solution based on the *ML* model, simply because of the exponentially growth of the possible tree topologies with the number of taxa. Though applying the heuristics can reduce the search space for a near-optimal solution, the tree evaluation is computationally demanding and is being used repeatedly. The long tree evaluation time results in the reduction of the overall search speed. In order to reduce the computational load of tree evaluation, we intro

[1] An unrooted *n*-taxa tree refers to a tree with *n* leaf nodes (or taxa) and *n-2* internal nodes. Internal nodes and leaf nodes are of degree 3 and 1, respectively. The degree of a node is the number of branches (or links) directly connected to the node.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1076–1079, 2004.

duce a speed-up scheme by using dedicate hardware for realizing the required computing. In this paper, we present an FPGA-based implementation for the phylogenetic tree likelihood computation, which is based on the idea of partial likelihood. The architecture provides fine-grained parallelism on the likelihood computation, which can speed up the tree evaluation process.

2 Maximum Likelihood Model

The maximum-likelihood model of interest is a probability function which has three input sets: an unrooted bifurcating tree *T*, a molecular substitution model *M*, and *n* aligned DNA sequences with *l* based pairs (i.e. each individual base pair is called a *site*), given as an $n \times l$ matrix *D*. The substitution model has a set of parameters that defines the rate of mutation from one base DNA to another. A simple model (i.e. Jukes-Cantor Model [3]) is considered, where all rates of substitution are equal. Following [2], the probabilistic likelihood is defined as

$$\ln L = P(D \mid T, M) = \sum_{s=1}^{l} \ln P(D_s \mid T, M) \tag{1}$$

where $D= [D_{rs}]_{n\times l}$, $D_{rs} \in \{A,T,G,C\}$, D_s is the site pattern of *D* at site *s* (i.e. the s^{th} column of *D*), and $P(D/T,M)$ is the conditional probability calculated from the specified substitution model. *M* and tree *T* are yet to be determined. The likelihood $P(D_s/T,M)$ is the product of transition probabilities and the prior probability representing the incurred mutations. In general the number of product terms can be increasing exponentially[2] with the number of taxa.

A "pruning" algorithm described in [2] can effectively compute the likelihood value and it is formally formulated by introducing the idea of *partial likelihood* [4]. The likelihood value can be computed by using a recursive routine, in which the expression is evaluated by working from the partial likelihood of smaller sub-tree to the larger sub-tree. Subsequently, the likelihood value for the whole tree is estimated. Based on this approach, number of multiplications is reduced[3] and is linearly proportional to the number of taxa. The partial likelihood can be regarded as a matrix $Q=[Q_{ik}]_{c\times d}$, where *c* is the number of states and *d* is the nodes index, the recursive routine can be "expanded" to be the evaluation of a matrix. The order of computing these entries is importance, as data dependency exists among these partial likelihood values. The dependency is defined by the tree topology. The entries of the matrix are estimated while following a post-order tree traversal. After all entries of the matrix are computed, the likelihood value can be readily obtained.

[2] For *n* DNA sequences each with *l* nucleotide sites, the likelihood computation for the *n*-taxa unrooted tree requires $4^{n-2}(2n\text{-}4)\ l$ multiplications

[3] The required number of multiplications is $4(5n\text{-}12)l$, where $n>3$.

3 Implementation

We design a *likelihood computational unit*, which can be used to calculate the likelihood values of a set of DNA sequences. The unit inputs the DNA sequences and the partial likelihood updating order, and outputs the overall likelihood value. There are a few routines in the computational unit, which are the partial likelihood computation, root likelihood estimation and binary logarithmic computation. As the partial likelihood computation is the most important routine and it takes most of the computation effort, our discussion will focus on the partial likelihood computation in the following paragraphs.

Each row of the partial likelihood matrix is independently estimated. Therefore they can be computed in parallel. For DNA, there are always 4 states. Thus, we have 4 rows in the matrix. Fig. 1 shows the sequential and parallel ways to obtain the partial likelihood. The figure is referring to a tree with 4 terminal nodes (i.e. A, B, C and D). The node E and F are the internal nodes, which are the parent nodes of the leaves. The nodes in the figure represent the process of computing the corresponding entry. For the software implementation, the matrix is computed in sequential order. As the partial likelihood values for each state is independent, the computation of each node (partial likelihood) can be executed in parallel (See Fig. 1).

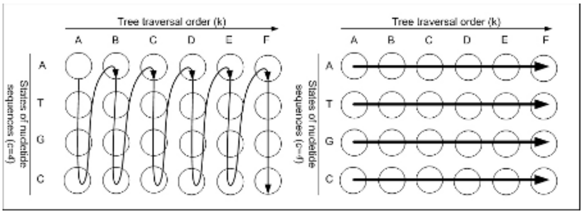

Tree traversal order (k)
A
B
C
D
E
F
States of nucleotide sequences (c=4)
A
T
G
C
Tree traversal order (k)
A
B
C
D
E
F
States of nucleotide sequences (c=4)
A
T
G
C

Fig. 1. Comparison for the execution order between software and hardware implementation

Since the partial likelihood can be expressed as a matrix, it is stored in the partial-likelihood RAMs and addressed by the node index. We introduce 4 RAMs in parallel and each stores one row of the matrix. Therefore the partial likelihood of different state (i.e. a column of the matrix) can be obtained at the same time. The RAMs are tightly coupling with 4 processing units running in parallel, in which the entries of the partial likelihood matrix are estimated. To update a matrix entry involves a number of multiplications and additions, which can be done in either sequential or parallel. The design of processing unit can be varied at different degree of parallelism and it is up to the availability of hardware resources. Another RAM is used to store the matrix estimation order, which defines the dependency of the values in the matrix. The overall likelihood value can be obtained readily once finished the partial likelihood matrix evaluation. The overall value is taken logarithm and accumulated before stored in the register.

The likelihood for each site pattern D_s is independent of each other. Therefore, we can partition the DNA sequences into many equal length segments and stored in distributed memories. Then a number of likelihood computational units are assigned to access these segments at the same time. The final results of each computational unit will be summed up as the overall likelihood of the whole sequence. This parallelism can be effective to reduce the computational load of the likelihood value for a long sequence. The speed-up factor (i.e. the ratio of time for sequential execution to parallel execution) equals to the number of computational units implemented. Certainly, the speed-up factor is limited by the availability of FPGA resources, where the resources consumption is also directly proportional the number of computational units implemented.

We implemented our design using a Virtex-II XC2V6000 FPGA on the ADM-XRC-II board which communicates with the host PC by the PCI interface. The FPGA is running at 77.5MHz. Comparison is made between the software and FPGA implementation. The time is measured by varying the number of taxa on a fixed length (l=500) and frequency. We found that the computation time increases linearly with problem size (n). FPGA with single computation unit offers around 20 times speed-up when compared to the software. For up to five computational units, that each unit estimates the sequences of 100 sites, the FPGA provides 100 times acceleration.

4 Conclusion

An efficient FPGA implementation for maximum-likelihood (ML) phylogenetic tree evaluation is proposed. The design can provide speed-up for the critical tree evaluation process and reduce computational burden for the phylogenetic tree reconstruction. The idea of partial likelihood matrix estimation is presented. It can be mapped to a FPGA-based fine-grained parallel architecture, which provides parallel computation for the likelihood value. The design has been realized with Xilinx Virtex-II FPGA. Experimental results show that the FPGA-based tree evaluation is 100 times faster than the software solution. Evaluation of large-scale phylogenetic tree for hundreds DNA sequences using FPGA is regarded as future work.

References

[1] Kishino H., et al., "Maximum Likelihood Inference of Protein Phylogeny and the Origin of Chloroplasts", J. Mol. Evol., 31:151-160, 1990

[2] Felsenstein J., "Evolutionary trees from DNA sequences: a maximum likelihood approach", J. Mol. Evol., 17:368-376, 1981

[3] T.H. Jukes and C.R. cantor, "Evolution of protein molecules", in H.N. Munro (ed.), Mammalian Protein Metabolism, pp. 21-123, Academic Press, New York, 1969

[4] J. Adachi et al., "MOLPHY version 2.3, program for molecular phylogenetics based on Maximum Likelihood", Technical report, The Institute of Statistical Mathematics, Tokyo, Japan, 1996

Processing Repetitive Sequence Structures with Mismatches at Streaming Rate*

Albert A. Conti, Tom Van Court, and Martin C. Herbordt

Department of Electrical and Computer Engineering
Boston University, Boston, MA 02215
{herbordt|alconti|tvancour}@bu.edu

Abstract. With the accelerating growth of biological databases and the beginning of genome-scale processing, cost-effective high-performance sequence analysis remains an essential problem in bioinformatics. We examine the use of FPGAs for finding repetitive structures such as tandem repeats and palindromes under various mismatch models. For all problems addressed here, we process strings in *streaming mode* and obtain processing times of 5ns per character for arbitrary length strings. Using a Xilinx XC2VP100, we can find: (i) all repeats up to size 1024, each with any number of mismatches; (ii) all precise tandem arrays with repeats up to size 1024; (iii) all palindromes up to size 256, each with any number of mismatches, or (iv) a somewhat smaller size of (i) and (iii) with a single insertion or deletion. The speed-up factors range from 250 to 6000 over an efficient serial implementation which is itself many times faster than a direct implementation of a theoretically optimal serial algorithm.

1 Background, Models, and Results

The dominant production techniques for biological sequence analysis have been based on partial string matching, either with the goal of obtaining a precise solution (e.g. with dynamic programming) or, much more commonly, a fast solution (e.g. with heuristic techniques such as BLAST). However, there are sequences of DNA for which these heuristic techniques do not work very well. These are often repetitive sequences; processing these has been found essential as a complement to—or as a preprocessing step for—not only partial string matching but also sequence assembly [1]. Repetitive sequences are also critical in their own right; see e.g. surveys in [2,3]. There has been much theoretical work (e.g. [4] and the fine introduction by Gusfield [3]), but unlike sequence alignment—where there has been some interest in FPGA-based acceleration—analysis of non-trivial repetitive structures has as yet received little such attention.

In this initial study, we investigate the capabilities of a high-end FPGA, the Xilinx XC2VP100, with respect to a set of basic problems in the analysis of repetitive sequence structures. To constrain the study, we use a single algorithmic

* This work was supported in part by the NSF through award 9702483 and the NIH through award RR020209-01.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1080–1083, 2004.

model: the data are passed through the FPGA in streaming mode and processed systolically. We have found that hardware structures based on simple cells are sufficient to provide the basic functions of finding all repeats and palindromes of arbitrary size with an arbitrary number of mismatch errors constrained only by what can fit on the chip. The simplicity of the underlying structures ensures the fast cycle time: all processing described here can be done at a rate of one character per 5ns. With the cycle time fixed, the question becomes the complexity of the problem that can be solved within the given hardware constraint.

The tasks shown in Table 1 were examined, implemented (in VHDL; synthesis and place-and-route using Xilinx Tools), and analyzed. Each of these tasks are for strings of arbitrary length but with n determined by available hardware. By *mismatch,* we mean Hamming Distance; by *edit error,* we mean an insertion, a deletion, or a mismatch error. By *tandem array,* we mean a periodic string that can be written a^r for some $r \geq 2$. Two tasks not addressed here are tandem repeats (or palindromes) with multiple edit errors and tandem arrays with errors. Dealing with multiple edit errors is probably best done using dynamic programming, while tandem arrays with errors may be best analyzed statistically [2].

Table 1. Enumeration of tasks and maximum sizes that fit on the Xilinx XC2VP100.

Task	n
1. All tandem repeats of size l from 1 to n with up to n mismatches	1024
2. All palindromes of size l from 1 to n with up to n mismatches	256
3. Same as 1 with one edit error	64
4. Same as 2 with one edit error	40
5. Tandem arrays with period of size l from 1 to n	1024

For performance comparison, we also implemented the tasks in C and ran them on a 3GHz Xeon-based PC. We found the FPGA versions to be from 250 to 6000 times faster. The C programs use trivial algorithms; we found that programs based on the theoretically optimal serial algorithms are complex and many times slower for the relevant problem sizes.

2 Algorithm and Implementation Sketches

Here we very briefly sketch the algorithms and implementations. For extended versions, including methods of filtering the output also in streaming mode, please refer to our web site (www.bu.edu/caadlab). Generally, we use a two tier structure. In the first, we feed the string through an array of comparators/counters and get a series of results for every point. These include the size of the repeat or palindrome about that point and the number of mismatches. In the second tier, which we call postprocessing, we decide what information to send off chip, and determine higher order structure such as arrays of repeats. Tasks 1 and 2 are done with basic structures alone (see Figures 1 and 2); the other three depend also on output generated during postprocessing.

Task 1. Let α_1 and α_2 be two strings with equal length which would be identical but for k mismatches. Let characters be shifted through $\alpha_1\alpha_2$ with i_1, i_2, o_1, and o_2 being the characters that get shifted into α_1 and α_2 and out of α_1 and α_2, respectively. We keep running counts of k mismatches as shown in Figure 1.

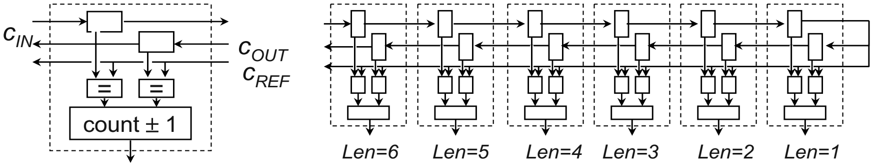


Fig. 1. Structure for initial tracking of tandem repeats of all lengths and mismatches.

Task 2. A substring about the center $\alpha_1\alpha_2$ is a palindrome of length l when the first l characters of α_2 match those of $\alpha_1^{reverse}$. We again fold the string upon itself (Figure 2) but instead of comparing a single mid-point character with all other characters in the string, we perform pair-wise comparisons for all characters from 1 to $n/2$ of $\alpha_1^{reverse}$ and α_2. The results form a bit vector of length $n/2$. For each length l from 1 to $n/2$, the number of errors in the palindrome is equal to the number of zeros (mismatches) between 1 to l.

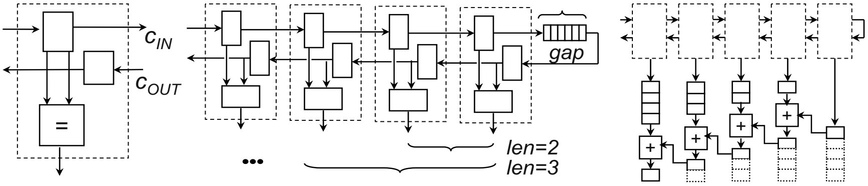


Fig. 2. Part of structure for tracking palindromes repeats of all lengths and all mismatches. We generate the initial bit vector and then sum for each length.

Task 3. For every substring $\alpha_1\alpha_2$ of size $2l$, a matching prefix of length p in α_1 and α_2 will be a matching suffix of the same length in $l - p$ cycles. This allows us to generate, on every cycle and for every length l from 1 to $n/2$, the maximal prefix and suffix common to α_1 and α_2.

Task 4. The circuit described above for k mismatches was replicated twice with the second and third copies offset one spot ahead and behind, respectively. A two-dimensional array keeps track of whether or not a palindrome for each possible edit error position in each length has a number of matches greater than or equal to the length minus the number of allowable mismatches.

Task 5. Using the hardware in Task 1, when the counter for any length l (the l-counter) reaches l, a tandem repeat at that position has no errors. As the string is streamed character by character, the precise tandem array of that period continues for as long as the contents of the l-counter remains equal to l.

References

1. Schuler, G.: Sequence alignment and database searching. In Baxevanis, A., Ouellette, B., eds.: Bioinformatics: A Practical Guide. Wiley (2001)
2. Benson, G.: Tandem repeats finder: A program to analyze DNA sequences. Nucleic Acids Research **27** (1999) 573–580
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge, U.K. (1997)
4. Landau, G., Schmidt, J., Sokol, D.: An algorithm for approximate tandem repeats. J. Computational Biology **8** (2001) 1–18

Artificial Neural Networks Processor – A Hardware Implementation Using a FPGA

Pedro Ferreira, Pedro Ribeiro, Ana Antunes, and Fernando Morgado Dias

Escola Superior de Tecnologia de Setúbal do Instituto Politécnico de Setúbal,
Departamento de Engenharia Electrotécnica, Campus do IPS, Estefanilha, 2914-508 Setúbal,
Portugal Tel: +351 265 790000,
pedromdsf@netcabo.pt; {pr-apinf,aantunes,fmdias}@est.ips.pt

Abstract. Several implementations of Artificial Neural Networks have been reported in scientific papers. Nevertheless, these implementations do not allow the direct use of off-line trained networks because of the much lower precision when compared with the software solutions where they are prepared or modifications in the activation function. In the present work a hardware solution called Artificial Neural Network Processor, using a FPGA, fits the requirements for a direct implementation of Feedforward Neural Networks, because of the high resolution and accurate activation function that were obtained. The resulting hardware solution is tested with data from a real system to confirm that it can correctly implement the models prepared off-line with MATLAB.

1 Introduction

Artificial Neural Networks (ANN) became a common solution for a wide variety of problems in many fields, such as control and pattern recognition to name but a few. It is therefore not surprising that some of the solutions have reached an implementation stage where specific hardware is considered to be a better solution than the most common implementation within a personal computer (PC) or workstation.

A number of reasons can be pointed out as the motivation for this: need for higher processing speed, reduced cost for each implementation and reliability.

Considering the possible solutions for a digital implementation the FPGA solution is the most interesting taking into account the balance performance/price.

In the literature it is possible to verify that several solutions have already been tested in the FPGA context. Nevertheless, the solutions that were found do not allow the direct use of the neural models that are prepared frequently with software (like MATLAB or specific software for ANN) within PCs or workstations.

All the solutions that the authors were able to verify present either a much lower precision when compared with these software solutions [1], [3] or modifications in the activation function that make them unacceptable to use directly the weights previously prepared [6], [7].

In the present work a hardware solution called Artificial Neural Network Processor (ANNP), using a FPGA, designed to fit the above requirements for a specific application is presented.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1084–1086, 2004.

2 Hardware Implementation

The notation chosen was 32 bits floating point according to the IEEE 754-1985 standard. Although it has been stated in [2] that "A few attempts have been made to implement ANNs in FPGA hardware with floating point weights. However, no successful implementation has been reported to date.", and [5] have concluded that "floating point precision is still not feasible in FPGA based ANNs", there were at least two applications reported: [3] which used floating point notation of 17 bits and [4], which used 24 bits.

In the present work, the activation function used was the hyperbolic tangent.

As implementing the full functions is too expensive, the implementation was done using piece-wise linear approximation according to the following steps: Choose the first linear section; Compare this linear approximation with the hyperbolic tangent implemented in MATLAB in order to verify when the maximum allowed error is met; Start a new linear section; Repeat the operation until the region needed was fully represented.

This algorithm led to the representation of the hyperbolic tangent with 256 linear sections and provides a maximum error of 0,0000218 in values that are in the range of [-1,1] (function output). This can be compared with other solutions like the one using the Taylor series used in [3], which obtained an error of 0,51% and piecewise-linear approximation used in [4], which obtained 0,0254 of "standard deviation with respect to the analytic form".

It is worth to verify that to obtain this error with the classical LUT approach 18110 samples of the function were needed. These samples represented in the 32 bits notation used would require more than a single FPGA of the type used, just to represent the activation function.

The hardware platform used is the Cyclone EP1C20F324C7 FPGA from ALTERA, in the kit Cyclone SmartPack from Parallax.

The ANNP was developed using the VHDL language and implements the following components: 32 bits multiplier in floating point notation, 32 bits adder in floating point notation, activation function, memory blocks, 3 finite state machines, serial communication using RS-232 protocol.

The implementation of the activation function uses 3 ROMs of 256x32 bits, that is 24.576 bits, which are implemented in 6 blocks of 256x16 bits.

The complete implementation uses 235.008 bits, which are in fact 285.696 bits if the parity bits (and others that are used for special functions) are taken into account, of a total amount of 294.912 bits available, that means that 96,9% of the bits were used.

3 Test System and Results

The hardware implementation of the ANN was tested using several models of a system, which were previously prepared in MATLAB. This system is a reduced scale prototype kiln, which was working under measurement noise. For further details please see [8]. The comparison of the accuracy obtained in control simulations against the MATLAB software is summarized in table 1.

Table 1. Mean Square Error comparison between the MATLAB and FPGA results

Model	MATLAB Simulation			FPGA Results		
	Ramp	Square	Random	Ramp	Square	Random
M-1	1,2146e^{-4}	0,0146	1,3343e^{-4}	1,2151e^{-4}	0,0146	1,3345e^{-4}
M-2	7,8682e^{-5}	0,0040	1,3445e^{-4}	7,8666e^{-5}	0,0040	1,3440e^{-4}
M-3	8,1468e^{-6}	0,0069	1,5426e^{-5}	8,1456e^{-6}	0,0069	1,5426e^{-5}

4 Conclusions

This work proposes a hardware implementation of an ANN using a FPGA. The FPGA was chosen because of the lower prices for a single implementation.

The goal that was searched was to obtain a hardware solution that allowed the direct use of the weights, usually prepared in a software environment with a much higher resolution than the ones usually obtained in FPGAs implementation.

This objective was successfully accomplished as was shown by the control tests done with models of a real system, where the maximum error obtained between the MATLAB and the hardware solution, using the MSE as a measure was of 5e^{-8}.

A new algorithm to apply with piece-wise linear approximation was also presented that allowed high resolution in the implementation of the hyperbolic tangent. This algorithm allows the definition of the maximum error that is acceptable and supplies the number and equations of the corresponding linear sections.

References

1. Lysaght, P., J. Stockwood, J. Law and D. Girma, "Artificial Neural Network Implementation on a Fine-Grained FPGA", in R. W. Hartenstein, M. Z. Servit (Eds.) Field-Programmable Logic, Springer Verlag, pp. 421-431, Czech Republic, 1994.
2. Zhu, J. H. and Peter Sutton, “FPGA implementations of neural networks – a survey of a decade of progress”, 13th International Conference on Field Programmable Logic and Applications, Lisbon, 2003.
3. Arroyo Leon, M. A., A. Ruiz Castro and R.R. Leal Ascenccio, "An artificial neural network on a field programmable gate array as a virtual sensor", Proceedings of The Third International Workshop on Design of Mixed-Mode Integrated Circuits and Applications, Puerto Vallarta, Mexico, pp. 114-117, 1999.
4. Ayala, J. L., A. G. Lomeña, M. López-Vallejo and A. Fernández, “Design of a Pipelined Hardware Architecture for Real-Time Neural Network Computations”, IEEE Midwest Symposium on Circuits and Systems, USA, 2002.
5. Nichols, K., M. Moussa and S. Areibi, “Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks”, CAINE, San Diego California, pp:8-13, 2002.
6. Skrbek, M., “Fast neural network implementation”, Neural Network World, vol.9, nº5, pp.375–391, September 1999.
7. Wolf, D. F., R. A. F. Romero and E. Marques, “Using Embedded Processors in Hardware Models of Artificial Neural Networks”, V Simposio Brasileiro de automação inteligente, Brasil 2001
8. Dias, F. M. and A. M. Mota, ”Comparison between different Control Strategies using Neural Networks”, 9th Mediterranean Conference on Control and Automation, Dubrovnik, Croatia, 2001.

FPGA Implementation of the Ridge Line Following Fingerprint Algorithm

E. Cantó, N. Canyellas, M. Fons, F. Fons, and M. López

Escola Tècnica Superior d'Enginyeria ETSE-URV, Tarragona, Spain
ecanto@etse.urv.es

Abstract. Most biometrics systems are implemented on high performance microprocessors executing complex algorithms on software. In order to develop a low-cost and high-speed coprocessor, floating-point computations have been substituted by fixed-point ones, and a pipeline scheme has been developed.

1 Introduction

An automatic fingerprint authentication system is structured in three different stages: image acquisition, feature extraction and matching. Minutiae matching is the most widely used because it contains useful distinctive information, however it requires complex computations during extraction. Most minutiae extraction algorithms use orientation-field, directional filters, binarization and thinning steps in order to obtain a 1-bit wide image, from where minutiae is easily extracted. In our work we use the Maio-Maltoni ridge line following algorithm [1] because minutiae is extracted directly from the gray-scale image, it is less time-consuming and it can be rewritten without floating point operations. These facts are very important bearing in mind that the goal is to implement a high-speed and low-cost embedded system.

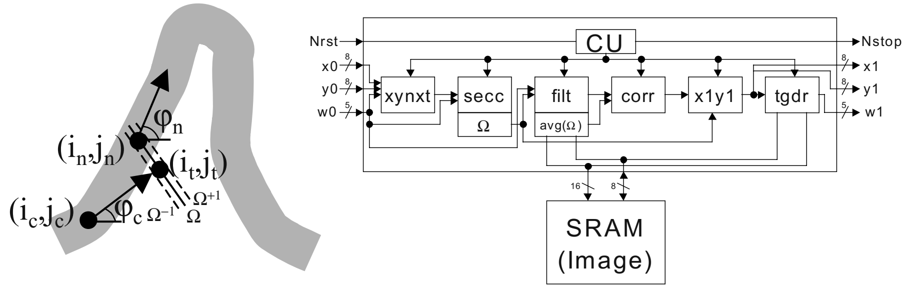
The use of fingerprint biometrics coprocessors is still a young field. A majority of fingerprint OEM modules are based on high performance microprocessors or DSPs, but feature extraction times are about 1 second or more. IKENDI offers the IKE-1 ASIC [2] based on an ARM7TDMI microprocessor plus a coprocessor, and it claims to encode 10-20 times faster than other DSP-powered solutions. The UCLA group has developed the ThumbPod [3] prototype based on a LEONII microprocessor, plus a coprocessor that permits a 55% execution time reduction for the minutiae extraction, although 4 seconds of execution time is still quite high.

2 The Ridge-Following Algorithm

The algorithm is based on the idea of tracking the fingerprint ridge lines on the gray-scale image. As depicted in Fig. 1(a), given an starting point (i_c,j_c) and a starting direction θ_c the algorithm computes a new point (i_t,j_t) by moving μ pixels along direction φ_c. Then, it is computed the section Ω as the set of $2\sigma+1$ points orthogonal to φ_c. The third step involves the computation of an averaged section $\text{avg}(\Omega)$ as the

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1087–1089, 2004.

local average of the pixels belonging to section Ω with the pixels belonging to the two parallel planes Ω^{+1} Ω^{-1} that distant ±1 pixel. The next step computes a correlation corr(Ω) of avg(Ω) against the gaussian mask, and returns the weak local maximum as the closest point to the centre of Ω that accomplishes $\text{corr}(\Omega_{k-1}) \le \text{corr}(\Omega_k) \le \text{corr}(\Omega_{k+1})$, and it is determined the new point (i_n,j_n) and its ridge flow direction φ_n. Then it is updated a binary image of the tracked points in order to prevent examine each ridge more than one time. Finally (i_n,j_n) and φ_n become the new starting point and direction and these steps are repeated until it is reached a stop criterion.



(a) (b)

Fig. 1. (a) Ridge line following algorithm. (b) Overview of the coprocessor

We have analysed the algorithm with the parameter values adopted in [1], and then we have modified some computations in order to minimise the computational requirements bearing in mind a FPGA implementation of critical tasks.

The most important change in our modified algorithm relates to the ridge direction computation. The method used in [1] is computationally much more expensive than the presented in [4], that we have modified in order to obtain 12 discrete directions with angles varying in steps of 15°, which corresponds with $\mu=3$ as used in [1].

An inspection of the extraction algorithm shows another computational expensive step. During the process of finding the local maximum in a section Ω, the algorithm performs first the local average of the gray levels of the pixels that belong to Ω, Ω^{+1} and Ω^{-1}, to finally compute a correlation with a gaussian mask. The computations of the ridge direction and correlation involve divisions by constants, but in our algorithm they are not performed. This is equivalent to a change of scale, and does not affect the position of the weak local maximum. Moreover, a correlation involves products that can be substituted by shift operations if performed with constants that are powers of 2, in a very efficient way if they are implemented on hardware.

3 The Coprocessor

It has been profiled the execution times for an entire fingerprint image on a ARM7TDMI running at 50MHz, to finally implement on the FPGA the most critical functions that occupy the 73,5% of the overall execution time.

As depicted in Figure 1(b), the coprocessor has been divided in six stages plus a control unit (CU). The coprocessor works with a 256*256 pixels image of 8-bit gray levels stored on a external SRAM, and with fixed parameters μ=3 and σ=7 as used by the Maio-Maltoni algorithm [3]. The starting point (i_c,j_c) and indexed angle φ_c are read from the x0,y0 and w0 ports, returning the weak local maximum point (i_t,j_t) and indexed angle φ_t of the ridge flow direction in the x1,y1 and w1 ports. The stages were implemented using a pipeline scheme to increase the clock frequency and improve throughput. Intermediate results are stored to and read from internal FPGA memory.

The coprocessor has been described on VHDL and mapped on a low-cost FPGA SPARTAN-II of 30Kgates equivalent, occupying 87% of logic cells and running at a maximum clock frequency of 76MHz. The execution of the steps performed by the FPGA is 14.4µs running at 50MHz, while the time devoted by the ARM7TDMI at the same clock frequency is 211µs.

The FPGA reduces in 93% the computation time when compared with the microprocessor, due to the pipeline scheme of the coprocessor architecture and the parallel execution of computations into its stages. When it is compared the overall execution time of the algorithm running in the ARM not aided or aided with the FPGA, it is reduced from 700ms to 215ms, that is a reduction of about 70%.

4 Conclusions

Biometrics algorithm designers usually do not have into account the computational requirements of the platform that runs the software implementation. This fact leads that fingerprint authentication systems are usually based on high-performance PCs, embedded microprocessors or DSPs. This work demonstrates that algorithms can be rewritten to avoid floating-point operations and other complex functions in order to reduce the computation requirements. This way it is possible to effectively implement, in terms of low-cost and high-speed, the critical tasks on a coprocessor mapped on a low-cost FPGA.

References

[1] Maio, D.; Maltoni, D. “Direct Gray-Scale Minutiae Detection In Fingerprints” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol 19, No. 1. January 1997

[2] IKE-1 Multifunction Controller for Image Processing Applications. http://www.ikendi.com

[3] S. Yang, K. Sakiyama, I. M. Verbauwhede, “A Compact and Efficient Fingerprint Verification System for Secure Embedded Devices”, Proc. Of the 37th Asilomar Conference on Signals, Systems and Computers, Novermber 2003.

[4] Halici, U.; Ongun G. “Fingerprint Classification Through Self-Organized Feature Maps Modified to Treat Uncertainities” Proceedings of the IEEE, vol.84, no. 10, October 1996.

A Dynamically Reconfigurable Function-Unit for Error Detection and Correction in Mobile Terminals

Thilo Pionteck, Thomas Stiefmeier, Thorsten R. Staake, and Manfred Glesner

Darmstadt University of Technology
Institute of Microelectronic Systems
Karlstr. 15, 64283 Darmstadt, Germany
pionteck@mes.tu-darmstadt.de

Abstract. This work presents the design of a dynamically reconfigurable function unit supporting Cyclic Redundancy Checks and Reed-Solomon Codes with different code length. The architecture is designed for the usage in mobile wireless communication systems and is optimized concerning area and power consumption.

1 Introduction

Error detection and error correction are important tasks in wireless communication systems. Due to channel distortions, transmitted data can be falsified. It is essential for the receiver to detect such errors and it is desirable that the receiver can correct them as well. A very popular error detection code which is used in actual WLAN standards is the Cyclic Redundancy Check (CRC). The basic idea is to expand a given message by a check sequence. Therefore a message word $u(k)$ is divided by a generator polynomial $g(x)$ using modulo-2 arithmetic. The remainder of this devision forms the check sequence and is appended to the original message. There exist two common approaches to perform CRC computations, a bit-serial and a bit-parallel one. For this work, a bit-parallel approach is used which is based on operations in Galois Fields [1].

For the correction of errors, Reed-Solomon (RS) Codes were chosen. Like the bit-parallel approach for CRC calculation, they are based on operations in Galois Fields. A $RS(n, k)$ code word $v(x)$ consists of n symbols of length m, divided into k message symbols and $(n-k)$ parity symbols. Up to $(n-k)$ symbol errors can be detected and $t = (n-k)/2$ symbol errors can be corrected [2].

Both algorithms were implemented in a function-specific dynamically reconfigurable function unit (RFU). The RFU is optimized concerning area efficiency, and fulfills the performance requirements for actual wireless communication standards. In combination with a processor, the RFU allows the design of a flexible solution for the MAC (Medium Access Control) layer of WLANs. All memory elements and arithmetic blocks of the RFU can directly be accessed by the processor. This enables the processor to utilize these blocks for additional tasks like multiplication in the Galois Field required for encryption standards like AES (Advanced Encryption Standard).

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1090–1092, 2004.

2 Reconfigurable Function-Unit

The RFU is composed of four different blocks. Hardware support for error detection and error correction is provided by the *ECM (Error Control Module)* while a *AES* block is used for encryption/decryption tasks. The *AES* block is mentioned for completeness reasons only. As the RFU is designated for the use in processors realizing the MAC-layer of WLANs, the *AES* block is integrated to provide hardware support for encryption/decryption tasks. A description of the *AES* block can be found in [3]. The two remaining blocks are the *LUT Module* and the *Common Resource* block. The *LUT Module* combines all memory elements of the *AES* and the *ECM* block while the *Common Resource* block combines all complex arithmetic elements like the configurable Galois Field multipliers.

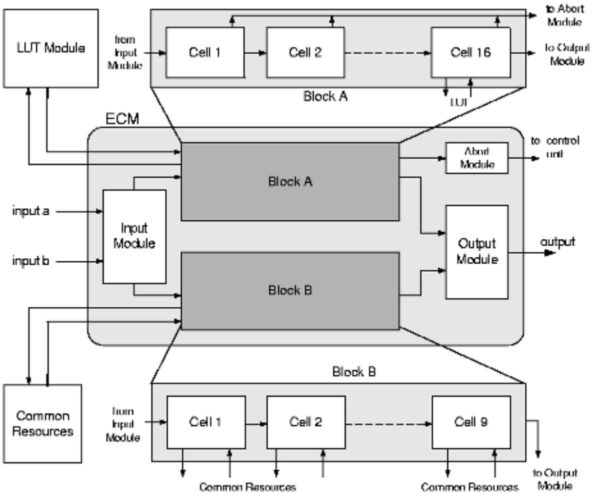


Fig. 1. Structure of the Error Control Module

The structure of the ECM block is depicted in figure 1. It is comprised of two major blocks, *Block A* and *Block B*. These two blocks are derived from the symmetry of the underlying hardware structure. *Block A* is used for the *Syndrome Calculation* and RS encoding. *Block B* is used for *Forney Algorithm* and CRC encoding/decoding. *Euclid's Algorithm* and *Chien Search* require both, *Block A* and *Block B*.

Computations like Reed-Solomon Code decoding require several reconfigurations during runtime. In order to release the processor from the control overhead of continuous reconfiguration, the control logic of the RFU is capable of performing a sequence of configuration steps autonomously. The control logic is placed in the *ID* stage of a pipelined RISC processor, while the RFU is placed in parallel to the existing ALU in the *EX* stage.

3 Results

For achieving synthesis and performance results, the RFU was integrated into a 32 bit 5 stage pipelined RISC core, derived from the DLX architecture. Synthesis was done using Synopsys' Design Analyzer with a $0.25\mu m$ 1P5M CMOS standard cell technology. For larger memories like the *LUT Module* inside the *ECM* block and the configuration tables in the RFU control logic, RAM macro cells have been deployed.

Table 1. Synthesis Results of the Reference Design and *ECM* Block

Module	**Area** $[\mu m^2]$	**Freq.**$[MHz]$
CRC En-/Decoder	204.711	205
RS Encoder	235.151	209
RS Decoder	1.955.140	59
Total Area	2.395.002	-

(a) Reference Design

Module	**Area** $[\mu m^2]$	**Freq.**$[MHz]$
Block A	191.704	140
Block B	207.755	172
Other Blocks	61.908	694
ECM	461.367	137

(b) *ECM* Block

Table 1 shows the synthesis and performance results of a reference design and the reconfigurable *ECM* block. Table 1(a) gives the results for parallel CRC-8 encoding/decoding using Galois Field arithmetic and for RS(255,239) encoding/ decoding using a standard approach. The synthesis results for the *ECM* block can be found in table 1(b). The *ECM* block requires only about 20 % of the area of the reference design. Even if the *Common Resource* block and the *LUT Module* of the RFU are counted to the area of the reconfigurable design, hardware savings of about 28 % can be achieved. For RS(255,239) encoding and CRC8 encoding/decoding the reference architecture is faster than the reconfigurable design, but for RS(255,239) decoding a speed-up of 1,68 could be achieved. Despite of this, all throughput values are more than sufficient for the data rates required for mobile terminals.

4 Conclusion

In this paper a function-specific dynamically reconfigurable architecture for error detection and error correction has been presented. Synthesis and performance results proved that the architecture offers an attractive alternative to standard implementations.

References

1. Michael Ji, H., Killian, E.: Fast Parallel CRC Algorithm and Implementation on a Configurable Processor. IEEE Intern. Conference on Communications **3** (2002)
2. Lee, H., Yu, M.L., Song, L.: VLSI Design of Reed-Solomon Decoder Architectures. IEEE Intern. Symposium on Circuits and Systems **5** (2000)
3. Pionteck, T., Staake, T., Stiefmeier, T., Kabulepa, L.D., Glesner, M.: Design of a Reconfigurable AES Encryption/Decryption Engine for Mobile Terminals. IEEE International Symposium on Circuits and Systems (2004)

Flow Monitoring in High-Speed Networks with 2D Hash Tables

David Nguyen, Joseph Zambreno, and Gokhan Memik

Department of Electrical and Computer Engineering
Northwestern University
Evanston, Illinois 60208
{dnguyen, zambro1, memik}@ece.northwestern.edu

Abstract. Flow monitoring is a required task for a variety of networking applications including fair scheduling and intrusion/anomaly detection. Existing flow monitoring techniques are implemented in software, which are insufficient for real-time monitoring in high-speed networks. In this paper, we present the design of a flow monitoring scheme based on two-dimensional hash tables. Taking advantage of FPGA technology, we exploit the use of parallelism in our implementation for both accuracy and performance. We present four techniques based on this two-dimensional hash table scheme. Using a simulation environment that processes packet traces, our implementation can find flow information within 8% of the actual value while achieving link speeds exceeding 60 Gbps for a workload with constant packet sizes of 40 bytes.

1 Introduction

There is a tremendous growth in the complexity of networking applications. Many applications (such as QoS, fair packet scheduling, intrusion/anomaly detection, firewalls, traffic engineering) require flow information[1] [5]. Because of increasing wire speeds, there is a need for hardware-based flow monitoring techniques in high-speed networks. However, most routers do not implement flow monitoring. The existing solutions, which also include software solutions, are either too slow or inaccurate.

In this paper, we present four novel techniques utilizing a two dimensional hash table to gather flow information. We implement our Flow Monitoring Unit (FMU) using a Xilinx Virtex II XC2V8000 [8] chip and achieve throughput speeds up to 73 Gbps without sacrificing accuracy. We observe different designs perform better for certain workloads/requirements. Because network traffic profiles are all unique, FPGAs are an attractive implementation choice for their reconfigurable properties.

In Section 2, we explain the flow monitoring techniques. Section 3 presents the FPGA implementation. In Section 4, we discuss the experimental results and Section 5 concludes the paper.

[1] In this paper, the terms flow and session are used interchangeably; both correspond to a TCP session. Hence, flow information is statistics (such as total traffic generated) about a TCP connection collected at a router.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1093–1097, 2004.

2 Flow Monitoring Unit (FMU)

2.1 Queries

The host machine uses the FMU unit by sending two types of queries.

UPDATE(k, v)	:	Increase the value of the key k by v.
GET(k)	:	Return the value for the key k.

The key k is a combination of five TCP packet header fields: source IP, destination IP, source port, destination port, and protocol. In our design, the FMU stores the number of packets for each flow. Note, while tracking different flows, it is possible for collisions to occur. Therefore, the GET query does not always return the correct value. We will discuss this in Section 4 with error analysis.

2.2 Flow Monitoring Techniques

As mentioned, our FMU is based on hashing. We chose the Jenkins hash function [4] for this study for its proven performance for hash tables. For a two-dimensional hash function with dimensions NxS, there are N hash tables each with S elements. Each of these tables is addressed by a different hash function[2]. The GET(k) function takes the results from each hash table N and uses one of the following techniques:

Min FMU: The simplest technique is called the *min FMU (MIFMU)*. MIFMU reads the corresponding values from the tables and returns the smallest value. This method is most accurate for large flows.

$$\mathrm{min}\ \{T_i[h_i(k)]\},\quad \forall i \in \{0,..,N-1\}$$

Median FMU: The second technique is the *median FMU (MEFMU)*. As stated, this method returns the median of the values (corrected with a balanced hash factor sum/S) from the tables for a key k.

$$\mathrm{median}\ \{T_i[h_i(k)] - \frac{sum}{S}\},\quad \forall i \in \{0,..,N-1\}$$

Collision Estimate FMU: The *collision estimate FMU (CEFMU)* estimates the number of collisions for each hash bucket and returns the output values according to a collision counter. The collision counter is incremented when the current access does not match the last access to the hash table. An additional table ($C_i[h_i(k)]$) stores the collision counters. This method is most accurate for small flows.

$$\mathrm{min}\ \{T_i[h_i(k)] - C_i[h_i(k)]\},\quad \forall i \in \{0,..,N-1\}$$

Hybrid FMU: The final technique is the *hybrid FMU (HYFMU)*, which runs both CEFMU and MIFMU techniques in parallel and returns one value. Again by virtue of FPGA design, there is a negligible performance hit. A threshold value is used to select the hybrid value. HYFMU subtracts the output of CEFMU from the output of

[2] In our implementation, to achieve equal timing, we utilized the same hash function with different hash seeds to generate the effect of different hash functions.

MIFMU. Given a threshold, MIFMU is used for larger estimates and CEFMU for smaller estimates.

3 FPGA Implementation

We implemented the FMU using the Synplify Pro synthesis tool [7] and Xilinx Design Manager implementation tools. We chose Xilinx VirtexII XC2V8000 FPGA as our target chip. The FMU architecture is presented in Figure 1. According to the FMU type (MIFMU, MEFMU, CEFMU, HYFMU), the selection mechanism returns the corresponding output. The resource limit for this chip was memory (S=80,000 entries) for the hash tables. This fact favors using two-dimensional hash tables. For one, there is smaller access latency because multiple smaller tables are accessed in parallel. Also, the two-dimensional hash table design inherently performs better than a single hash table.

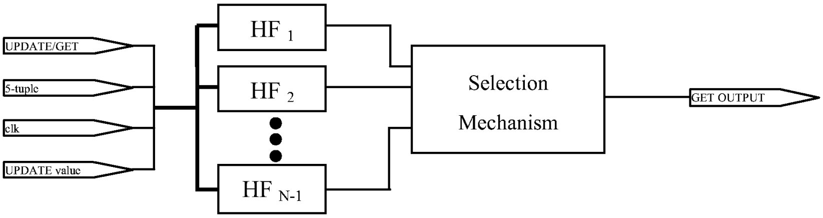


Fig. 1. Overview of the FMU

Table 1 presents the critical path for different FMU designs. Extensive pipelining increases the overall throughput significantly. The rightmost column of Table 1 presents the corresponding maximum bandwidth supported for GET queries. This value is calculated for constant packet sizes of 40 bytes.

Table 1. The latency of critical paths of various FMU components

Configuration	Critical Path Delay	Max. Bandwidth
N = 1, S = 8K	6.63 ns.	48.3 Gbps
N = 4, S = 2K, MIFMU	4.33 ns	73.9 Gbps
N = 4, S = 2K, MEFMU	4.99 ns.	64.1 Gbps
N = 4, S = 2K, CEFMU	4.33 ns.	73.9 Gbps
N = 4, S = 2K, HYFMU (combining min and CE)	4.99 ns.	64.1 Gbps

4 Experimental Results

In this section, we present the simulation results for the different FMU techniques (MIFMU, MEFMU, CEFMU, and HYFMU as explained in Section 2.2) we have developed. We implemented a simulator that processes NLANR packet traces [6] and executes the FMU techniques. For error analysis, the simulator finds the exact number of packets for each flow in a packet trace. We report the *error rate*, which is the average error for finding flow size of all the flows in a trace.

First, we compare different FMU techniques. Figure 2 presents the average error rates for the four FMU techniques and varying table size S=500 to S=16000 entries. We set the number of parallel hash functions to N=4. The CEFMU technique has the best overall performance for varying configurations. For the largest setup, N=4 and S=16,000, HYFMU gives the best performance with an error rate of 7.3% obviously because it employs both MIFMU and CEFMU methods.

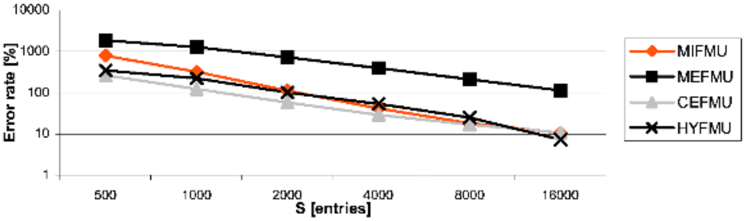


Fig. 2. Average error rates of various FMU techniques.

4.1 Sensitivity Analysis

For sensitivity analysis, we fix the total size for the hash tables (S) and vary the number of parallel hash functions (N) to find the optimal parallelism. The results for S_{total}=32K entries are presented in Figure 3. For all techniques except MEFMU, increasing the number of parallel hash functions initially results in a reduction in the error rate. Particularly, for S_{total}=32K, the error rate reduces from 69% to 17%, from 59% to 11%, and from 63% to 19% for MIFMU, CEFMU, and HYFMU, respectively. The CEFMU technique, on the other hand, improves its performance slightly until N=4. This is because CEFMU is more immune to collisions, which occur more frequently in smaller hash tables. CEFMU is designed to perform well under these circumstances.

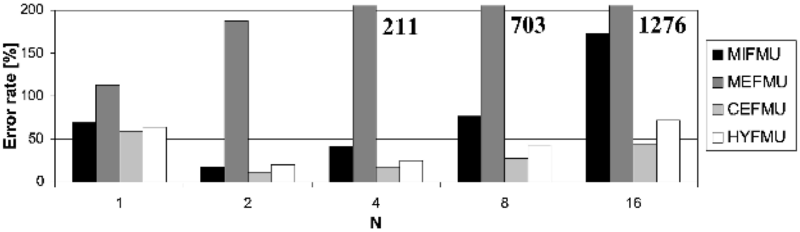


Fig. 3. Error rates for a total table size S_{total}=32K.

5 Conclusions

Flow monitoring is an important task in computer networks. However, almost all techniques are implemented in software and designing hardware for them is very hard if not impossible. With the increase in the link speeds and the wider usage of flow information, hardware flow monitoring is becoming essential for most state-of-the-art routers. In this paper, we presented a hardware flow monitoring design implemented on FPGAs. The FMU unit is based on two-dimensional hashing. With hashing, it has two major advantages over alternative techniques. First, high speeds can be achieved. And second, the access latency to any data is constant. Clearly, any inaccuracies are a result of depending on the performance of the hash function. Multiple hash functions running in parallel on an FPGA alleviates the inherent accuracy penalty. We have applied four different techniques to address this problem. The best technique (HYFMU), which combines CEFMU and MIFMU, can process 200M packets per second (corresponding to a link speed of 64 Gbps for 40-byte packets), while having an average error rate of 7.3%.

References

1. Arsham, H. Time Series Analysis and Forecasting Techniques, http://obelia.jde.aca.mmu.ac.uk/resdesgn/arsham/opre330Forecast.htm
2. Carter, J. and M. Wegman, *Universal classes of hash functions.* Journal of Computer and System Sciences, 1979, 18: p. 143--154.
3. Cheung, O. Y. H., P. H. W. Leong. Implementation of an FPGA Based Accelerator for Virtual Private Networks. In *IEEE International Conference on Field-Programmable Technology (FPT' 02)*, Dec. 2002. Hong Kong, China.
4. B. Jenkins, Hash Functions and Block Ciphers, http://www.burtleburtle.net/bob/hash/index.html
5. C. Madson, , L. Temoshenko, C. Pellecuru, B. Harrison, and S. Ramakrishnan, *IPSec Flow Monitoring MIB Textual Conventions*. Mar. 2003, Internet Engineering Task Force.
6. NLANR Project. Network Traffic Packet Header Traces, http://moat.nlanr.net/Traces
7. Synplicity Inc. Synplify Pro: The Most Powerful HDL Synthesis Solution for Multi-Million Gate Programmable Logic Designs, 2000, www.synplify.com
8. Xilinx Inc., *SPEEDRouter v1.1 Product Specification*. Oct. 2001.

A VHDL Generator for Elliptic Curve Cryptography

Kimmo Järvinen, Matti Tommiska, and Jorma Skyttä

Helsinki University of Technology
Signal Processing Laboratory
Otakaari 5 A, FIN-02150, Finland
{kimmo.jarvinen,matti.tommiska,jorma.skytta}@hut.fi

Abstract. A VHDL generator called SIG-ECPM is presented in this paper. SIG-ECPM generates fully synthesizable and portable VHDL code implementing an elliptic curve point multiplication, which is the basic operation of every elliptic curve cryptosystem. The use of automated design flow significantly shortens design times and reduces error proneness.

1 Introduction

Elliptic Curve Cryptography (ECC) has been of much interest in the world of cryptography during the past few years, because a high level of security can be achieved with considerably shorter keys than with conventional public-key cryptography, e.g. RSA [1]. Because of the short keys, ECC is an attractive alternative in applications, where bandwidths or memory resources are limited [2]. The basic operation of any elliptic curve cryptosystem is the Elliptic Curve Point Multiplication (ECPM) defined as $Q = kP$ where Q and P are distinct points on an elliptic curve E and k is a large integer [1].

Automation of the design flow significantly shortens design times and reduces error proneness of the process. A Hardware Description Language (HDL) generator called SIG-ECPM is presented in this paper. SIG is an acronym for the Signal Processing Laboratory at Helsinki University of Technology. SIG-ECPM generates fully synthesizable and device independent VHDL-code implementing ECPM, when a set of parameters is given. Use of SIG-ECPM is beneficial in applications where several designs have to be implemented in short time. The research work was performed in the GO-SEC project at HUT (see go.cs.hut.fi).

2 The VHDL Generator

There is a large variety of recommended curves which can be used. Because the design process of an ECPM implementation is complex, it would be too complicated to implement every design by hand. Thus, a VHDL generator, which automatically generates synthesizable VHDL-code implementing an ECPM, was designed. This generator is called SIG-ECPM and it was written in the C programming language.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1098–1100, 2004.

ECPM architecture utilized by SIG-ECPM is presented in [3]. The architecture implements entire ECPM while traditionally only Galois field operations are implemented. It uses the point multiplication algorithm developed by López and Dahab in [4]. An inverter proposed by Shantz in [5] was used for Galois field inversions. Multiplier structure was optimized especially for the 4-to-1-bit LUT structure of FPGAs. Details of the multiplier architecture are also presented in [3]. To achieve maximum performance, the ECPM architecture was designed so that the Galois field cannot be changed in the design, i.e. non-reconfigurable designs are created [3]. Thus, a different design has to be designed for every Galois field, which justifies the use of a VHDL generator. The use of SIG-ECPM vastly reduces the disadvantages of non-reconfigurable implementations.

Parameters are given for SIG-ECPM and it generates VHDL-code, which implements ECPM on a curve described by the parameters. The parameters are given as a parameter file of about 25 lines of text. SIG-ECPM generates over 7000 lines of VHDL-code into 26 files. Thus, it is obvious that SIG-ECPM saves a considerable amount of design time. An example of a parameter file, implementing a curve called sect113r1 recommended by SECG (Standards for Efficient Cryptography Group) in [6], is presented in Fig. 1.

```
**** PARAMETER-FILE FOR SIG-ECPM ****
*** Curve Sect113r1

*** Field Specifications.
FIELD {
    m: 113;
    irrpolycoeffs: 3;
    irreducible: 113 9 0;
};

*** Elliptic Curve Specifications.
ECC {
    fixedP: 1;
    Px: 09D73616F35F4AB1407D73562C10F;
    Py: 0A52830277958EE84D1315ED31886;
    fixedC: 1;
    a: 03088250CA6E7C7FE649CE85820F7;
    b: 0E8BEE4D3E2260744188BE0E9C723;
    tvectors: 5;
};

*** Multiplier Specifications.
MULT {
    multipliers: 2;
    latency: 12;
    tvectors: 5;
};

END;
```

Fig. 1. A parameter file

Parameters of Fig. 1 consist of field, elliptic curve and multiplier specifications. The field specifications define the Galois field over which the elliptic curve E is defined. In Fig. 1, the field is $GF(2^{113})$ and the irreducible polynomial generating this field is $m(x) = x^{113} + x^9 + 1$. The elliptic curve specifications define the elliptic curve E and point P. These values are optional for SIG-ECPM, but fixing E and P results in slightly faster and smaller implementations. The number and structure of Galois field multiplier(s) is defined in the multiplier specifications. The use of a second multiplier speeds up ECPM calculation significantly [3]. Latency constraint sets an upper bound for the latency of a single Galois field multiplication, which is the critical operation of ECPM. SIG-ECPM optimizes multiplier so that it satisfies this constraint.

3 Results

Several ECPM implementations using elliptic curves recommended by the Standards for Efficient Cryptography Group (SECG) in [6] were created with SIG-ECPM. Very competitive results were obtained, e.g. a SIG-ECPM design calculates an ECPM on sect163r1 curve in 106 μs when Xilinx Virtex-II XC2V8000-5 FPGA was used as a target device. The design flow used for the designs is presented in Fig. 2 with typical run-times.

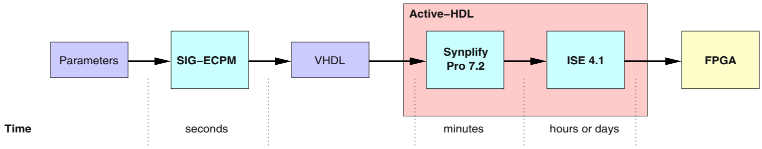


Fig. 2. Design flow with typical run-times for the designs created with SIG-ECPM

The benefits of SIG-ECPM are obvious when the time required for the design process is considered. It would require days to make the entire design by hand. With SIG-ECPM, the same work can be done in a couple of seconds. However, the synthesis and implementation (place & route) still require a considerable amount of time. The real bottleneck of the design process is the implementation, which requires time from hours to even days. Anyhow, design time can be significantly shortened with a VHDL-generator, such as SIG-ECPM.

4 Conclusions

A VHDL generator called SIG-ECPM was presented in the paper. SIG-ECPM generates fully synthesizable and portable VHDL code implementing an ECPM. The use of a VHDL generator speeds up the design process significantly and reduces error proneness. VHDL generators are very useful in ECC applications, where a large variety of different parameters (curves, Galois fields, etc.) are used. They reduce the disadvantages of fast non-reconfigurable ECPM implementations, thus, allowing faster performance.

References

1. Blake, I., Seroussi, G., Smart, N.: Elliptic Curves in Cryptography, London Mathematical Society Lecture Note Series 265. Cambridge University Press (2002)
2. Bednara, M., Daldrup, M., von zur Gathen, J., Shokrollahi, J., Teich, J.: Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'02, 9th Reconfigurable Architectures Workshop, RAW-2002, Marriott Marina, Fort Lauderdale, Florida, USA (2002) 157 – 164
3. J rvinen, K., Tommiska, M., Skytt , J.: A Scalable Architecture for Elliptic Curve Point Multiplication. Submitted to the International Conference on Field Programmable Technology, FPT 2004 (2004)
4. López, J., Dahab, R.: Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation. Proceedings of Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, LNCS 1717, Springer-Verlag (1999) 316 – 327
5. Shantz, S.C.: From Euclid's GCD to Montgomery Multiplication to the Great Divide. Technical Report SMLI TR-2001-95, Sun Microsystems Laboratories (2001)
6. Certicom Research: SEC 2: Recommended Elliptic Curve Domain Parametres. Standards for Efficient Cryptography (2000) URL:
`http://www.secg.org/collateral/sec2_final.pdf`, (visited: June 3, 2004).

FPGA-Based Parallel Comparison of Run-Length-Encoded Strings

Alessandro Bogliolo[1], Valerio Freschi[1], Filippo Miglioli[2], and Matteo Canella[2]

[1] STI - University of Urbino, Urbino 61029, Italy,
{bogliolo,freschi}@sti.uniurb.it
[2] DI - University of Ferrara, Ferrara 44100, Italy

Abstract. The length of the longest common subsequence (LCS) between two strings of M and N characters can be computed by $O(M \times N)$ dynamic programming algorithms that can execute in $O(M + N)$ on a linear systolic array. If the strings are run-length encoded, LCS can be computed by an $O(mN + Mn - mn)$ algorithm, called RLE-LCS, where m and n are the numbers of runs of the two strings. In this paper we propose a modified RLE-LCS algorithm mappable on a linear systolic array.

The similarity between two strings X and Y of M and N characters ($X = x_1, x_2, ..., x_M$ and $Y = y_1, y_2, ..., y_N$) can be expressed in terms of the length of their *longest common subsequence* (LCS), that can be computed by an $O(M \times N)$ dynamic programming algorithm [1] that implements the following recursion:

$$LCS(i,j) = \begin{cases} \max\{LCS(i-1,j), LCS(i,j-1)\} & x_i \neq y_j \\ LCS(i-1,j-1)+1 & x_i = y_j \end{cases} \tag{1}$$

$LCS(i,j)$, representing the LCS between the first i characters of X and the first j characters of Y, is incrementally computed (for $i, j > 0$) from $LCS(i-1,j), LCS(i-1,j-1)$ and $LCS(i,j-1)$. The entries of the dynamic programming matrix LCS are shown in Fig. 1.a for a simple example. The inherent parallelism of the algorithm and its regular structure can be exploited for computing LCS in $O(M+N)$ steps by means of a linear systolic array [2], as shown in Fig. 1.b. All matrix entries on the same column are computed by the same unit in subsequent steps. The relative data dependencies of each unit do not change over time.

If X and Y are *run-length-encoded* (RLE) strings [3] of m and n runs, the encoding induces a partitioning of the LCS matrix into sub-matrices (*blocks*) associated with ordered pairs of runs (Fig. 1.c). With respect to a block B we call: *input elements* the entries of LCS that do not belong to B but provide inputs to some of its elements, *root element* the input element with smallest row and column indexes, *output elements* the elements of B that feed entries of LCS that do not belong to B, *inner elements* all other elements of B.

Any output element $LCS(i,j)$ of a block B rooted in $LCS(i0,j0)$ can be incrementally computed directly from the input elements of B [4]:

$$LCS(i,j) = \begin{cases} \max\{LCS(i0,j), LCS(i,j0)\} & x_i \neq y_j \\ LCS(i-d,j-d)+d & x_i = y_j \end{cases} \tag{2}$$

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1101–1103, 2004.

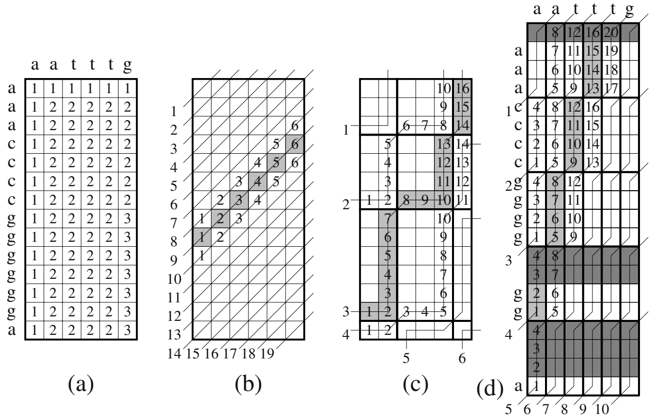


Fig. 1. Basic dynamic programming algorithm for LCS: a) 14x6 LCS matrix; b) Computation steps on a systolic array of 6 elements; c) Binding between computation tasks and systolic array elements; d) $RLL_{(4,1)} - LCS$ matrix.

where $d = \min\{x - x0, y - y0\}$. The algorithm based on eq. 2 (called RLE-LCS) has complexity $O(mN + Mn - mn)$ and improved parallelism that enables computation in $O(m + n)$ steps (as shown in Fig. 1.c). However, the degree of parallelism and the data dependencies among computational units change at each step, making it difficult to map RLE-LCS on a linear systolic array.

To enforce a regular structure of data dependencies we decompress the shorter string (say Y) and we impose an upper bound (L_{Max}) to the length of the runs of the longest string (say, X), thus obtaining a modified algorithm (denoted by RLL$_{(L_{Max},1)}$-LCS) that executes in $O(\tilde{m} + N)$, where $\tilde{m}$ is the number of runs of X with maximum length limited to L_{Max}. Upper bounds limit the size of the blocks associated with run pairs, making it possible to allocate a maximum-size block to each pair of runs, obtaining an oversized matrix of $\tilde{m}L_{Max} \times N$ entries (shown in Fig. 1.d for $L_{Max} = 4$) with regular structure. We align each run of X to the end of the corresponding block, adding gaps to all runs shorter than L_{Max}. Matrix entries associated with gaps (shaded in figure) do not need to be computed.

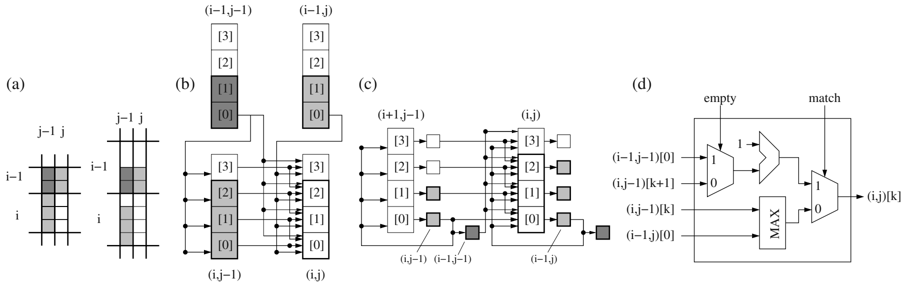


Fig. 2. a) Generic block in position (i, j) of basic and extended LCS matrixes. b) Local connections of block (i, j). c) Mapping of RLL$_{(L_{Max},1)}$-LCS algorithm on a linear systolic array. d) Schematic of the basic computational unit.

The L_{Max} elements of each block are indexed from 0 to $L_{Max} - 1$. We use $(i, j)[k]$ to denote element k of $B(i, j)$, associated with the i-th run of X and the j-th run of Y. Figure 2.a shows the position of $B(i, j)$ in the original and extended LCS matrix, while Figure 2.b shows its data dependencies. In case of a match between x_i and y_j, element $(i, j)[k]$ is computed by adding 1 to the value found either in $(i, j-1)[k+1]$ or in $(i-1, j-1)[0]$, depending on whether or not the i-th run of X is longer than $k+1$. In case of mismatch, the value of $(i, j)[k]$ is the maximum between $(i, j-1)[k]$ and $(i-1, j)[0]$. The interconnect structure shown in Figure 2.b is general enough to support all possible situations.

The mapping on a linear systolic array is shown in Figure 2.c, where only two blocks are shown and the computational units associated with the same block are vertically aligned for readability. At a generic step, the right-most elements are processing block (i, j), while the left-most elements are processing block $(i+1, j-1)$. The inputs of $B(i, j)$ have been computed by the same processing elements during the last two execution steps. Hence, output values need to be stored in registers to be made available for subsequent computations. Memory elements are represented in Figure 2.c by small squares. All outputs need to be stored for one clock cycle, except the output of element 0 that needs to be stored for 2 cycles.

The RTL schematic of the computational unit associated with each element is shown in Figure 2.d. Signal *match* is a Boolean flag that represents the result of the comparison between x_i and y_j. Signal *empty* is a Boolean flag raised whenever entry $(i, j-1)[k+1]$ is empty. The comparator is not represented within the basic computational element, since it is shared by all elements of the same block.

We used a Xilinx Virtex XCV300E FPGA to implement the RLL$_{(L_{Max},1)}$-LCS algorithm for L_{Max} ranging from 1 to 10. We used 2 bits to encode the characters and 8 bits to encode the score (i.e., the length of the LCS computed by the algorithm). For all implementations we obtained maximum clock frequencies in the range from 111MHz to 123MHz, the variation depending only on different routings. The speedup achieved w.r.t. LCS depends on the effectiveness of the compression provided by the RLL encoding of X, that depends on the limit L_{Max} imposed. Analyzing the run-length distribution of DNA sequences and bitmaps representing scanned handwritten texts, we found that $L_{Max} = 5$ is sufficient to exploit 99% of the compression ratio of RLE.

References

1. T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 48 (1981), 195-197.
2. D. Lopresti, P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences, Computer 24 (11) (1987), 6-13.
3. K. Sayoood, E. Fow editors, Introduction to Data Compression, 2nd edition, Morgan Kaufmann (2000).
4. V. Freschi, A. Bogliolo, Longest Common Subsequence between Run-Length-Encoded Strings: a New Algorithm with Improved Parallelism, Information Processing Letters 90 (2004), 167-173.

Real Environments Image Labelling Based on Reconfigurable Architectures

Juan Manuel García Chamizo, Andrés Fuster Guilló, and Jorge Azorín López

U.S.I. Informática Industrial y Redes de Computadores. Dpto Tecnología Informática y Computación. Universidad de Alicante. Apdo. Correos 99. E-03080. Alicante. Spain
{juanma, fuster, jazorin}@dtic.ua.es, http://www.ua.es/i2rc

Abstract. In this paper a vision system of images perceived in real environments based on reconfigurable architecture is presented. The system provides surface labelling of the input images of unstructured scenes, irrespective of the environmental lighting or scale conditions and the capture frequencies. The nucleus of the system is based on querying SOMs constructed with supervised training by means of descriptors extracted from images of different surfaces perceived for successive values of the optical parameters (lighting, scale). To improve the labelling process the system estimates the environmental optical parameters and then SOMs are reconfigured. A segmented architecture is proposed for the central module of the labelling process to improve the performance of image sequences. A prototype implemented on FPGAs applied as a guidance aid for vehicles is provided.

1 Introduction

Within the framework of the research project DPI2002-04434-C01-01 the overall objective can be broken down into two partial aims: to provide a vision model of unstructured scenes in realistic conditions and an architecture that fulfils the performance expectations. Vision problems in real conditions (lighting, scale, angle...) have been discussed in literature at low and intermediate level [1] [2]. The objective of providing a vision model and its application in scene labelling using techniques that query databases with non-specific descriptors are discussed in [3] [4]. The wide consolidation of the self-organizing model [5] and the implementation possibilities at a low level have been the criteria in its choice as classifier. The brightness histogram was selected due to its generality and tolerance with geometrical transformations [6]. For the simplification of database queries, the use of context information is proposed [7]. The objective that is discussed in greater depth in this article is the performance. Among the digital implementations of the SOM model we can find two fundamental trends, systolic arrays and SIMD [8], which have inspired the proposal for the segmented design. The capacity for rapid prototyping of reconfigurable computation has been used [9]. The prototype has been tested as an outdoor guidance aid for vehicles based on surface labelling of the sequences of the incoming images.

[1] This study is part of the research carried out in the project MCYT. DPI2002-04434-C01-01.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1104–1106, 2004.

2 Vision System Architecture

The architecture proposed in this paper is based on the model developed in [3][4]. The model proposes a preprocessing phase in order to estimate the context (lighting conditions), followed by a processing phase that performs image labelling in real environments querying databases. The parallelization possibilities depend both on the descriptors and on the implementation resources used. In addition, continual pre-processing is not necessary since the real values (lighting conditions) only change at specific time intervals. This way, pre-processing is only performed after specific time intervals. If conditions change, a hardware reconfiguration of the central part of the processing should be necessary. In the processing phase, a hybrid architecture (Fig. 1) has been proposed. The histogram modules are organized according to a SIMD schema and the neuron modules in a segmented one. The prototipe is implemented on a Celoxica RC1000 prototyping card. The vision system structure is shown in figure 1. It is made up of four functional units: *memory, host, uc, histogram and som.*

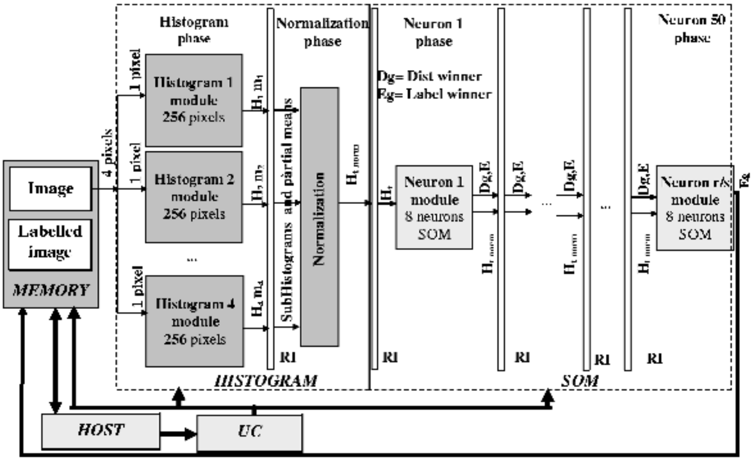


Fig. 1. Prototype structure based on reconfigurable hardware

The following lines are devoted to system functioning. First, a 1024x1024 8 bit image is captured. This image is transferred via DMA to the system memory and control is delegated to the *uc*. Each segmentation step calculates a 32x32 pixel window of the input image. The *histogram* is made up of two segmentation phases: the *histogram phase* (*hp*) and the *normalization phase* (*np*). In the *hp*, 4 modules are organized to calculate the accumulation of the histogram frequencies (subhistograms) in 32 grey levels and the sum of the pixel values in order to obtain the mean for each subwindow. The *np* adds up the subhistograms calculated as well the partial sums of the mean. In addition, it shifts the histogram frequencies to the central component and adds the mean to the last component of the histogram. After two segmentation cycles have passed, the *som* unit receives the normalized histogram of a sample window. This unit calculates the window label, which will be stored in the memory using 50 *neuron* modules (in this prototype, a 400-neuron map has been used). Each module is designed to query sequentially the winning neuron to a subset of 8 neurons of the SOM. The Manhattan distance between two vectors is calculated to give the minimum distance. Finally, the labelled image is transferred from the *memory* to the *host*.

The labelling success rates of the system are calculated by comparing the results obtained with those provided in the previous supervised labelling stage [3][4]. In the performance tests, the system works at a clock frecuency of 40 MHz (clk) and occupies 17,334 slices in the FPGA. A comparative test has been made between the prototipe and software implemention for AMD (Athlon 1.7 GHz) and Intel (Pentium IV 1.7 GHz). The prototype that is mounted in a vehicle is able to calculate from 2.52 to 130.71 images per second (images of 1024x1024 for different sample frequencies, distances in pixels between samples). However, the software implemetation is able to calculate from 0.05 to 3.56 and 0.05 to 3.80 for the AMD and Intel respectively.

3 Conclusions

A system for real scene labelling has been presented. The system provides the labelling of unstructured scenes irrespective of the lighting and scale capture conditions and the frequencies required. The system is based on a model for vision in real conditions that stores descriptors of images perceived with successive values of the optical parameters. The model proposes a preprocessing phase in order to estimate the context and lighting conditions, followed by a processing phase where the architecture reconfiguration takes place and partial views of the databases can be queried to improve labelling. These databases are implemented by means of self-organizing maps that organize brightness histograms extracted from the images. A FPGA based prototype has been provided and tested. The processing frequencies obtained enable the system to be applied as a guidance aid for vehicles in outdoor scenarios leaving the specific use of the labelling open (land maps, route planning…).

References

1. Flusser J., and Suk T.: Degraded Image Analysis: An Invariant Approach. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 20, No. 6. June 1998.
2. Malik J., Belongie S., Leung T., and Shi J.: Contour and Texture Analysis for Image Segmentation. IJCV 43(1).2001.
3. García, J.M., Fuster, A., Azorín, J. and Maciá, F.: Architecture for Image Labelling in Real Conditions. ICVS 2003. Lecture Notes In Computer Science. Springer-Verlag.
4. García, J.M., Fuster, A., Azorín, J.: Image Labelling in Real Conditions. Kybernetes (The International Journal of Systems & Cybernetics) Ed: MCB University Press. Vol.33. 2004.
5. Kohonen, T.: Self-Organizing Maps. Springer-Verlag, Berlin Heidelberg, 1995.
6. Hadjidemetriou, E., Grossberg, M. D., Nayar, S. K.: Histogram Preserving Image Transformations. International Journal of Computer Vision 45(1). October 2001.
7. Strat, T., and Fischler, M.: The Role of Context in Computer Vision. In Proceedings of the ICCV. Workshop on Context-Based Vision.1995.
8. Hammerstrom, D. and Nguyen, N.: An Implementation of Kohonen's Self-Organizing Map on the Adaptive Solutions Neurocomputer. Artificial Neural Networks. pp. 715-719. 1991.
9. Ratha N.K. and Jain A.K.: Computer Vision Algorithms on Reconfigurable Logic Arrays. IEEE Transactions on Parallel and Distributed Systems, 10(1):29-43, 1999.

Object Oriented Programming Paradigms for the VHDL

Jan Borgosz

AGH - University of Science and Technology
30-059 Krakow, Poland
borgosz@agh.edu.pl

Abstract. Nowadays, the object oriented technology can be met in different programming languages, also in the hardware description languages (HDLs). Unfortunately, in the last case the situation is more complicated. Popular and wide known tools do not support the object oriented technology, which has to be incorporated in designs with additional overlays. These overlays are academic solutions, or rather, it is hard to use them in real and complicated designs. So, we have developed a simply technique for building designs in the object oriented style. This technique involves well known tools of well known vendors and the set of the coding rules. The additional overlays are not needed. The presented solution was successfully tested with many projects and tools and gave very promising results. Due to lack of the space, this paper should be treated as a report rather than a detailed description.

1 Introduction

The group of the hardware description languages has been standardized since 1987 [7]. These standards are developed until now. Unfortunately, due to this fact, the market of the hardware programming tools is divided in two parts: the well known commercial part and the unknown part related to research centers. We want to join these groups of the interest together using our solution.

We started our research with studies on available tools and solutions [5][6]. We found some interesting projects, like: Handel-C [11], the Aldec and Celoxica ANSI-C Software Environment [4], the SUAVE project, the SAVANT project, the REQUEST [9] project and JHDL. Some of these tools are high level programming languages without the object oriented background. The other projects are highly specialized overlays, which are not integrated with commercial tools. Their common feature is lack of the conjunction with practice [12]. Generated RTL is often oversized and not optimized [11]. So, we did not find any ideal solution as a result of our search.

Our solution combines simplicity and utility. It can be used for simple but also for complicated projects and may be easily incorporated into standard HDL environments for free. The quality of the generated RTL depends only on the quality of the vendor tools.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1107–1109, 2004.

2 The Description of the Object Oriented Paradigms

We present the short description of the paradigms in this chapter. Our idea may be built with four “puzzles”. First of all we divide the design into segments. Next we have to assure the synchronization between segments. We also will treat all parts of the design as independent objects. Finally, we use the specialized syntax [10].

As we mentioned before, our design is divided into segments. Of course each segment is independent of each other and becomes an object. Also, each segment may be replaced without any changes to neighbor segments. Moreover, each segment is a template, what means it is described with a set of generic parameters. Also, interfaces between segments are described with generic constrains.

Segments are in a queue, so data are passed from the segment to the segment. We delay all signals passing through the segment with the same number of clock cycles, or rather, all signals are aligned in time. So, we do not use the “star” structure for synchronizing segments. In this case, there is not a possibility that signals will be out of sync. Furthermore, all operations are synchronized with the same clock edge.

The next puzzle is the syntax. This problem is marginalized or authors present the simplified approach. Sometime we can find strongly hardware dependent syntax [1]. In our paradigms, the base of each name in our system comes from the description of the function. We add the prefix and the suffix to this base. The prefix reflects a type of a VHDL object when the suffix reflects a type of a FPGA module. The prefix is connected with the VHDL, the suffix is connected with the structure of hardware. The sum of these elements gives us the name of the object [3]. So, in this system the name of the object reflects its function. It is always clear what kind of the object we use.

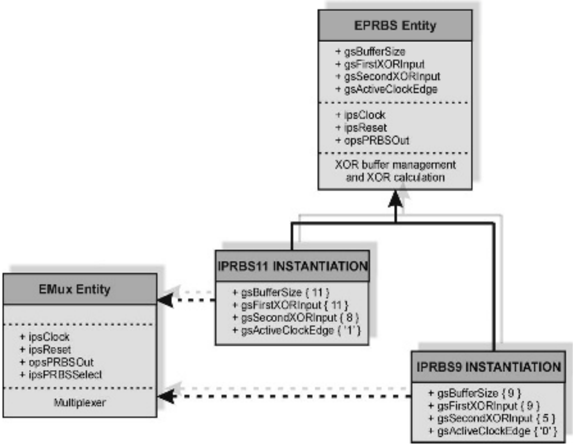


Fig. 1. The example of the UML diagram for the design

The object oriented systems are often described and visualized with UML. We decide to modify standard notation for our purposes [8]. The UML block in adapted notation consists the name of the entity, generic settings, ports and the function. We can make particular visualization of the part of the system and focus our attention on the entity and the derived specialized entities. We can also make general visualization of the system and see not only the deriving relation but also the owning relation then (Fig. 1).

Described rules allow to obtain a very clear code, easy to modify, which is portable between platforms. Figure 1 depicts an example. Here is a system which consists two generators of the pseudo random binary sequence and the multiplexer. So, we have two instantiations IPRBS9 and IPRBS11 derived from the EPRBS entity and the EMUX entity which owns the instantiations [2].

3 Conclusions

The limited number of pages allows to give only a brief overview of the subject and the main ideas are described in general terms. However, it is enough piece of information which shows our object oriented system for VHDL designs. This group of simple rules can be used for complicated projects with very good results. These rules make the code the error-proof and readable structure for everyone, give portability between different vendors, allow to describe problems in the object oriented way without specialized tools, make the significant decrease in time of development.

Authors used presented paradigms for a wide spectrum of complicated projects. The STM-16 framer or the system for the detection of signs on the road are the best exemplars. In these applications we obtained a very promising results for the future and the significant increase in the quality of the code.

This paper is a result of research which is registered in The State Committee for Scientific Research of Poland (KBN) and numbered: 3T11C 045 26. The State Committee for Scientific Research of Poland sponsors this publication.

References

1. Ameseder T., Pühringer T., Wieland G., Zeinzinger M.:VFIR Coding Conventions 1.0., WWW (2003)
2. Ashenden P., Wilsey P.A.: Considerations on Object – Oriented Extensions to VHDL. Proc. VIUF Spring '97 Conference, Santa Clara, CA (1997) 109 –118
3. Borgosz J., Cyganek B.: „Proposal of the Programming Rules for VHDL Designs", Springer LNCS, M. Bubak (Eds.), ICCS 2004, June 6-9, 2004, Krakow, Poland.
4. Celoxica WWW Service
5. Cohen B.: Coding HDL for Reviewability, MAPLD (2002)
6. Gord A.: HDL Coding Rules and Guidelines, WWW (2003)
7. IEEE: IEEE Standard - VHDL, (1987)
8. McUmber W. E., Cheng B.H.C.: UML-Based Analysis of Embedded Systems Using a Mapping to VHDL. 4th IEEE International Symposium on High-Assurance Systems Engineering, Washington, D.C. (1999) 56-64
9. OFFIS and Politecnico di Milano. Prepared by Martin Radetzki, Wolfram Putzke-Röming, Wolfgang Nebel Language architecture document on Objective VHDL REQUEST - Reuse and Quality Estimation ESPRIT Project 20616
10. Oliver I.: Model Driven Embedded Systems, Nokia Research Center, Finland (2003)
11. Ortigosa E.M., Ortigosa P.M., Cañas A., Ros E., Agís R., Ortegaı J.: FPGA Implementation of Multi-layer Perceptrons for Speech Recognition. Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York (2003) 1048 – 1052
12. Swamy S., Molin A., Covnot B.: OO-VHDL. Object-oriented extensions to VHDL. Computer, Vol. 28, Issue: 10., Vista Technol., Schaumburg, IL, USA (1995) 18-26

Using Reconfigurable Hardware Through Web Services in Distributed Applications*

Ivan Gonzalez, Javier Sanchez-Pastor, Jorge L. Hernandez-Ardieta,
Francisco J. Gomez-Arribas, and Javier Martinez

Escuela Politecnica Superior, Universidad Autonoma de Madrid, Cantoblanco E-28049
Madrid, Spain
{Ivan.Gonzalez, Javier.Sanchez, Jorge.Lopez, Francisco.Gomez,
Javier.Martinez}@ii.uam.es

Abstract. This paper proposes a simple solution to use reconfigurable hardware in the context of distributed applications. The remote access to the reconfigurable resources is carried out through Web Services technology. So it is possible to exploit the synergy of reconfigurable computing and distributed applications. A web service has been developed to remotely use the whole functionality of a reconfigurable platform. An example application has been developed in order to study the advantages and drawbacks of this methodology.

1 Introduction

Nowadays Internet allows the remote use of different reconfigurable platforms offered in Web laboratories [1, 2] accessible from anywhere at any time. A step further will be extending the use of reconfigurable hardware, connected to Internet, to Distributed Computing applications. Both Distributed and Reconfigurable computing are quickly gaining in popularity. The aim is exploit the synergy of these two rather complementary technologies.

Currently distributed applications are based on PC computers organized as clusters embedded in a communication infrastructure. On the other hand, there are several implementation of clusters based on reconfigurable hardware [3, 4]. These previous works present very complex architectures. This paper presents a simple solution to use reconfigurable hardware in the developing process of Distributed Applications based on communication networks. Web Services [5] technology is proposed to use the reconfigurable platform RC1000PP of Celoxica. A Web Service offers a set of functions with a specific task through a network, usually Internet. The communication between the different parts of a distributed application is established using this technology.

[*] This work has been supported in part by the European Program No: 100671-CP-1-2002-1-FR-MINERVA-M, and in part by Project 07T/0052/2003-3 Comunidad de Madrid, Spain.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1110–1112, 2004.

2 Using Reconfigurable Hardware Through Web Services

The reconfigurable platform RC1000PP of Celoxica is a PCI board built around a XCV2000-E FPGA. A local C library allows performing all necessary operations over the platform from a PC: configure the FPGA, set the clock frequency, exchange data between the FPGA design and the PC host using shared memory, etc.

A first Web Service provides remote access to the FPGA design tools. It is accessible through the web application shown in figure 1a. This web tool allows the user to run the different design stages.

An additional Web Service implements all functions contained in the C library. The formal use of these functions is similar to the original, with the same function name and input/output parameters. This makes possible to adapt easily the application execution from local to remote, changing the library functions calls by their remote version. As an example, a local application developed in C language to encrypt/decrypt using the DES algorithm has been adapted to be used by the platform through the developed web service. Two different languages have been used, Python for a window client interface, and PHP for a web client interface (figure 1b). Both programs are identical, except the differences imposed by the syntax languages and the implementation of the SOAP [5] library. This is possible because Web Services are independent of the language used in the application that requests the service.

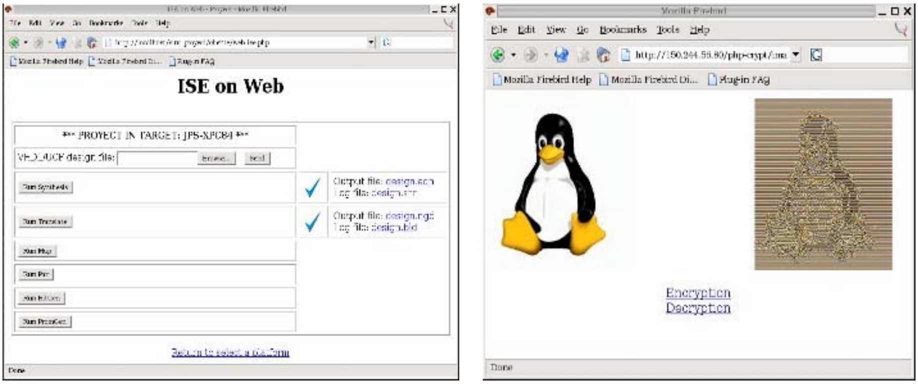


Fig. 1. (a) Remote access to design tools. (b) Web client interface of the cipher application

Using the RC1000PP platform through a web service allows a user, who doesn't have this platform, to design applications and test them. However, in order to increase the performance of distributed applications, the use of the remote library functions could present drawbacks for some specific applications. When a function is invoked, the execution time is the result of adding the time wasted to execute the function locally to the time spent sending a receiving the data. So the final application has a penalization due to the remote execution. To minimize this situation it is necessary to consider the following methodology in order to design new web service functions:

- The library functions offered by the web service are used as a first approach to design and debug quickly the applications. On this way, the designer that knows the library functionality implements the application easily. The performance of this first approach is low.
- Once the applications work correctly, new functions can be added to the web services. These new functions must implement an upper level on the platform management and they are related to the specific application. This reduces the number of remote calls, so the execution time is smaller and the performance is better.
- Finally, if the performance is not good enough, a new remote function that implements all the application functionality is necessary to be developed. For example, a remote function for encrypt or decrypt using a specific algorithm. This new approach reduces the network communication activity and the performance could be the best one.

3 Conclusions and Future Work

Web Services technology is presented as a solution to integrate functions implemented in reconfigurable hardware, with an application developed using high level programming languages. The application developer can improve the execution time of critical tasks, without the knowledge of how the reconfigurable hardware works. The web service technology allows the remote use of a specific reconfigurable platform and offers all hardware functions to be used. Although this is a previous work, the preliminary results give rise to many appealing opportunities.

The web service presented in this work requires that the reconfigurable platforms are included in PC architecture. However, the new reconfigurable platforms available with embedded microprocessors will allow implementing the web service in the same board. The use of the Virtex II Pro family to implement web services will be carried out as future work. Another experiment will study the throughput improvement expected in a distributed applications based on the use of reconfigurable hardware through Web Services.

References

1. S. Hauck, “The Future of Reconfigurable Systems”, Proceedings of the 5th Canadian Conference on Field Programmable Devices, June 1998.
2. Ivan Gonzalez, Ruben Cabello, Francisco Gomez-Arribas, Javier Martinez "Labomat-Web: A Web Laboratory Based on Reconfigurable Computing Technology" *International Conference on Engineering Education (ICEE 2003)*, July 21–25, 2003, Valencia, Spain.
3. J. M. Lehrter, F. N. Abu-Khzam, D. W. Bouldin, M. A. Langston and G. D. Peterson, “On Special-Purpose Hardware Clusters for High-Performance Computational Grids”, *Proceedings International Conference on Parallel and Distributed Computing and Systems*, 2002
4. D. F. Newport and D. W. Bouldin, “Using Configurable Computing Systems”, *Computer Engineering Handbook*, V. Oklobdzija, editor, CRC Press, 2001.
5. Web Service Activity (http://www.w3c.org/2002/ws/)

Data Reuse in Configurable Architectures with RAM Blocks

Extended Abstract

Nastaran Baradaran, Joonseok Park, and Pedro C. Diniz

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001, Marina del Rey,California 90292, USA,
{nastaran,joonseok,pedro}@isi.edu

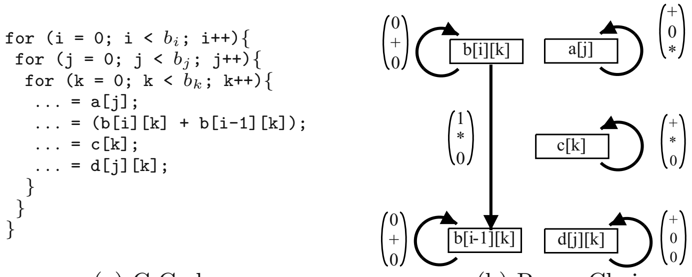
The heterogeneity of modern configurable devices makes the problem of mapping computations to them increasingly complex. Due to the large number of possibilities for partitioning the data among storage modules, these architectures allow for a much richer memory structure. One general goal in managing this memory is to minimize the number of external memory accesses. A classic technique for reducing this number is to keep reusable data as close to the processor as possible. In order to do so one needs to have a good idea as to *if and when* the data in a specific memory location is going to be reused. General compilers are capable of detecting the reuse as well as applying different techniques in order to exploit this reuse. In this paper we describe how to utilize a compiler reuse analysis to map the data to a configurable system, aiming at minimizing the number of external memory accesses. Our target architecture is a system with an external memory, a *limited* number of internal registers, and a *fixed* number and capacity of internal RAM blocks.

A compiler technique for exploiting the reuse in array variables is known as *scalar replacement* or *register promotion*. For array data values that are repeatedly accessed across many iterations of a loop nest, *scalar replacement* allows a compiler to replace an array reference by a scalar. The compiler then saves this scalar in a register or an internal RAM block in order to reuse it in later loop iterations. As a result of this "caching", scalar replacement substantially reduces the number of memory operations. The fundamental issues here are to first identify the data reuse of a memory location, and then to determine the number of registers required by scalar replacement for capturing this reuse.

In order to identify the reuse of an array reference, the compiler makes use of the data dependence analysis information embedded in distance vectors. Two array references are data dependent if they access the same element either by writing or by reading. The data dependence relation between different references could be presented as a directed graph, in which each connected component is called a *reuse chain*. Also, the distance between dependent accesses is captured in a *distance vector*[1]. In this vector each element represents the iteration count difference between a reference and its reuse for the corresponding loop level. Figure 1(b) illustrates the distance vectors and reuse chains for the example code

[1] The elements of this vector are either a constant (constant difference), a '+' (all positive differences), or '*' (unknown difference).

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1113–1115, 2004.



```
for (i = 0; i < b_i; i++){
 for (j = 0; j < b_j; j++){
  for (k = 0; k < b_k; k++){
   ... = a[j];
   ... = (b[i][k] + b[i-1][k]);
   ... = c[k];
   ... = d[j][k];
  }
 }
}
```

(a) C Code.

(b) Reuse Chains.

Fig. 1. Example Code.

presented in Figure 1(a). For instance the elements of array `c[k]` are reused across the iterations of i and j loops. The corresponding distance vector of $(+, *, 0)$ means that an array location is *first* reaccessed in the next iteration of the i loop, any iteration of the j loop and the same iteration of the k loop. In order for the data accessed by `c[k]` to be reused, the generated code must capture *all the values* between reuses and save them in registers, in this case b_k registers, thereby saving $b_i \times b_j$ memory accesses.

Clearly, exploiting the reuse of a loop nest at the outer levels can require a large number of registers and hence increase the register pressure in a given implementation. In the presence of limited registers, a compiler algorithm can take advantage of local storage such as RAM blocks. Typically, RAM blocks have much higher density than discrete registers at the cost of reduced bandwidth. In this research we are exploring effective compiler algorithms that can strike a balance between the opportunities for data reuse, and the availability and bandwidth of registers and RAM blocks. In order to compute the required number of registers to exploit the reuse at a given level of the loop and for a particular set of reuse chains, the compiler must first select which reuses to exploit. It then needs to stage the allocation and flow of data between registers and RAM blocks for maximum resource efficiency.

We have conducted a preliminary set of experiments for a small set of image processing kernels and present here the results for a a Finite-Impulse-Response (`FIR`) computation code. For this code we have developed a C reference specification and manually translated it to behavioral VHDL for various C code versions exploiting reuse at different loop levels. After converting these C codes into a structural VHDL design using Mentor Graphics' Monet™ high-level synthesis tool, we used Synplify Pro 6.2 and Xilinx ISE 4.1i tool sets for logic synthesis and Place-and-Route (P&R), targeting a Xilinx Virtex™ XCV 1000 BG560 device. We then extracted the real area and clock rate for each design and used the number of cycles derived from simulation of the structural design with the memory latency information of our target FPGA board (read latency of 6 cycles and write latency of 1 cycle) to calculate wall-clock execution time.

In these experiments we have artificially defined a maximum limit of 64 for the number of registers used to capture the data reuse. With this constraint on the number of registers, we applied a simple allocation algorithm to assign the data to registers or RAMs at different loop levels. We then observed the impact of the decision of using RAM vs. registers in the area and performance of the resulting hardware design. In these results, version `v1` denotes the *base* code version in which no reuse is exploited at any level; versions `v2`, `v3` denote the versions where reuse is exploited exclusively using registers at levels 1 and 2 respectively. Finally version `v4` denotes a version which exploits reuse at the top level of the nest, but because of the limitation on the number of registers the implementation uses internal RAM blocks rather than registers.

We present two plots, the first depicting the number of clock cycles of the actual computation as well as the corresponding memory operations (with and without pipelined access mode). In the same plot we present the speedup with respect to the original version (`v1`) using the same memory access mode. The speedup calculation uses the actual clock rate obtained from the hardware implementation after P&R for each design. In the second plot we present the number of registers used to exploit reuse and the FPGA area in terms of slices.[2]

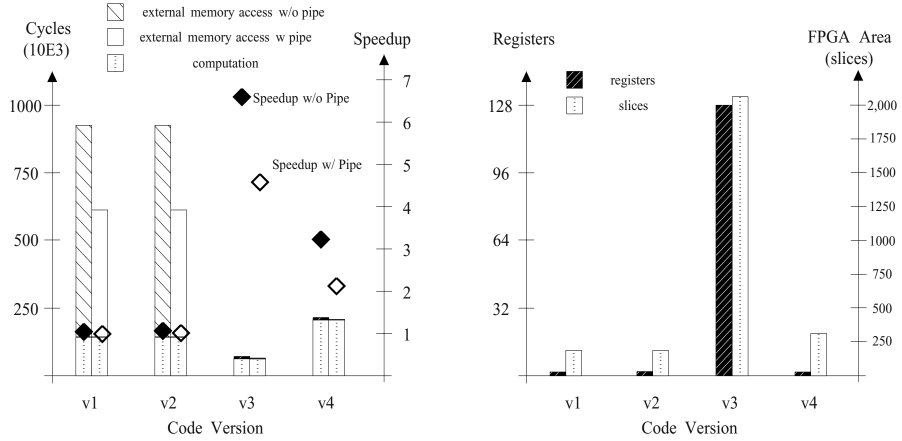


Fig. 2. Experimental Results for `FIR`.

As the results illustrate, exploiting data reuse at all levels of the loops dramatically reduces the number of memory accesses and consequently improves the performance of the implementation. However, this comes at a steep price for the versions that do not use RAM blocks. For example, version `v3` eliminates 77.7% of the original memory operations but the corresponding hardware design uses 16% of the 12, 288 FPGA slice resources. As to the versions that use RAM blocks (`v4`) to exploit the reuse at all levels, they exhibit comparable performance to versions that exclusively use registers (`v3`), while using a very small fraction of the FPGA resources. We also note that the impact of exploiting data reuse is less dramatic when the overhead of the individual memory accesses is reduced by exploiting pipelining techniques.

[2] In the area metric we have excluded the slices due to the RAM blocks, since they are fixed for every design whether the design uses them or not.

A Novel FPGA Configuration Bitstream Generation Algorithm and Tool Development*

K. Siozios, G. Koutroumpezis, K. Tatas, D. Soudris, and A. Thanailakis

VLSI Design and Testing Center, Department of Electrical and Computer Engineering, Democritus University of Thrace, 67100, Xanthi, Greece
{ksiop, gkoytroy, ktatas, dsoudris, thanail}@ee.duth.gr

Abstract. A novel configuration bitstream generation tool for a custom FPGA platform is presented. It can support a variety of devices of similar architecture. The tool exhibits technology independence and is easily modifiable. The tool also allows partial reconfiguration as long as the target platform also does.

1 Introduction and Related Work

In this paper, a configuration bitstream generation tool is introduced. It is part of a complete framework for mapping logic on a custom FPGA platform, starting from a VHDL circuit description down to the FPGA configuration bitstream. This framework was developed as part of the AMDREL project [1]. The tool can operate as a standalone program as long as the necessary input files are provided in the appropriate format, and the architecture of the target FPGA to be configured is supported.

2 Target Fine-Grain Reconfigurable Hardware Architecture

The proposed configuration bitstream generator was developed to support a specific custom FPGA platform, which was presented in detail in [2, 3]. The tool that is presented in this paper was developed in order to support this FPGA, but it can target a variety of architectures.

3 Proposed Configuration Bitstream Generation Tool

DAGGER (DEMOCRITUS UNIVERSITY OF THRACE E-FPGA BITSTREAM GENERATOR) is the tool that programs the fine-grain reconfigurable hardware platform described in the previous section. DAGGER accepts four input files: i) the

[*] This work was partially supported by the project IST-34793-AMDREL which is funded by the E.C.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1116–1118, 2004.

placement file showing the positions of the used CLBs, ii) the routing file showing the connections between CLBs, iii) the netlist file showing the clustering among CLBs and iv) the function file showing the function produced by the CLB LUTs.

To summarize:

- DAGGER is FPGA architecture and process technology independent
- It generates from scratch the configuration bitstream instead of modifying an existing one like JBits does

3.1 Theoretical Approach

The FPGA is represented with 5 different arrays (A, B, C, D, and E). Schematically, the whole layout of the FPGA is shown in Fig. 1. The specific example shows a square FPGA 4x4, with 16 (4 in each side) I/O blocks and 5 horizontal and 5 vertical channels. Arrays A (top), B (right), C (bottom) and D (left) correspond to the upper, right, lower, and left line of I/O blocks accordingly. The central blocks of the FPGA, which accomplish the logic functions, are described with array E (center).

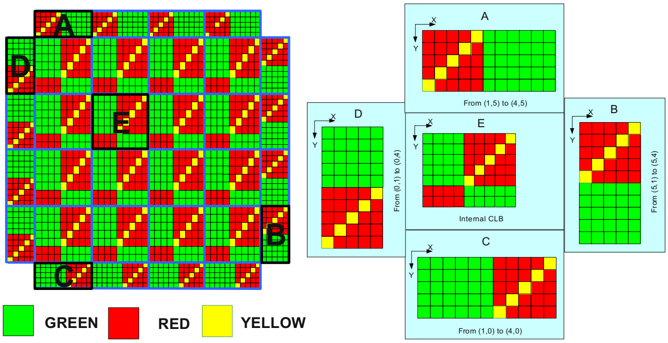


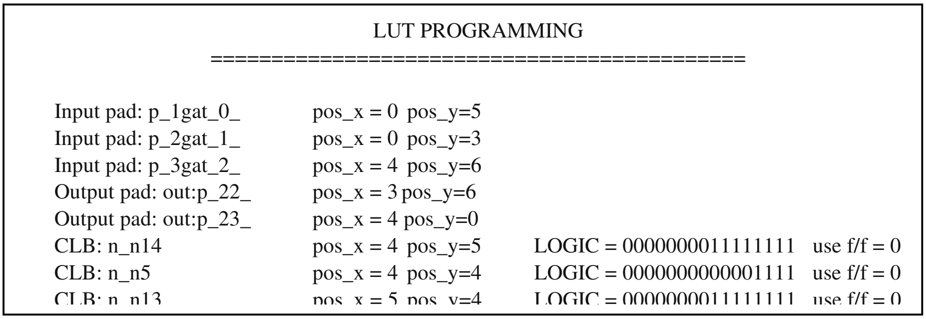
Fig. 1. FPGA encoding with arrays

In the arrays of Fig. 1, green represents the points that a connection can be made, yellow is used for the points that connect routing channels to other routing channels, and red represents the points that no connection is allowed to be made. Specifically, the green squares correspond to the connection made by the I/O pins either from the logic block, or from the I/O block to the routing channels. The yellow squares signify the possible places of connection among the routing channels at the switch box. Finally, the red squares do not represent connections and their sole reason of existence is the completion of the array.

3.2 CLB Configuration

Essentially, part of DAGGER's functionality is also the programming of the components inside the logic block. These components are the *k*-input LUT, the F/F, and the MUX which is used to multiplex the output signal (either from the LUT, either from the F/F) that comes out of the logic block.

An example of a typical output file that gives information about the logic block programming is shown in Fig. 2.



```
                              LUT PROGRAMMING
                  ==========================================

   Input pad: p_1gat_0_        pos_x = 0  pos_y=5
   Input pad: p_2gat_1_        pos_x = 0  pos_y=3
   Input pad: p_3gat_2_        pos_x = 4  pos_y=6
   Output pad: out:p_22_       pos_x = 3 pos_y=6
   Output pad: out:p_23_       pos_x = 4 pos_y=0
   CLB: n_n14                  pos_x = 4  pos_y=5     LOGIC = 0000000011111111   use f/f = 0
   CLB: n_n5                   pos_x = 4  pos_y=4     LOGIC = 0000000000001111   use f/f = 0
   CLB: n_n13                  pos_x = 5  pos_y=4     LOGIC = 0000000011111111   use f/f = 0
```

Fig. 2. Example of an output file information about the logic block programming

4 Comparisons

The configuration bitstream sizes of the AMDREL fine-grain reconfigurable hardware core and larger virtual devices are given in Table 1. The bitstream sizes are compared to the corresponding ones of Xilinx devices with similar resources (LUTs and flip-flops). It is evident that the bitstream sizes are of the similar order. Additionally, the size of the reconfiguration file from DAGGER can be further reduced using well-known data compression techniques.

Table 1. Configuration bitstream sizes

AMDREL			XILINX		
Device	**Config. bits**	**LUTS/FFs**	**Device**	**Config. bits**	**LUTS/FFs**
8×8	33,600	320/320	XCS05	53,984	200/360
			XCS10	95,008	362/616
18×18	294,880	1620/1620	XC2S50	559,200	1536/1536
27×27	1,649,120	3645/3645	XC2S150	1,040,096	3456/3456
34×34	2,331,240	5780/5780	XCV300	1,751,808	6144/6144
39×39	2,748,034	7605/7605	XCV400	2,546,048	9600/9600

References

1. http://www.vlsi.ee.duth.gr/AMDREL
2. K. Tatas et al.: "FPGA Architecture Design and Toolset for Logic Implementation", PATMOS 2003, Turin, Italy (2003) 607-616
3. V. Kalenteridis et al. "An Integrated FPGA Design Framework: Custom Designed FPGA Platform and Application Mapping Toolset Development", Accepted for publication in RAW 2004, April 26-27, 2004, Santa Fe, New Mexico, USA.

AAA and SynDEx-Ic: A Methodology and a Software Framework for the Implementation of Real-Time Applications onto Reconfigurable Circuits

Pierre Niang[1], Thierry Grandpierre[1], Mohamed Akil[1], and Yves Sorel[2]

[1] ESIEE Paris, Lab. A^2SI, Cite Descarte BP99, 93162 Noisy Le Grand Cedex France
[2] INRIA, Domaine de Voluceau BP 105, 78153 Le Chesnay Cedex France
{niangp,t.grandpierre,akilm}@esiee.fr, yves.sorel@inria.fr

Abstract. AAA is a methodology developed for the fast prototyping of real-time embedded applications and SynDEx is the software tool based on this methodology. Based on formal transformations, AAA helps the designer to implement signal and images processing algorithms onto multicomponent. This includes the support of both algorithm and architecture specifications, resources allocations and optimizations, performances prediction and multicomponents code generation. Since AAA did not initially support configurable components, this paper presents an extension of AAA/SynDEx for FPGA. This still includes the support of specification, optimization, performance prediction and automatic VHDL code generation. This paper focuses on the implementation of this work in SynDEx-Ic.

1 Introduction

Digital image-processing applications require growing computational power especially when they must be executed under real-time constraints. This power can be achieved by high performance mixed hardware architectures (called multicomponent) that are based on programmable components (RISC, CISC, DSP) and/or non programmable components (FPGA, CPLD). Furthermore, once an application algorithm and a multicomponent architecture are specified, the set of possible implementations is huge since there is still several problems to solve: HW/SW partitioning of the algorithm, distributing and scheduling of the SW part of the algorithm, adding communications, generating code for each component (processor, configurable circuit). To implement these mixed applications while meeting the timing constraints is actually hard to make, therefore there is a need for dedicated high level design methodology, to solve the specification, validation and synthesis problems. Since we want to cover mixed architectures, we have extended AAA [2] in order to support reconfigurable components such as FPGA therefore we have to address two problems. First we need to generate RTL code from the algorithm graph specification including automatic data

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1119–1123, 2004.

and control paths synthesis [1]. Secondly, since resources are limited in the configurable component (i.e. the number of CLB for FPGA), it is often better to make a factorized implementation of the specification (we will create “loop” in silicium in order to decrease the number of required resources) but since we deal with real-time application, the resulting implementation must also satisfy the given time constraint. Consequently we have developed heuristics to find a good space/time compromise. The automatic RTL code generation and the optimizing heuristics have been implemented in SynDEx and lead to a version dedicated to circuit that we named “SynDEx-Ic”. In the future, we will focus on mixed architectures. We will merge SynDEx-Ic and SynDEx. Several researches for HW/SW co-design have addressed the issue of design space exploration and performance analysis of embedded systems. Among them, we can cite the SPADE methodology which enables modeling and exploration of signal processing systems onto coarse-grain data-flow architectures and the CODEF tool (MOSARTS team) which allows the design space exploration based on a specification of partitioning/distributing/scheduling and interconnection synthesis. The next section introduces the algorithm specification of applications. The section 3 introduces briefly the architecture specification. Thus, the optimization principles and the optimization heuristic developed for SynDEx-Ic are briefly presented in section 4. Section 5 is dedicated to the automatic RTL synthesis of the applications.

2 Algorithm Specification

In the AAA methodology, an algorithm is specified as a directed acyclic graph (DAG) infinitely repeated. Each vertex is an operation. Each edge is a data-dependence between operations. A data-dependence may be a scalar or a vector of elements. Each operation may be hierarchic or atomic. It is hierarchic when its behavior is described by a sub-graph of operations, otherwise it is atomic. An operation corresponds to a sequence of instructions if it is implemented onto a programmable component (processor) or correspond to a RTL module if it is implemented onto a configurable component (FPGA).

Specification of repetitive parts: In order to specify his algorithm, the designer frequently has to describe repetitions of operation patterns (identical operations that operate on different data) defining a ”potential data parallelism”. In order to reduce the size of the specification and to highlight these repetitive parts we use a graph factorization specification. It consists in replacing a repeated pattern by only one instance of the pattern called a “repeated hierarchical operation”. It is important to notice that factorization is just a way to reduce the specification, it does not necessarily imply a sequential execution. We will see in the “optimization” section (page 1121) that the choice between a sequential implementation or a parallel implementation depends on the resources constraints and the timing constraints. Our algorithm model permits also the specification of conditioned operations allowing alternatives depending on condition.

3 Architecture Specification

Model: Here is the fundamental difference between SynDEx and SynDEx-Ic. For SynDEx, the target hardware architecture is made of programmable components, it is a multiprocessor architecture while for SynDEx-Ic, the target hardware architecture is a single configurable component (FPGA). It is modeled by the configurable logic blocks (FPGA) and their interconnections. When we will merge SynDEx and SynDEx-IC, the target architecture graph will be a mixed architecture of programmable and configurable components.

Characterization: Once the architecture is specified, it is necessary for the optimization process presented hereunder to characterize each operation of the algorithm graph. For SynDEx-Ic the characterization consists in specifying the worst execution duration of each operation as for SynDEx, but it is also necessary to specify the number of configurable units required for the implementation of each operation on the targeted FPGA.

4 Implementation and Optimization in SynDEx-Ic

Principles: From a given pair of algorithm and architecture specification, SynDEx use a distribution and scheduling heuristic in order to minimize the execution duration of the algorithm. Since SynDEx-Ic target a configurable component, it uses a heuristic that try to build an implementation that fits in the available circuit and satisfies the latency constraint. For each repeated hierarchical operation there is several possible implementations. Let's take the example of the operation "adap" given in figure 2. On a FPGA, this repeated operation may be implemented fully sequentially (figure 1-a), fully parallel (figure 1-b), or any mix of these solutions (figure 1-c). Sequential implementations require to synthesize a dedicated control path (made of multiplexer, demultiplexer, register and a control unit in figure 1-a and c). Now if we look for the performances of these three implementations, we can see that the fully sequential one (a) required the minimum space but is the slower. The fully parallel one (b) is the faster but consumes the maximum space. Finally the sequential-parallel implementation depicted in (c) is an example of compromise between space and time, it is twice faster than (a) but requires also more than twice the size. The implementation space which must be explored in order to find an "optimized" solution, is then composed of the combination of all the possible implementations of repeated operations. Consequently, for a given algorithm graph, there is a large, but finite, number of possible implementations which are more or less sequential, among which we need to select one which satisfies the real time constraint, and which uses as less as possible the hardware resources (number of logic gates for ASIC and number of Configurable Logic Blocks CLB for FPGA). This optimization problem is know to be NP-hard, and its size is usually huge for realistic applications. This is the reason why we use heuristics guided by a cost function, in order to compare the performances of different implementations of the specification.

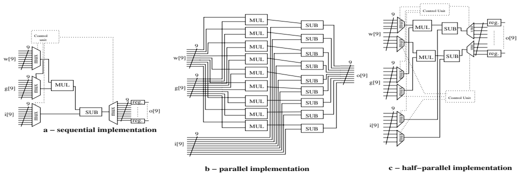

Control Unit
w[9]
g[9]
i[9]
9
MUL
SUB
reg.
o[9]
a – sequential implementation
b – parallel implementation
c – half–parallel implementation

Fig. 1. Three examples of implementation of the repeated operation "adap" specified in figure 2

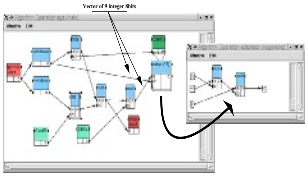


Fig. 2. SynDEx/SynDEx-Ic snapshot of the Equalizer application

These heuristics allow us to explore only a small but most interesting set of all the possible implementations into the implementation space. Since we aim at rapid prototyping, our heuristic is based on a fast but efficient greedy algorithm, with a cost function f based on the critical path length metric of the implementation graph: it takes into account both the latency T and the area A of the implementation which are obtained by the preliminary step of characterization presented in section 3. This heuristic is based on estimator algorithms in order to find the surface and the latency of each implementation.

5 Automatic RTL Code Generation in SynDEx-Ic

Once SynDEx-Ic has built an optimized implementation, it is able to automatically generate the corresponding RTL code including the specified data path and the synthesized control path. The RTL code generation performed by SynDEx-Ic entails two steps. In the first step, SynDEx-Ic generates two intermediate files based on a macro code, one corresponding to the design package of the application and the other to the application design. For each vertex of the graph, we produce macro functions symbolizing the definition of each component and each connecting signal and the mapping of components. In the second step, these intermediate files are convert into VHDL code with the help of given VHDL libraries and a macro-processor called GNU-m4. So, using intermediate files instead of generating directly the VHDL code permits to have a RTL code generator independent of a target language which can be Verilog language for example or SystemC.

6 Conclusion and Future Work

We have presented the different steps to generate a complete RTL design corresponding to the optimized implementation of an application specified with SynDEx-Ic. Given a algorithm graph specification it is then possible to generate a multiprocessor optimized implementation using SynDEx, or a FPGA optimized implementation using SynDEx-Ic. Moreover, since SynDEx-Ic is based on SynDEx the development flow is unified from the application designer point of view. The next step is to merge SynDEx and SynDEx-Ic in order to support **mixed** parallel architectures: architectures with programmable components **and** reconfigurable components (FPGA). To support such architectures, we first have to solve the partitioning problem between programmable and configurable components. For that purpose we plan to link the two kinds of optimization heuristics of AAA (i.e SynDEx heuristic for programmable components and SynDEx-Ic heuristics for configurable components). We must also support the automatic communication synthesis between the different components (programmable and reconfigurable). So, this methodology will be used for optimized hardware/software co-design, leading to the generation of either executives for the programmable parts of the architecture **and** RTL for the non-programmable part.

References

1. L.Kaouane and M. Akil and T. Grandpierre and Y. Sorel. A methodology to implement real-time applications on reconfigurable circuits. In *Special issue on Engineering of Configurable Systems of the Journal of Supercomputing.* Kluwer Academic, 2004 (to appear).
2. T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *First ACM & IEEE Intl. Conference on formal methods and models for codesign.* MEMOCODE'03, Mont Saint-Michel, France, june, 2003

A Self-Reconfiguration Framework for Multiprocessor CSoPCs

Armando Astarloa, Jesús Lázaro, Unai Bidarte, Jose Luis Martín, and Aitzol Zuloaga

Department of Electronics and Telecommunications, Faculty of Engineering,
University of the Basque Country
Urquijo s/n, 48013 Bilbao - SPAIN
{jtpascua,jtplaarj,jtpbipeu,jtpmagoj,jtpzuiza}@bi.ehu.es

Abstract. In this paper present a partial run-time reconfiguration framework focused on multiprocessor core-based systems implemented on FPGA technology. It is called *Tornado*, and it is composed of an infrastructure of signals, protocols, interfaces and controller to perform safe hardware/software reconfigurations.

1 Introduction

G. Martin in the chapter “The History of the SoC Revolution” in [1] emphasizes how the IP core-based design with commercial reconfigurable FPGA platforms is a strong reality in SoC today and it will continued in the future. Focused on applying self-configuration to these design types it must be highlighted the work of Blodget et al. [2]. They present a Self-Reconfiguring Platform (SRP) for Xilinx Virtex-II and Vitex II Pro. Danne et al. [3] present a technique to implement multi-controller systems using partial reconfigurable FPGAs. They use an external configuration manager which receives reconfiguration request from an internal supervisor.

In this paper, we present a self-reconfiguration framework that we call *Tornado*, which applies partial Run-Time Reconfiguration (RTR) to *SoPC* core based designs that include multiple cores with tiny microprocessors embedded, converting them in a *Configurable-System-on-a-Programmable-Chip* (CSoPC) designs. We pretend to go forward into the application of the partial run-time hardware and software reconfiguration among multiple processors cores in IP-Core FPGA designs.

2 Tornado Self-Reconfiguration Framework

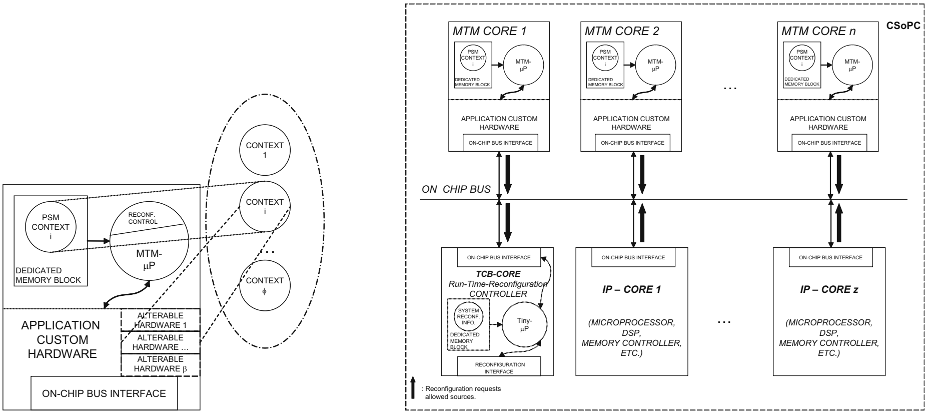
2.1 Targeted Architectures

The integration of tiny microprocessors into cores offers a very flexible tool to the designer. Complex state machines or control loops can be efficiently implemented using little embedded RISC microprocessors [4].The dominant implementation

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1124–1126, 2004.

factor of these small processors is size. We will call "Mixed Cores" (MC) to the cores built with an embedded small CPU and additional hardware.

Figure 1(a) shows an abstraction of a MC that includes a small microprocessor with its software embedded into the FPGA dedicated RAM, a custom hardware for the specific application and an interface to link the core to the *on-chip* Bus. An standard specification for IP-Cores interconnection is used for the interface. We propose to this widely employed architecture the addition of RTR benefits. In order to convert a SoPC with mixed cores in a CSoPC we consider two major issues:



(a) (b)

Fig. 1. MTM-Core with PSM multi-context and partially modificable and the generalized *Tornado* architecture.

- *Software context:* The size of the program memory for the tiny CPU embedded into the core is strongly constrained by the size of the dedicated RAMs of the FPGA. The application of the partial RTR to MC allows the addition of multiple software contexts that enhance the FPGA use in applications where the MC architecture fits (control, protocol processing, half-duplex transceivers, etc.). Figure 1(a) represents the proposed enhancement. Note that the embedded *tiny* processor has been provided with an interface called *reconf control*. Basically we propose the control of the Program and Stack Counters plus the possibility to disable the reconfiguration temporally. We have called the microprocessor capable of managing multi-context small software units *Multicontext Tiny Microprocessor (MTM).*
- *Hardware context:* To apply small modifications into the custom HDL described hardware. This feature is being integrated successfully into industrial-quality cores [3,5,6]. Nevertheless, this hardware reconfiguration is restricted to changing the contents of an area-located elements of the FPGA obtaining a slightly different core instead a new one.

Figure 1(b) represents a simplified architecture of a SoPC that includes *n* *MTM-Cores* and *z* *IP Cores* (without *Tornado* reconfiguration capabilities). All of them use an standard interface to be linked with the *on-chip* bus. The bus topology is only constrained by the bus specification used, having selected for the representation a *Shared Bus Topology.*

To manage these different partial reconfigurations a *Reconfiguration Controller Core* called *Tornado Controller Basic (TCB)* is included into the system. As it is represented in figure 1(b), in our approach the sources of the reconfiguration requests could be very diverse. The reconfiguration request to the *TCB* can come from any core, either *MTM-Cores* or general *IP-Cores* including Hard or Soft powerful microprocessors embedded into the platform. It also allows a *MTM-Core* request to the *TCB* for a self-reconfiguration through its *on-chip* bus interface.

3 Conclusions

Acknowledgment In this paper we have presented a run-time partial reconfiguration framework focused on multiprocessor core-based systems implemented on FPGA technology. Future works will be focused on the application of the *Tornado* to more complex systems and the design of more flexible reconfiguration controllers that support dynamic partial bitstream generation.

Acknowledgment. The work described in this paper has been supported by the Ministerio de Ciencia y Tecnologi´a of Spain and FEDER in the framework of the TIC2001-0062 research project.

References

1. G. Martin and H. Chang (Eds.). Winning the SoC Revolution: *Experiences in Real Design.* Kluwer Academic Publishers, Massachusetts, USA, 2003.
2. B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan. A Self-reconfiguring Platform. *Field-Programmable Logic and Applications: FPL 2003*, Lecture Notes in Computer Science, Springer, 2778:565–574, September 2003.
3. K. Danne, C. Bobda, and H. Kalte. Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration. *Field-Programmable Logic and Applications: FPL 2003*, Lecture Notes in Computer Science, Springer, 2778:272–281, September 2003.
4. K. Chapman. PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices. Xilinx Application Notes, February 2003.
5. V. Eck, P. Kalra, R. LeBlanc, and J. McManus. In-Circuit Partial Reconfiguration of RocketIO Attributes. Xilinx Application Notes, January 2003.
6. M. Dyer, C. Plessl, and M. Platzner. Partially Reconfigurable Cores for Xilinx Virtex. *Field-Programmable Logic and Applications: FPL 2002*, Lecture Notes in Computer Science, Springer, 2438:292–301, September 2002.

A Virtual File System for Dynamically Reconfigurable FPGAs

Adam Donlin[1], Patrick Lysaght[1], Brandon Blodget[1], and Gerd Troeger[2]

[1] Xilinx Research Labs, 2100 Logic Drive, San Jose, California, USA.
[2] Technische Informatik, Universitaet Leipzig, Leipzig, Germany

1 Introduction

Platform FPGAs have dramatically changed the role of FPGAs in embedded systems. With increased density and immersed complex IPs, FPGAs no longer simply play 'a role' in embedded systems – FPGAs *are* embedded systems. To accommodate the increased system capability of Platform FPGAs, they also host a rich embedded software environment. Embedded Linux has emerged as a common software infrastructure for embedded systems in general and is also being employed in FPGA-based embedded systems.

The computational power of platform FPGAs make them a compelling implementation technology but they also offer a unique degree of reprogrammability. Manipulating the configuration memory of the FPGA allows us to influence both the behavior and structure of the digital logic implemented on the device. Some important embedded systems functionality is enabled by this: in-field upgrade; management of both the hardware and software; system introspection and in-field debug are each enabled by observing the configuration of the FPGA through its programming interface. Our challenge, however, is making the programmability of the FPGA easily accessible to the embedded system designer and user. This paper outlines a highly innovative mechanism and interface to the FPGA's programmability. Our approach is intuitive to the user and its implementation has a natural fit within the Platform FPGA's embedded system infrastructure.

2 Harnessing FPGA Programmability

Most reconfigurable systems use software APIs to control and manipulate the FPGAs in the system. A user of the API writes a program to invoke the methods of the API and hence manipulate the bitstream data held in FPGA's configuration memory. Early FPGAs, because of their limited (serial) access mechanism to the configuration memory, had correspondingly simple APIs that allowed the entire memory to be read or written. Later APIs took advantage of greater flexibility in configuration memory interfaces of modern FPGAs. The Xilinx JBits API[2], for example, allowed the data in the memory to modified in finer granularity. More importantly, JBits went beyond simply transporting fragments of data between the control program and the target FPGA's configuration memory: it also provided the means of *interpreting* the semantic content of the data.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1127–1129, 2004.

Recently, the self-reconfiguring platform (SRP)[1] was introduced. It provides a subset of JBits' capabilities in the C-language idiom and is optimized for use in embedded systems. Furthermore, logic configured onto the FPGA may access and manipulate the configuration data of their host FPGA.

Whilst functionally correct, these APIs do have certain limitations. Firstly, they are language dependent: using the API requires the system designer adopt the API vendor's chosen language. Additionally, they are much less likely to be standardized: the methods and data structures that comprise the API are likely to vary between vendors. Combined, these limitations raise the barrier of entry to implementing reconfigurable systems. The API's language choice must be justified and the designer must 'acclimatize' to the complex API. Our contribution in this paper is to present a novel means of manipulating the content of FPGA configuration memory. It is based on the concept of a "virtual file system" and is language-neutral, flexible, and highly intuitive.

3 The FPGA Virtual File System

A standard file system presents the data encoded on the long-term storage media of a computer system as a hierarchy of directories and data files. The file system is, essentially, a convenient "user-interface" to the stored data. Virtual file systems (VFS) extend coverage of the file and directory metaphor to other devices and resources in a system. The VFS prescribes a mapping between the abstract notions of "file" or "directory" and target dataset. It also determines the behavior of the basic file and directory operations (open, read, etc.), converting them to a series of native transformations on the target data set. Linux's `/proc` is a familiar example of a VFS. Its files and directories actually represent objects and live data structures within the local Linux kernel. Through `/proc`, users of the system may inspect and manipulate live kernel data structures just by performing standard file system operations on the `/proc` files and directories.

We have created a prototype of the VFS concept applied to FPGA configuration memory. Through our FPGA-VFS we have mapped the file and directory metaphor to allow users to interact with the resources of their system's FPGAs in new and convenient ways. To that end, the Xilinx FPGA-VFS allows a user to traverse a file system view where files and directories actually correspond to regions of configuration data from the FPGAs within their system. Related resources are grouped within the FPGA-VFS's directories and its files are containers holding the configuration data for a specific instance of a resource. We may navigate the FPGA-VFS directory hierarchy with standard navigation commands like `ls` and `cd`. Upon locating the file representing the configuration data of the given resource, we may use standard applications (`echo`, `cat`, `touch`, `rm`, etc.) to open, read, write, close or delete, etc., the file. In designing the VFS, we select the particular configuration data to be represented by each of the files in the VFS. We also decide how to group them into directories. In our initial prototype, each instance of a physical resource in the FPGA is presented as a distinct file. We group the files for the same type of resource into VFS directories. Sub-directories within the resource-type directories organize the specific instance files according to the geometric position of the resource in the FPGA floorplan, or an ascending numeric sequence.

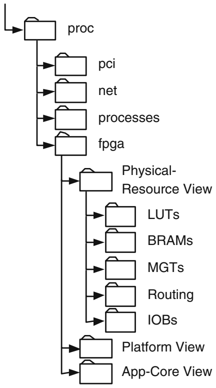


Fig. 1. Filesystem view of the FPGA's Physical Resources

A "physical resource" VFS is only one potential file system view of the FPGA configuration data. We may create other VFS views that interpret and present *the same* configuration data in different ways. Different views may co-exist as different top-level directories in the FPGA-VFS file space. In Figure 1 we show the FPGA-VFS directory structure and the main subdirectories of the physical resources view. If, for example, we wish to modify the configuration data of a given BRAM in our host FPGA, we need only `cd` to the appropriate BRAM directory and manipulate the data files we find there.

4 Conclusions and Future Directions

FPGAs have a central role in future embedded systems. In this paper we have introduced the FPGA-VFS as a powerful mechanism that an embedded system designer may use to control the FPGAs within their system. Our approach overcomes some important problems in traditional APIs for manipulating FPGA configurations: it is flexible, language neutral and highly intuitive. We are actively exploring and prototyping alternative views and expanding the nature of an FPGA-VFS to systems with ubiquitous internet access.

References

1. Blodget B., et al., "A Self-reconfiguring Platform", in FPL 2003. pp 565–574.
2. Guccione S., et al., "JBits: A Java-based interface for Reconfigurable Computing", in MAPLD 1999.

Virtualizing the Dimensions of a Coarse-Grained Reconfigurable Array

T. Ristimäki and J. Nurmi

Institute of Digital and Computer Systems, Tampere University of Technology, Finland
{tapio.ristimaki, jari.nurmi}@tut.fi

Abstract. In this paper a context switching mechanism is implemented into an existing MIMD based coarse grain reconfigurable IP block. Context switch is not used only to hide reconfiguration latency, but the emphasis is on virtualizing the dimensions of an array of processors by folding the array to multiple configurations and by using node based internal schedulers.

1 Introduction

In the coarse grain reconfigurable systems the configware is very hardware specific, and thus the same configuration bit stream cannot usually be used in differentially configured instances of the same architecture. However it is obvious that if the implementation details of the hardware could be hidden, the configware development would be a far more profitable. The size of an array is one of the most varying parameter in coarse grain devices, and thus one of the most interesting ones to be virtualized

The benefits of the context switch as a method for hiding the configuration delay in reconfigurable devices is noticed and scientific work is done concerning among others coarse grain reconfigurable systems [1,2,3,4]. However, the use of context switch in coarse grain reconfigurable structures to virtualize the dimensions of an array is rarely studied. In [5] the model of extensions needed to hide the size of an array is given. However the [5] sidesteps the actual implementation and leaves all needed scheduling to the controlling processor.

2 Context Switch Hardware Extensions in RAA

RAA [6] is a coarse grain reconfigurable algorithm accelerator IP block based on MIMD topology. It consists of tiny 16-bit DSP processors each coupled with its own local data and program memories. RAA uses two communication models. (1) Processors communicate via FIFOs such that each processor can read FIFOs of its four neighbors and (2) the outside controller sees all local memories via an internal global bus attached to the all memories via *nodes*' bus interfaces. Each node has row and

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1130–1132, 2004.

column addresses according its place and the node memories are accessed via those. The internal structure of the node is illustrated in figure 1a.

In the architecture level the context switch was added to the RAA by separating the CPU core and FIFOs from *memory part* and by duplicating the separated memory part such that each context has its own as shown in figure 1b. The interface block of the first memory part is similar to the original one, but the others recognize virtual addresses such that the address of the first added context of node **x(row,column)** is **x(row+***max***(row),column+***max***(column))** and address of the second context is **x(row+**2×*max***(row),column+**2×*max***(column))** etc.

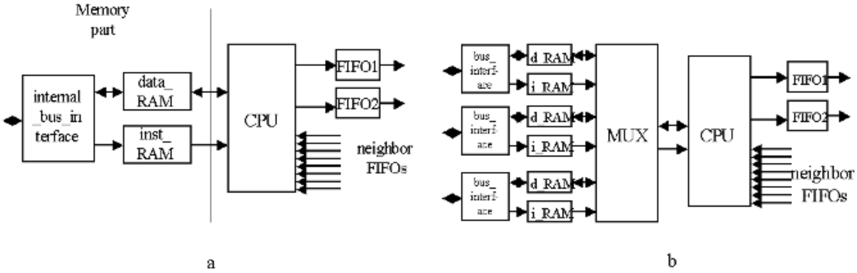


Fig. 1. (a) Structure of the basic node. (b) Structure of the node with three contexts.

Duplicated memory parts are seen differently from outside controller and CPU core points of view. Because the whole memory part, including the interface, is copied the outside controller does not see if the configuration memory has its own CPU core or is it only one context among others in a single node. Thus presented addressing mechanism not only implement accessing method for context memories but also automatically hides the number of CPU cores in RAA from outside controller point of view. The hash function used to address the duplicated memory parts automatically places configurations such that the neighbor configurations in the bigger array are also located adjacent nodes in the context switch extended smaller array. Because in the RAA the nodes see only neighbors the previous condition keeps communication solid if original to the bigger array synthesized configware bit stream is uploaded to the smaller context switch extended array.

From CPU core point of view there are really many memory parts and thus selection between those has to be done with multiplexer. In addition the CPU core three instruction cycle level registers i.e. accumulator 1, accumulator 2 and program counter are duplicated such that each context has its own one and a selection between those has to be done as well. Because the size of the array is wanted to be virtualized in hardware level, the internal context scheduling was implemented. The scheduler is added into the every CPU core as a hardware extension. The scheduler algorithm uses non-pre-emptive tactic i.e. if the context in execution is not capable to continue, the context switch to the next one is made implicitly. The possible context swiching events are (1) end of program, (2) the outside operand in FIFO or in memory is not ready and (3) the memory is locked by outside controller's memory access.

Because each node does scheduling itself, the different processors can execute different configurations at a certain moment and thus the sender of data in FIFO can belong to a different configuration than the receiver in certain time instant. Thus the configuration identification number stamp is added to the every element of every FIFO. If first data element in FIFO belongs to the context not in the execution the context switch is done.

3 Results and Conclusion

In hardware side the extensions described in section 2 were implemented to the existing RAA architecture with VHDL and the developed new design was synthesized to 0.18μm technology with Synopsys tools. The area of non-context switch extended, two contexts, four contexts and eight contexts node was 0.17, 0.23, 0.32 and 0.51 square millimeters, respectively. Thus the size of an array increases linearly and virtually four times bigger array takes ~1.7 times more area than the original one. On the other hand it was noticed that it was not the scheduler located inside the core, but the memories, which causes the extra area in practice.

In this paper it is presented how a fast context switch can be implemented to the MIMD based coarse grain reconfigurable algorithm accelerator IP block and the actual implementation is done in RAA framework. It is shown how the context switch can be used to virtualize the dimensions of the array by using node based hardware scheduler. The synthesis results show that an area penalty of the extra controlling structures needed is insignificant but the increasing area comes from the additional context memories i.e. the feature of abstracting the size of the array is got in practice free on top of the context switch.

References

1. Kitaoka, T., Amano, H., Anjo, K.: Reducing the Configuration Loading Time of a Coarse Grain Multicontext Reconfigurable Device. Proc. of FPL 2003. pp. 171-180.
2. Fujii, T, Furuta, K., Motomura, M., Nomura, M., Mizuno, M., Anjo, K., Wakabayashi, K., Hirota, Y., Nakazawa, Y., Ito, H., Yamashina, M.: A Dynamically Reconfigurable Logic Engine with a Multi-Context/multi-mode Unified-Cell Architecture. Proc of Solid-State Circuits Conference. pp. 364-365. 1999.
3. Resano, J., Mozos, D., Verkest, D., Vernalde, S., Catthoor, F.: Run-Time Minimization of Reconfiguration Overhead in Dynamically Reconfigurable Systems. Proc. of FPL 2003. pp. 585-594.
4. Maestre, R., Kurdahi, F.J., Fernandez, M., Hermida, R., Bagherzadeh, N., Singh, H. Very.: A Framework for Reconfigurable Computing: Task Scheduling and Context Management. Transactions on Very Large Scale Integration (VLSI) Systems. Vol. 9, Issue: 6, pp. 858-873. 2001.
5. Plessl, C., Platzner1, M.: Virtualizing Hardware with Multi-context Reconfigurable Arrays. Proc. of FPL2003. pp. 151-160.
6. Ristimäki, T., Nurmi, J.: Reprogrammable Algorithm Accelerator IP Block. Proc. of VLSI-SOC, pp. 228-232. .2003.

Design and Implementation of the Memory Scheduler for the PC-Based Router

Tomáš Marek[1], Martin Novotný[1], and Luděk Crha[2]

[1] Faculty of Electrical Engineering, Czech Technical University Prague,
Technická 2, Praha 6 160 00, Czech Republic,
{marekt|novotnym}@liberouter.org

[2] Faculty of Information Technology, Brno University of Technology,
Božetěchova 2, Brno 612 66, Czech Republic,
crha@liberouter.org

Abstract. This paper deals with a design of a memory scheduler as a part of the Liberouter project.

Nowadays, the majority of the designs of memory schedulers is aimed at providing a high throughput while using a high-capacity DDR SDRAM memory. The memory scheduler is FPGA-based. This allows us to test many versions of the design with real network traffic and to set optimal parameters for the memory scheduler units. For reasons of capacity and throughput we use DDR SDRAM memory. The effective DRAM access time is reduced by overlapping multiple accesses to different banks in a special queue composed of the FPGA embedded Block SelectRAM$^{\text{TM}}$s.

Keywords. Router, Virtex II, FPGA, DDR SDRAM, memory scheduler.

1 Introduction

Packets it the Liberouter router are processed by several blocks: Header Field Extractor (HFE), Lookup Processor (LUP), Packet Replicator and the Priority Queues Block, Edit Engine (EE), Memory Scheduler (MSH).

The HFE pushes the body of the packet into the dynamic memory. Meanwhile, it parses the packet's headers and creates a structure called Unified-header. The Unified-header is a fixed structure containing relevant information from packet headers. The LUP processes the Unified-headers by performing the lookup program and sends the result to the Edit Engine. The Replicator block replicates the packet identification (dynamic memory allocation block number) as well as the pointers to the editing programs into the dedicated queues. The EE block modifies the headers of the packets.

The Memory Scheduler has a number of tasks: it must store packets and the Unified-header from the HFE units in the dynamic memory, keep reference counts for each packet (at its address in the memory), increment or decrement reference counts upon a Replicator or an EE demand, and read a packet from the memory upon an EE demand. Another major MSH function is to guarantee equal access of the units to the shared memory and shared address resources.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1133–1135, 2004.

2 Analysis

The M/M/1 Kendall mass service system model will be used for the analytic system model. In the M/M/1 mass service system model with limited queue length for full queue probability stands:

$$p_n = \frac{1-\rho}{1-\rho^{n+1}} \cdot \rho^n,\ \rho = \frac{\lambda}{\mu}$$

where n is queue length and ρ is traffic intensity, λ is the average number of incoming requests per time unit and μ is the average number of served requests per time unit. Input processes can be merged since merging the Poisson processes is a Poisson process with λ equal to the sum of subprocesses parameters.

The memory scheduler is compared for three different parameters λ, the first two λ parameters were obtained from the real CESNET2 network [4], the third parameter is calculated for the 20% ballast of a 1 Gbit network. Parameter μ is the throughput of the respective solution. Parameter Σn is the sum of all buffers length in the request count units. The parameters are depicted in the table below.

Table 1. ρ for different λ and the result

λ_{IN} [Mbit/s]	λ_{OUT} [Mbit/s]	ρ_{DRAM} [–]
200	200	0.47
75	129	0.24
23	39	0.08

The throughput of the memory scheduler is given by the Low Level Scheduler throughput and the length of the internal burst described in chapter 3. In the worst case scenario, one DDR SDRAM memory read/write operation takes 8 clock cycles [5]. For 64 bits DDR SDRAM memory with 133MHz clock frequency, the throughput is $\rho = (133 \cdot 10^6/8) \cdot 64 \cdot 4 = 4256\ MBit/s$ Full queue probability for different queue length (Σn) is depicted in Figure 1.

The memory scheduler cannot transfer a burst of the data immediately, thus each connected unit has its own FIFO memory. For capacity reasons the Block SelectRAM$^{\mathrm{TM}}$ seems to be optimal. In Virtex II [2], the maximum bus width is 32b. All the input interfaces include input queues and the data are transfered from these interfaces to the DDR SDRAM queue in a 32b wide burst.

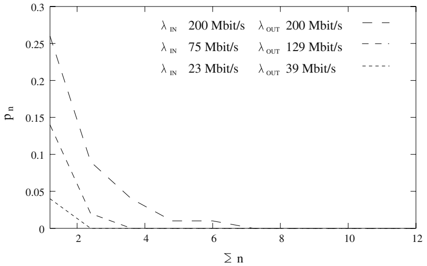


Fig. 1. Full queue probability

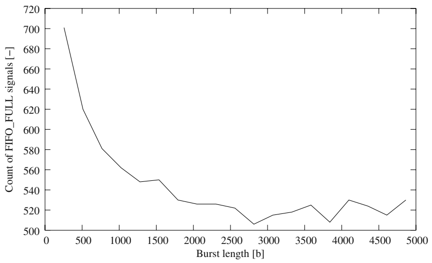


Fig. 2. Internal burst length

3 Realization

The MSH block structure is divided into three parts: Low level communication with the DDR SDRAM (Low Level Scheduler), Data streams management (Data Management) and Addresses and address references management (Address Management). The throughput of the memory scheduler also depends on the internal burst length. Excessively long internal bursts can cause a starvation of the output units and overloading of the input interfaces queues. Short bursts waste the transfer capacity. Measurements were taken to derive the optimal length of internal bursts. As an optimization parameter, the average number of clock cycles with the input interface queue full during the full load of all interfaces was taken. The measurement was performed in five 30 us intervals. The results of the measurement are depicted in Figure 2.

To define the optimal length of the internal bursts, the average length of packets in local area networks must be considered. This length is about 600B (4800b), but packets about 64B (512b) long are also important for the network traffic [4][6]. We arrived at a value of 1024b as a trade off between these demands.

4 Conclusions

In this paper we have presented the design of a memory scheduler as a part of the Liberouter project [1]. We discussed the problem of data transfers from/to input/output interfaces to/from the DDR SDRAM while attempting to find an optimal length of internal bursts. We also discussed the length of the input queues on the basis of a mass service system theory and the support of collective communication network operations using the references counters. Currently, the memory scheduler is implemented and tested on the COMBO6 board.

References

1. Liberouter: Liberouter Project WWW Page. http://www.liberouter.org, 2004
2. Xilinx: DS031 Virtex-II 1.5V Field-Programable-Gate-Arrays. http://direct.xilinx.com/bvdocs/publications/ds031-v2.pdf,2002
3. Novotný, J., Fučík, O., and Antoš, D.: Project of IPv6 Router with FPGA Hardware Accelerator. Field-Programmable Logic and Applications - FPL 2003, Springer, Berlin 2004, pp.964-967
4. CESNET: CESNET2 Load Map. http://www.cesnet.cz/provoz/zatizeni/ten155_mapa/mapa.last_month.20030700.html, 2003
5. Micron Technology, Inc.: DOUBLE DATA RATE (DDR) SDRAM.
6. Whetten, B., Steinberg, S., and Ferrari, D.: The Packet Starvation Effect in CSMA/CD LANs and a Solution. http://www.ethermanage.com/ethernet/pdf/FDDQ_LCN_1.pdf, 2002

Analog Signal Processing Reconfiguration for Systems-on-Chip Using a Fixed Analog Cell Approach

Eric E. Fabris, Luigi Carro, and Sergio Bampi

Universidade Federal do Rio Grande do Sul, Informatics Institute,
90035-190 Porto Alegre, Brazil
{fabris, carro, bampi}@inf.ufrgs.br
http://www.inf.ufrgs.br/~gme

Abstract. This work proposes the combination of a set of fixed identical analog cells and a programmable digital array to create an infrastructure for reconfigurable analog signal processing. The *fixed analog cell* (FAC) implements the frequency translation (mixing) concept of input signal, followed by its conversion to the Σ domain. This approach makes possible the use of a constant analog block, and also a uniform treatment of input signals from DC to high frequencies with a significant advantage: no analog part tuning, sizing, or redesign required. The proposed topology allows easy integration with widely used fine grain FPGAs or general purpose SOC, and it is ideal to develop highly programmable mixed signal devices, since the tradeoffs among resolution, linearity, bandwidth and area can be developed at the user design time, and not at the FPGA design time like in other mixed-signal programmable arrays.

1 Introduction

There are a significant number of mixed signal SOC applications that deal with heterogeneous signals (low and/or high frequency) and signal processing functions (linear and non-linear). Therefore, the incorporation of some degree of analog programmability is imperative in current digital reconfigurable devices or general purpose SOCs. This work discusses an analog interface architecture targeted to FPGA platforms and mixed signal SOCs. This interface allows the implementation of complex linear and non-linear circuits, permitting high performance analog prototyping and reuse, creating a viable path to fast mixed signal prototyping.

2 Interface Architecture

The basic structure of the *fixed analog cell* (FAC) is composed by an input mixer and a continuous time (CT) N-th order band-pass Σ modulator. The signals for controlling the mixer and the feedback DAC are generated by the digital reconfigurable block [1]. As this cell is fixed at structural and functional level, it can have an optimal

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1136–1138, 2004.

design to provide the appropriate bandwidth, signal-to-noise-ratio-dynamic-range (SNRDR) and frequency coverage for the target application set [2, 3]. Power and area budget can be used as constrains for the tailoring process at design time. Moreover, the base-band signal processing mechanism allows us to process signals from DC up to HF in the same infrastructure, which would not be possible with any of the known analog or mixed-signal programmable topologies. The channel processing flow proposed herein requires a demodulator after the input bit-stream data acquisition to bring the input signal to its base-band that can be implemented in the configurable digital block, easily. The steps that follow are associated with the application of the desired signal processing function over the input signal also using the digital infrastructure. The analog output signal is generated by a one bit DAC and a reconstruction filter.

Figure 1 shows the architecture of this general interface using a set of identical FACs. Each FAC has its own digital block to realize the acquisition of the generated digital signal and its conversion to signal base-band for processing. It is important to remark that all signal treatment is intended to be processed at signal base-band.

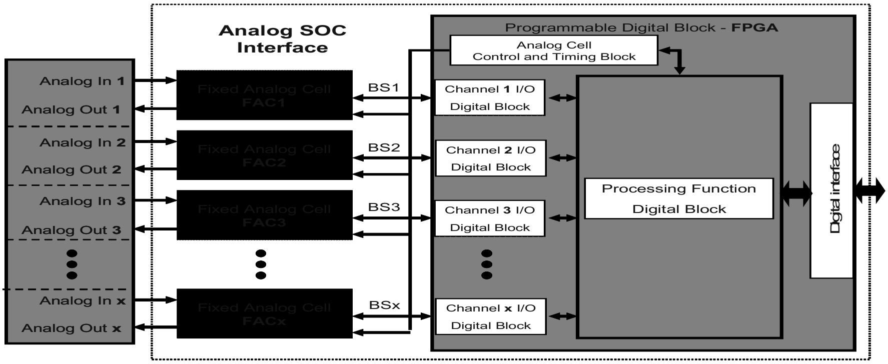


Fig. 1. Analog SOC interface architecture.

3 Application Mapping and Results

A FAC discrete prototype was built to provide support for the proposed methodology and, also, to demonstrate the application mapping potentiality of this interface. This prototype is comprised by the FAC and the digital infrastructure (a FPGA board with an ACEX1K EP1K100QC208 device). The FAC was designed using a second order (lowest order or lower achievable SNRDR and bandwidth) continuous-time band-pass Σ modulator and a passive CMOS mixer. Although discrete, the overall principle is certainly valid for any VLSI implementation, with the ensuing benefits.

The interface was configured as a multi-band analog-to-digital-converter employing an optimized *sinc2* decimator filter [2]. Table 1 summarizes the ADC testing results. These results show the frequencies coverage and the balance between SNR, area and power (logic elements – LE). With a higher order analog resonator, this trade-off could be made even more aggressive [2].

Table 1. The ADC test summary employing a variable size *sinc2* decimator filter.

	SNR [dB] / ENOB		
Input F_{norm}	**64 – 4BW – 268 LE**	**128 – 2BW – 278 LE**	**256 – BW – 289 LE**
0.1	42 / 6.5	52 / 7.5	55 / 8
10	39 / 6	48 / 7	50 / 7.5
75	40 / 6	47 / 7	51 / 7.5

It was also implemented an analog adder and an analog multiplier following the topologies proposed in [4]. The **n** channel analog adder is a simple **n** input multiplexer and a module **n** counter operating **n** times the bit stream sampling frequency. The two channel analog multiplier was synthesized resulting in a total 269 LE cost.

4 Conclusion

The results herein presented showed the potential of the proposed architecture to provide flexible analog programmability and fast prototyping to SOC designers. The proposed topology allows easy integration with the widely used fine grain digital FPGAs or general purpose SOCs, and it is ideal to develop highly programmable mixed signal devices, since the tradeoff among resolution, linearity, bandwidth and area can be developed at the user design time, and not at the FPGA design time like in other mixed-signal arrays. This way, the system designer can experiment different analog and mixed signal functions with digital FPGA programming, until reaching an optimum solution

Instead of changing the circuit topology, one shifts the signal band. This has many useful implications, since the redesign or migration of the proposed configurable cell to other technologies is greatly simplified. Only the analog part of the modulator must be redesigned or targeted at the physical level to a new technology. This certainly simplifies the design process. The remaining analog processing circuits can be easily ported to a new technology, since they are simpler large signal modules, and hence consolidated digital tools are available for this.

The paper has also shown that at the application level many blocks were used to build different target applications. Most important, we clearly demonstrated the reuse of the previously designed blocks.

References

1. Fabris, E.E., Carro, L., Bampi, S.: An Analog Signal Inerface With Constant Performance for SOCs: Proceedings of ISCAS 2003, Vol. 1, 773-776
2. Rodríguez-Vázquez, A. et al. (ed): CMOS Telecom Data Converter. Kluwer Academic Publishers, Boston, (2003)
3. Behzad Razavi.: RF Microeletronics. 1st edn, Prentice Hall, Upper Saddle River (1998)
4. V.F. Dias, “Signal Processing in the Sigma-Delta domain”, Microelectronics Journal, vol.26, (1995) 543-562

Intellectual Property Protection for RNS Circuits on FPGAs

Luis Parrilla, Encarnación Castillo, Antonio García, and Antonio Lloris

Department of Electronics and Computer Technology, University of Granada
18071 GRANADA (Spain)
{lparrilla,ecastillo,agarcia,lloris}@ditec.ugr.es

Abstract. A new procedure for Intellectual Property Protection (IPP) of circuits based on the residue number system (RNS) and implemented over FPL devices is presented. The aim is to protect the author rights in the development and distribution of reusable modules (IP cores) by means of an electronic signature embedded within the design. The presented protection scheme is oriented to circuits based on the RNS but can be easily extended to systems implemented on programmable devices. As an example, a 128-bit signature is introduced into a CIC filter without affecting performance and negligible area increase.

1 Introduction

The increasing complexity in digital IC designs, combined with the hard competition in the electronics market, is leading to substantial changes in design strategies, directed to minimize the development time and costs. These strategies [1] are based on the use of reusable modules (IP cores) and provide precious competitive advantages. This makes new challenges not yet considered to arise, one of the main being the intellectual property protection of those shared modules, being necessary to provide mechanisms to the author for claiming intellectual property rights. The usual procedures for IPP in media and hardware support [2] consist of hiding a signature (watermark) that is difficult, if not impossible, to change or remove.

In our approach, an MD5 [3] digital signature is introduced in the design, with appropriate techniques for embedding and extracting this signature in RNS-based systems. This is possible due to the RNS particularities [4], that have traditionally been exploited for performance enhancement and have made RNS particularly well-suited for FPL implementation [5-6] with an extensive use of the look-up tables available in programmable technologies.

2 Protection by Used Table Spreading on FPL

The key idea lies in "spreading" all the possible bits of the digital signature through non used cells of look-up tables included in the RNS-based design. The fact that these

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1139–1141, 2004.

tables are part of the design makes extremely difficult the possibility of an attacker finding these signature bits. Thus, problems related with previous IPP methods based on the use of tables or logic elements not in use are solved [2,7] and, as the signature embedding is performed in the high-level design description stage, this method is more secure than other IP protection techniques that rely on Place&Route modifications. The steps in the signature embedding process are:

1. The selected signature is stored in a public domain document.
2. MD5 is used to convert the signature in a bit stream or digital signature.
3. This bit stream is partitioned in blocks.
4. These blocks are embedded into empty positions of look-up tables, or signature memory positions (SMP).
5. The signature extraction stream (SES) detection hardware is included in the design.

There is not a fixed algorithm for spreading the bits, this will depend on each particular design, but this extra effort makes harder to change or remove the embedded signature. The signature extraction process consists of applying the SES to the circuit, which will make each one of the SMPs to be addressed, instead of the memory position that the normal RNS hardware would point to. During a few clock cycles, the output system is each one of these signature bit blocks. The extraction process thus require some extra hardware.

3 Design Example: CIC Filter

A 3-stage CIC decimation filter [8] was chosen as study case for embedding a 128-bit signature. It includes 2C-to-RNS conversion, four parallel RNS CIC channels and ε-CRT-based RNS-to-2C conversion [9]. Design examples were implemented using the Xilinx Virtex-II device family. Table 1 shows results for the conventional RNS-based CIC filter and the signed RNS-based CIC filter proposed in this paper. Analysis of these results shows that the area increase for the signed filters is less than 6% while throughput penalization is negligible. In fact, there is a small performance increase that can be explained because of a better Place&Route effort by the design tool. It must also be noted that the area increase is a fixed quantity for this signature length, so for more complex applications the area increase will be even less noticeable.

Table 1. Summary of simulation results

Speed grade	RNS-based CIC filter		signed RNS-based CIC filter			
	SLICEs	F_{max} (MHz)	SLICEs	Area increase	F_{max} (MHz)	Speed reduction
-6	420	106.85	445	5.95%	107.19	-0.32%
-5	420	101.34	445	5.95%	103.50	-2.13%
-4	420	88.63	445	5.95%	91.26	-2.97%

4 Conclusions

This paper describes a procedure that exploits empty memory positions in the look-up tables used in RNS-based hardware to embed a digital signature in RNS-based circuits. This allows its IPP usage for IP cores since it is extremely difficult to detect and/or remove this signature. An RNS-based CIC filter was used to embed a 128-bit signature with negligible penalties in both performance and area. Furthermore, the procedure described can be easily extended to any FPL hardware using look-up tables. In the case that the design to sign does not force the use of look-up tables, it is still possible to map into tables part of the design not included in the critical path, so the proposed method may be applied. Further work will be directed to this issue, as well as to the optimization of the resources and security of the embedded signature.

Acknowledgements. Authors wish to acknowledge financial support from the Dirección General de Investigación (Spain) under project TIC2002-02227. CAD tools and supporting material were provided by Xilinx Inc, trough Xilinx University Program agreements.

References

1. A.B. Karhng, J. Lach, W.H. Mangione-Smith, S. Mantik, I.L. Markov, M.Potkonjak, P. Tucker, H. Wang and G. Wolfe, “Constraint-Based Watermarking Techniques for Design IP Protection”, IEEE Transactions on Computer-Aided Design, 20(10):1236-52, 2001.
2. A.B. Kahng, J. Lach, W.H. Mangione-Smith, S. Mantik, I.L. Markov, M. Potkonjak, P. Tucker, H. Wang, G. Wolfe, "Watermarking Techniques for Intellectual Property Protection," Design Automation Conference, 776-81, 1998.
3. R.L. Rivest,“The MD5 Message Digest Algorithm ”Rivest. Internet RFC 1321 (1992).
4. Szabo, N. S., Tanaka, R. I.: Residue Arithmetic and its Applications to Computer Technology. McGraw-Hill, New York, 1967.
5. Hamann, V., Sprachmann, M.: “Fast Residual Arithmetic with FPGAs,” Workshop on Design Methodologies for Microelectronics, 1995.
6. U. Meyer-Bäse, A. Garcia and F. J. Taylor, “Implementation of a Communications Channelizer using FPGAs and RNS Arithmetic”, Journal of VLSI Signal Processing, vol. 28, no. 1/2, pp. 115-128, May 2001.
7. J. Lach, W.H. Mangione-Smith, M. Potkonjak, "Fingerprinting Techniques for Field Programmable Gate Array Intellectual Property Protection," IEEE Transactions on Computer-Aided Design, 20(10):1253-61, October 2001.
8. A. García, U. Meyer-Bäse and F. J. Taylor, “Pipelined Hogenauer CIC Filters Using Field-Programmable Logic and Residue Number System”. Proc. 1998 IEEE Int. Conf. on Acoustics, Speech and Signal Processing, Seattle, WA, vol.5, pp. 3085-3088, May 1998.
9. M. Griffin, M. Sousa and F. Taylor, “Efficient Scaling in the Residue Number System”, Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 1075-1078, 1989.

FPGA Implementation of a Tool Breakage Detection Algorithm in CNC Milling Machines

René de Jesús Romero-Troncoso[1] and Gilberto Herrera Ruiz[2]

[1] FIMEE – U. de Gto., Tampico 912, Col. Bellavista
36720 Salamanca, Gto., México
troncoso@salamanca.ugto.mx

[2] Universidad Autónoma de Querétaro, Facultad de Ingeniería, Cerro de las campanas s/n
76010 Querétaro, Qro., México
gherrera@sunserver.uaq.mx

Abstract. In this paper, an on-line tool breakage detection algorithm in CNC milling machines is implemented into a single 32,000-gate FPGA from Actel. The tool breakage detection algorithm is based on three pipelined processing units: a two-channel modulo operator, a one dimension wavelet transform and an asymmetry correlator. The two-channel modulo operator performs two twelve-bit square operations and a 25-bit square root. The one dimension wavelet transform performs a 256x8 point matrix per a 256 point vector multiplication over the incoming data from the modulo operator. The third processor performs an asymmetry correlation over the wavelet data to give a single value which contains the estimation of the tool condition. The overall processing unit cost is kept low by using optimized numeric digital structures, suited to fit into a 32,000-gate FPGA while allowing the system to give on-line tool condition estimation. Results are presented in order to show overall system performance.

1 Introduction

On-line tool breakage detection has been one of the main concerns on CNC milling machine processes in order to improve the overall system efficiency because of the manufacturing cost; typical manufactured pieces to be used as extrusive moldings can cost 3,000 USD and up, and tool wear and breakage can irreversibly damage the piece if it is not detected on-line. Three different aspects have to be considered to solve the problem: the sensor and data acquisition, the digital signal processing algorithm and the implementation cost. From the sensor point of view, Prickett and Johns [1] present an overview to end milling tool monitoring approaches up to date. Further signal conditioning and data acquisition of the cutting force signal provide the digital system with the suited data in order to perform the numeric algorithm to estimate tool condition. One dimension wavelet transform has proven its efficiency for the task as reported by Kasashima et. al. [2]. In addition, implementation cost is a major concern as reported implementations of commercially available solutions for on-line tool breakage detection systems are based on general purpose multiple DSP systems, making the implementation highly expensive for most real-world applications, Bejhem and Nicolescu [3].

In this paper a one dimension discrete wavelet transform (DWT) over the cutting force resultant with non-linear autocorrelation, used as a novelty compared with

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1142–1145, 2004.

previously reported works, is the algorithm used and the implementation is in the form of an application specific hardware signal processing (HSP) unit. The system on-a-chip (SOC) approach for HSP is implemented into an FPGA, which significantly improves the price/performance ratio over commercially available systems.

2 DSP Computation

The milling process model assumes that the cutting force is proportional to the volume of workpiece material removed by each tooth, Trang and Lee [4]. The cutting force patterns are the resultant of the cutting forces at the *X* and *Y* axis for end milling as stated in equation 1, where *F* is the resultant cutting force, F_x is the cutting force at *X* axis and F_Y is the cutting force at *Y* axis. DWT, Daubechies [5], provides a time-frequency representation of the original signal in a decimated form which means that depending on the applied detail level, the result will show the time-domain samples at the decimated frequency in a compressed form. For the present development, a 256 point original cutting force data is compressed via level-5 detail, Daubechies 12 DWT as stated in equation 2.

$$F = \sqrt{F_X^2 + F_Y^2} \ . \tag{1}$$

$$B = W_5 W_4 W_3 W_2 W_1 F \ . \tag{2}$$

DWT output *B* can be seen as a multi-rate finite impulse response (FIR) filter-bank [6] which, after matrix multiplication of the successive 5 down to 1 DWT matrix by the 256-row matrix *F*, will give an 8-row *B* vector. The proposed method to give a single-parameter estimation of the tool conditions is based on a non-linear autocorrelation function to calculate the asymmetry between the cutting force waveform of each insert at the tool-head. The asymmetry is defined as the point-to-point variance between cutting force signals for each insert in a full revolution of the tool-head, applied to *B* as stated in equation 3.

$$A = \sum_{i=1}^{4} \left(B_{i+4} - B_i\right)^2 \ . \tag{3}$$

3 HSP Architecture

The HSP unit contains three application specific DSP units with synchronization, control data acquisition system (DAS) interfacing and PC interface as shown in Fig. 1. Three application specific DSPs are necessary in order to compute, on-line, all the processes involved in breakage detection: Square-adder, DWT and asymmetry units.

The square-adder unit performs resultant force computation of the incoming data from the DAS, equation 1. A fully combinational fixed-point square block based on an adder-tree special case multiplier is used, Parhami [7]. Detail level-5 Daubechies-

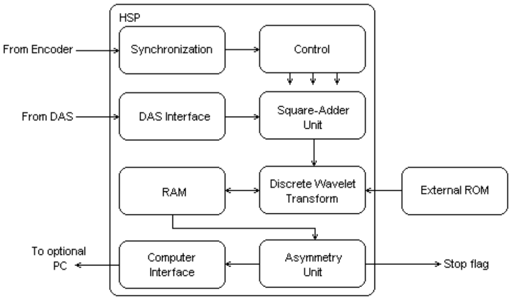


Fig. 1. Hardware signal processor general scope

12 DWT computation is performed by the second application specific DSP. The DSP unit is based on a 16-bit x 16-bit multiplier accumulator (MAC), a double-bank dual-port RAM and an external ROM containing DWT coefficients; this unit is based on the radix-4 sequential Booth algorithm [7]. Intermediate data is stored in the active bank of the RAM until one revolution on the cutting tool head is completed, equation 2. Once the DWT is computed, RAM banks are switched and the next DWT starts to be computed in the alternate bank while the original bank keeps the information that is accessed by the autocorrelation processor. FIR coefficients for the detail level-5 DWT are stored in an external ROM, which are pre-fetched during the multiplication cycle. The third DSP unit is used for autocorrelation computation and it is based on a fully combinational substractor and a 16x16 MAC, in square configuration, equation 3. The active RAM bank for the autocorrelation DSP is the inactive RAM bank for the wavelet transform DSP in order to avoid data collision and allowing pipeline computation.

The breakage detection process requires precise synchronization among the three DSP units and the DAS. The system is controlled by an application specific sequencer, which controls the DSP units by providing the handshake signal to keep consistency among data paths. Additional synchronization is necessary in order to fit timing requirements in the DAS. The synchronization sequencer provides precise timing to all processes involved. The HSP unit was implemented into an ACTEL 54SX32A-TQ144 with 32,000 usable gates FPGA with standard speed at an operating external frequency of 10 MHz.

4 Results

The experiments were set on a retrofitted to CNC milling machine, model FNK25A, with a two-insert tool head. Undamaged new carbide inserts were used for the experiments. The milling process started and it was stopped after tool breakage was detected. At the beginning of the milling process, where the tools are new and undamaged, the asymmetry value of the DWT is close to zero. When the milling process continues, one tool starts to wear and the asymmetry value is increased. Finally, when the tool is broken, the asymmetry is increased significantly and a peak in the asymmetry value can identify the cycle at which the breakage happens. After tool breakage, the detector system indicates tool condition degradation. It is shown that DWT with asymmetry, as a non-linear autocorrelation function, gives an estimation of the tool condition by comparison between the measured cutting forces at the two inserts, indicating tool wear and tool breakage.

5 Concluding Remarks

An HSP unit that has been designed and implemented into a single FPGA for a system on-a-chip design approach in a stand-alone special purpose processor performed the highly intensive computation requirements of the algorithm. The HSP unit has three embedded application specific DSPs interconnected in a multiprocessing pipelined architecture to allow the system on-line tool breakage detection by giving a single data result, 38.9 µs after the revolution is completed. Experimental results show that the tool breakage detection system works properly and the complete prototype containing the power supply, analog filtering and signal conditioning, special purpose DAS and HSP unit is estimated to be under 300 USD.

Algorithm complexity involves 512 square operations, 256 additions and 256 square root extractions for equation 1; 2048 multiplications and 2048 additions for equation 2; 4 square operations and 8 substractions for equation 3; over-all system control and DAS synchronization. A Pentium 4 based PC with a general purpose DAS were used to compare results with HSP performance giving the tool condition for a revolution after 4 s, which is totally unsuited for on-line detection. On the other side, general purpose DSP with 100 MOPS data processing speed will require 100 µs for numerical computation only; and synchronization, matrix data addressing, multiprocessing and DAS control requires extra clock cycles. In addition, it must be considered overall system cost. Investing in the development of HSP systems for status monitoring and control helps to improve the price/performance ratio of the instrumentation on CNC machines by providing an inexpensive built-in solution.

References

1. Prickett, P. W., Johns, C.: An overview of approaches to end milling tool monitoring. International Journal of Machine Tools & Manufacture Vol. 39 (1999) 105-122
2. Kasashima, K., Mori, K., Herrera-Ruiz, G.: Diagnosing cutting tool conditions in milling using wavelet transform. In: 7th International Conference on Production/Precision Engineering, JSPE. (Chiba, Japan): September 15-17 (1994) 339-344
3. Bejhem, M., Nicolescu, C. M.: Machining condition monitoring for automation. In: 3rd International Conference on Machining & Grinding, Society of Manufacturing Engineers. (Cincinnati, Ohio): October 4-7 (1999) MR99-231
4. Trang, Y. S., Lee, B. Y.: Use of model-based cutting simulation system for tool breakage monitoring in milling. International Journal of Machine Tools and Manufacture Vol. 32 (1992) 641-649
5. Daubechies, I.: Orthonormal bases of compactly supported wavelets. Communications on Pure and Applied Mathematics Vol. 41 (1988) 909-996
6. Vetterli, M., Herley, C.: Wavelets and filter banks: theory and design. IEEE Transactions on Signal Processing Vol. 40 (1992) 2207-2232
7. Parhami, B.: Computer arithmetic, Algorithms and hardware design. Oxford university press (2000)

Implementation of a 3-D Switching Median Filtering Scheme with an Adaptive LUM-Based Noise Detector

Miloš Drutarovský[1] and Viktor Fischer[2]

[1] Department of Electronics and Multimedia Communications,
Technical University of Košice, Park Komenského 13, 04120 Košice, Slovakia
Milos.Drutarovsky@tuke.sk

[2] Laboratoire Traitement du Signal et Instrumentation, Unité Mixte de Recherche
CNRS 5516, Université Jean Monnet, 10, rue Barrouin, 42000 Saint-Etienne, France
fischer@univ-st-etienne.fr

Abstract. We present a Field Programmable Logic Devices (FPLDs) based implementation of a scalable filter architecture capable of detecting and removing impulsive noise in image sequences. The adaptive filter architecture is built using switching spatiotemporal filtering scheme and robust Lower-Upper-Middle (LUM) based noise detector. It uses highly optimized bit-serial pipelined implementation in Altera FPLDs. The proposed architecture provides real-time performance for 3-D image processing with sampling frequencies up to 97 Mpixels/second.

1 Introduction

Digital images are often contaminated by impulsive noise introduced into the images during image acquisition and/or transmission [1]. Switching Median Filters (MFs) [2] adopt robust smoothing capabilities of the well-known MF. Recent hardware and software developments support the extension of the adaptive filtering schemes into video filtering [1,3]. Note that many switching methods have a serious limitation to be extended into the three-dimensional (3-D) case, because of their computational complexity.

2 LUM Smoother-Based Adaptive Switching Scheme

The 3-D, spatiotemporal adaptive filter introduced in [4] uses, instead of the switching among multiple smoothing levels [1,5], a flexible noise detector based on robust Lower-Upper-Middle (LUM) smoothing characteristics [6,7]. Since the LUM smoother requires data sorting, it leads to an ineffective FPLD implementation. This disadvantage is removed using the design employing LUM-based Positive Boolean Functions (PBFs) [8] and a bit-serial structure [9].

Let $W = \{x_1, x_2, \ldots, x_N\}$ be an input set of the samples within the running processing window and N an integer number denoting window size. The corresponding ordered set $x_{(1)} \leq x_{(2)} \leq \ldots \leq x_{(N)}$ contains order statistics $x_{(i)} \in W$,

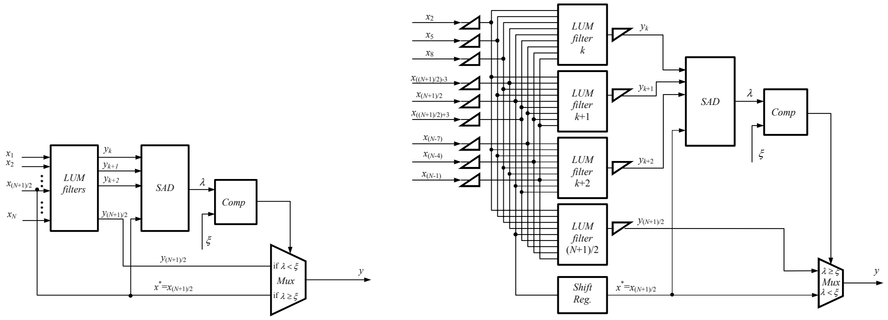
J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1146–1148, 2004.

for $i = 1, 2, ..., N$. The output of the standard LUM smoother [6] is defined by

$$y_k = med\{x_{(k)}, x^*, x_{(N-k+1)}\} \tag{1}$$

where med is a median operator and $k = 1, 2, \ldots, (N+1)/2$ denotes a smoothing parameter. The sample $x^* = x_{(N+1)/2}$ is the window center, $x_{(k)}$ is a lower- and $x_{(N-k+1)}$ is an upper-order statistic such that $x_{(k)} \leq x_{(N-k+1)}$.

Fig. 1(a) depicts the structure of the method. The parameter ξ is the non-negative threshold to be compared with $\lambda = \sum_k^{k+2} |x^* - y_k|$, where y_k, y_{k+1}, y_{k+2} represent the LUM smoothers defined for the consecutive values of the smoothing parameter k. This aggregated absolute difference between the central sample and the LUM smoother outputs constitutes computationally simple operations, which make the switching rule robust [4]. If $\lambda \geq \xi$, the output of the switching scheme is the median $y_{(N+1)/2}$. Otherwise, the central sample x^* is noise-free and remains unchanged. The default setting of k in defining λ is 6. The corresponding ξ should be set to 60.



(a) Filtering scheme

(b) Pipelined structure

Fig. 1. Proposed 3-D adaptive filter

3 Proposed Pipelined Realization

The complete pipelined 3-D switching median based on the LUM smoothing characteristics is depicted in Fig. 1(b). The spatiotemporal filter utilizes a 3×3×3 cube processing window ($N = 27$). Four pipelined LUM smoothers with $B = 8$ levels (for $k, k+1, k+2$ and $(N+1)/2$) compute B levels of input samples concurrently. It can be easily seen that 9 new samples appear in the input of the proposed filter for each new window position. Therefore, 9 input shift register blocks (triangles) are necessary. Filter output latency is $(B+1)$ clock periods.

The switching filtering scheme has been mapped onto selected Altera FPLDs. We synthesized them using Altera Quartus II v. 2.2 and Leonardo Spectrum

Table 1. Mapping of pipelined structure of the proposed adaptive video filtering framework into three Altera FPLD families using Leonardo Spectrum v.2002e.16 compiler

Device	EP20K160-1		EP1C6-6		EP1S25-5	
Parameter	LCs	f (MHz)	LCs	f (MHz)	LCs	f (MHz)
Result	4929	55.7	4496	91.0	4502	97.3

Level 3 v. 2002e.16 VHDL compilers. Placement, routing and timing analysis have been realized using Quartus II v.2.2. VHDL output generated by the fitter was used as simulation input for ModelSim v. 5.6 VHDL simulator. Output values have been compared with Matlab-generated test values in an automatic testbench procedure.

Table 1 summarizes the results corresponding to the mapping of the complete pipelined version of the switching scheme into three Altera FPLDs. Required absolute value addition (SAD) and comparison (Comp) blocks are realized using standard Library of Parameterized Modules (functions lpm_{abs} and $\text{lpm}_{\text{compare}}$).

We can conclude that the filter area is relatively small (only 17% of the Stratix EP1S25 device used in the Altera Stratix DSP evaluation board [10] that was used for filter implementation and testing) and very fast. The unused part of the device (almost 80%) was still big enough to constitute necessary resources for implementing additional image processing functions utilized in video compression, analysis and segmentation.

References

1. Lukac, R., Marchevsky, S.: LUM smoother with smooth control for noisy image sequences. EURASIP J. on Applied Signal Processing (2001) 110–120
2. Zhang, S., Karim, M.: A new impulse detector for switching median filters. IEEE Signal Processing Letters **9** (2002) 360–363
3. Kim, J., Park, H.: Adaptive 3-D median filtering for restoration of an image sequence corrupted by impulse noise. Signal Proc.: Image Comm. (2001) 657–668
4. Lukac, R., Fischer, V., Motyl, G., Drutarovský, M.: Adaptive video filtering framework. Int. Journal of Imaging Systems and Technology (submitted 2003)
5. Fischer, V., Drutarovský, M., Lukac, R.: Implementation of 3-D adaptive LUM smoother in reconfigurable hardware. LNCS 2438, (Springer-Verlag) 720–729
6. Hardie, R., Boncelet, C.: LUM filters: A class of rank-order-based filters for smoothing and sharpening. IEEE Trans. Signal Processing **41** (1993) 1061–1076
7. Lukac, R., Marchevsky, S.: Boolean expression of LUM smoothers. IEEE Signal Processing Lett. **8** (2001) 292–294
8. Lukac, R.: Binary LUM smoothing. IEEE Signal Processing Lett. **9** (2002) 400–403
9. Chen, K.: Bit-serial realizations of a class of nonlinear filters based on positive Boolean functions. IEEE Trans. Circuits and Systems **36** (1989) 785–794
10. Stratix EP1S25 DSP Development Board, www.altera.com.

Using Logarithmic Arithmetic to Implement the Recursive Least Squares (QR) Algorithm in FPGA*

Jan Schier and Antonín Heřmánek

Inst. of Information Theory and Automation
Academy of Sciences of the Czech Republic
schier@utia.cas.cz

Abstract. In this paper, we outline an FPGA implementation of the QR update algorithm with Givens rotations using the High Speed Logarithmic Arithmetic (HSLA) library. An advantage of this approach is low latency and accurate computation (comparable with single-precision floating point) of the operations.

Keywords: Givens rotations, logarithmic arithmetic, FPGA

1 Introduction

The QR-RLS algorithm is, thanks to its internal parallelism (it can be represented by regular parallel array of communicating processors), well suited for use in beamforming applications, where high data throughput is often critical.

In the case of its implementation in FPGA, one problem has to be tackled: the computations used in the rotations use floating-point operations including division and (possibly) square-root. Two approaches can be used: either is the algorithm transformed to a fixed-point domain, or a floating-point library of operators is used. A disadvantage of the first one is that the wordlength requirements are higher to achieve an SNR performance comparable with the floating point representation [1].

For the floating-point implementation, several solutions are available: the libraries developed by QuinetiQ Real-Time Systems division, by Celoxica or by Nallatech can be mentioned as examples. In our work, the High-Speed Logarithmic Arithmetic library (HSLA[1]) [2] is used.

The paper is organized in the following way: first, the recursive QR-RLS algorithm will be briefly reviewed. Then, the HSLA library will be briefly described and the architecture of our implementation of the QR algorithm will be outlined, followed by the resource utilization figures.

2 Givens Rotations and the QR Algorithm

The QR-RLS algorithm [3] is based on an update of an upper triangular matrix $\mathbf{R}$ with a data vector $\mathbf{u}$ using series of the Givens rotations. It can be mapped to a triangular array

* This work was partly supported by the Ministry of Education of the Czech Republic under the Project LN00B096.

[1] http://www.utia.cas.cz/zs/home.php?ids=hsla

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1149–1151, 2004.

of processors, with the diagonal ones computing c_i and s_i and updating r_i:

$$c_i = r_i/\sqrt{r_i^2 + u_i^2} \qquad s_i = u_i/\sqrt{r_i^2 + u_i^2} \qquad r'^2_i = \lambda^2(r_i^2 + u_i^2) \tag{1}$$

and the off-diagonal ones updating r_{ij} and u_{ij}:

$$r'_{ij} = \lambda(cr_{ij} + su_{ij}) \qquad u_{i+1,j} = -sr_{ij} + cu_{ij} \tag{2}$$

3 HSLA

The High-Speed Logarithmic Arithmetic library (HSLA) [2] is based on computations with the logarithmic equivalents of floating-point numbers. The numbers are represented by an integer part, always 8 bit long, a fraction part, 10/23 bit long (in the 19/32 bit library version) and a sign bit.

The operations are transformed accordingly: addition and subtraction take form

$$\log_2(x \pm y) = a + \log_2(1 \pm 2^{b-a}), \quad \text{where} \quad a = \log_2|x|, \quad b = \log_2|y|, \tag{3}$$

while multiplication and division transform to a simple fixed-point addition or subtraction and the square-root operation becomes a right-shift operation.

The non-linear function $\log_2(1 \pm 2^r)$ used in (3) is evaluated using a first-order Taylor-series approximation. Compared to traditional solutions, where the size of the look-up tables represents major problem, here they are kept small by using an error correction mechanism and a range-shift algorithm. For implementation details see [2].

The addition/subtraction is fully pipelined in the HSLA implementation and has 8 clock cycles latency, other operations have 1 clock cycle latency.

4 Implementation

The experimental design has been implemented in the Xilinx Virtex-E XCV2000E-6 device, using the HandelC language from Celoxica. In order to keep the design simple, only one instance of diagonal and one of the off-diagonal processor has been implemented, as shown in Fig. 1. The parameters of the 19-bit implementation are summarized in the table below:

Resource utilization: 19-bit, XCV2000E-6-BG560				
SLICEs	4492/19200	23%	CLK Freq.	75 MHz
BRAMs	30/160	18%	FLOPS	147 MFLOPS
TBUFs	288/19520	1%		
2x twin ADD/SUB, 11x MUL/DIV/SQRT				

The data flows in the algorithm allow for fully pipelined implementation of the operations in both diagonal and off-diagonal processor.

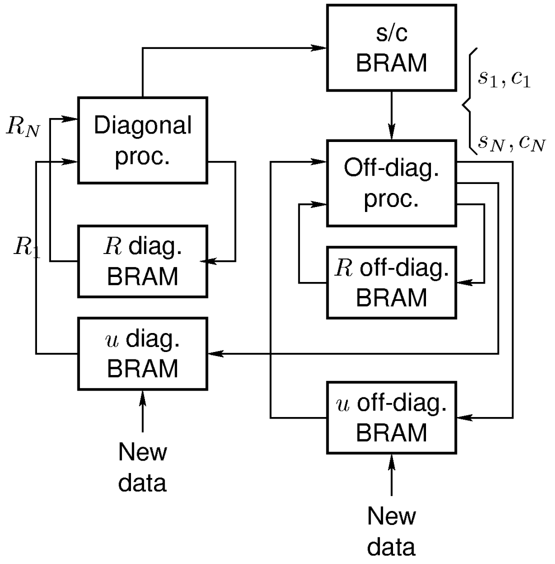


Fig. 1. Architecture of the QR implementation

5 Conclusions

In the paper, we describe an FPGA implementation of the recursive QR-RLS algorithm using the HSLA arithmetic library to implement floating-point arithmetic operations. In our opinion, with the low-latency operators and bit-exact Matlab/HandelC modules, this library represents an interesting option for development of the signal processing applications.

In our experimental implementation, the XCV2000E-6-BG560 device was used. As mentioned in the text, major limitation to the number of processors implemented in a single FPGA device is the size of look-up table used by the ADD/SUB unit. Our design uses roughly 25% of the device; in 32-bit accuracy, only one unit would fit into this device. In future, we have to improve the efficiency of the ADD/SUB units utilization.

References

1. R. Walke, R.W.M. Smith, and G. Lightbody, “Architectures for adaptive weight calculation on ASIC and FPGA,” in *Proc. 33rd Asilomar Conference on Signals, Systems and Computers*, 1999.
2. R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, and C. Softley, “Logarithmic number system and floating-point arithmetics on FPGA,” in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, M. Glesner, P. Zipf, and M. Renovell, Eds., Berlin, 2002, vol. 2438 of *Lecture Notes in Computer Science*, pp. 627–636, Springer.
3. J.G. McWhirter and I.K. Proudler, *Adaptive System Identification and Signal Processing Algorithms*, chapter The QR family, pp. 260–320, Prentice-Hall, Inc., 1993.

FPGA Implementation of a Vision-Based Motion Estimation Algorithm for an Underwater Robot

Viorela Ila[1], Rafael Garcia[1], Francois Charot[2], and Joan Batlle[1]

[1] University of Girona, C/ L.Santalo s/n, 17071, Girona, Spain
[2] IRISA/INRIA, Campus de Beaulieu 35042 Rennes Cedex, France

Abstract. This paper presents an FPGA implementation for real-time motion estimation of an underwater robot using computer vision. The algorithm searches for correspondences of a given number of interest points for every image acquired by the camera and some previous reference images. In order to minimise the lighting problems, normalised correlation is used as similarity measurement to match corresponding points in different images. The complexity of normalised correlation criteria determined two main parts in our hardware implementation: an array of Processing Elements (PE) and Post Processing Element (PPE).

1 Introduction

The motion of an underwater robot can be recovered from the camera motion by processing consecutive images acquired by an on-board down-looking camera. Point correspondences have to be found for motion estimation. Due to its regular processing scheme, parallel implementation of the correspondence problem can be an adequate approach to reduce the computation time. The computation can be break down into blocks to be processed in parallel. An extensive literature exists about array architectures applied to image processing, especially in Block Matching Algorithms (BMA) for motion estimation [1,4,6]. While in full search BMA the image is divided into blocks and the algorithm is looking for matching every block in a frame, our approach is looking for correspondences of the region in the neighborhood of specific points. These points are scene features which can be reliably found when the camera moves from one location to another, even when lighting conditions of the scene change. In our previous work we proposed a hardware implementation of interest points detection [3]. On the other hand, a complex error measurement criteria such as normalised correlation has proved to be very adequate to underwater imaging [2]. The computation of this error is divided into two parts. In the first part a parallel array architecture is used for multiple data computation. Then, a computational block containing multipliers, subtractors, square root and division computes the error measurement.

2 Hardware Implementation of the Motion Estimation Algorithm

A correlation methodology provides, for each interest point (x_c, y_c) of the current image I_c, its corresponding match (x_r, y_r) in the reference image I_r. The

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1152–1154, 2004.

correlation score is defined as the covariance between the grey levels of a region defined by the *correlation window* in the current image and the same region defined in the reference image. In order to simplify the hardware implementation, we propose a breaking down of a normalised correlation criteria successfully applied to underwater imaging [2]. There are five sums to be computed:

$$\begin{aligned}
&sum_1 = \sum_{i=-\alpha}^{\alpha}\sum_{j=-\alpha}^{\alpha} I_c(x_c+i, y_c+j); \quad sum_2 = \sum_{i=-\alpha}^{\alpha}\sum_{j=-\alpha}^{\alpha} I_c(x_c+i, y_c+j)^2;\\
&sum_3 = \sum_{i=-\alpha}^{\alpha}\sum_{j=-\alpha}^{\alpha} I_c(x_c+i, y_c+j)\cdot I_r(x_r+i, y_r+j);\\
&sum_4 = \sum_{i=-\alpha}^{\alpha}\sum_{j=-\alpha}^{\alpha} I_r(x_r+i, y_r+j)^2; \quad sum_5 = \sum_{i=-\alpha}^{\alpha}\sum_{j=-\alpha}^{\alpha} I_r(x_r+i, y_r+j).
\end{aligned} \tag{1}$$

where $(2\alpha+1)\times(2\alpha+1)$ is the size of the correlation window. Then, correlation score becomes:

$$C = \frac{sum_3 - \frac{1}{(2\alpha+1)^2}\cdot sum_1 \cdot sum_5}{\frac{1}{(2\alpha+1)^2}\cdot\sqrt{[(2\alpha+1)^2\cdot sum_2 - sum_1^2]\cdot[(2\alpha+1)^2\cdot sum_4 - sum_5^2]}} \tag{2}$$

This leads to an easy parallel implementation, while each Processing Element (PE) of an array architecture executes in parallel the computation of these five sums. The Post Processing Element (PPE) performs the remaining computation.

2.1 Hardware Implementation

When mapping an algorithm into an array of processors, the problem is to access multiple data to feed all the Processing Elements (PE) at the same time. Adopting the solution of local data exchange between PEs, two parallel memory accesses are used for the same reference image and one for the current image [6]. Once read from memory, the data are broadcasted to every PE. Buffers are used to delay data and multiplexers to switch between data. For higher utilisation efficiency of the architecture, the size of the search window must be defined according to the size of the correlation window by the equation $p = 2\alpha$, where $(2p+1)\times(2p+1)$ is the size of the search window. The number of PEs also depends on the size of the correlation window and is equal to $(2\alpha+1)$. One PE is in charge of the parallel computation of the five sums defined in equation (1). Two accumulations and three multiplication-accumulations are executed in parallel.

The results from the array of PEs are pipelined into the Post Processing Element (PPE). The PPE computes the correlation criteria defined in the equation (2). Seven multiplications, three subtraction, one 64-bit square root and one 32-bit division have to be implemented in hardware. Parallel implementation of these operations is performed. The square root implementation is based on the non-restoring algorithm proposed by Li [5]. The advantage of this method is the reduced space occupied on the FPGA device and that it generates an exact result value. The last step of the algorithm compares all the error measurements corresponding to every candidate match. The result of the algorithm are the coordinates of the pixel with the biggest value for the correlation score.

The algorithm was implemented, simulated and tested using Quartus design software from Altera. The design was synthesised for the Altera Stratix EP1S25F672 device. This permits a complex analysis of the functionality, timing, logic elements (LE) and dedicated multipliers (DM) blocks utilization. The array of PEs can be synthesize using 3360 LEs and 45 DMs and the PPE element using 1957 LEs and 30 DMs among those available on the FPGA device. Running the motion estimation algorithm on a PC/104+ computer requires about 2 seconds. Our implementation impose the time constraint to 0.04s (video-rate) for execution of this algorithm, clocking the computation according to this. The implementation and delay information for corner detection were detailed in our previous work [3]. For N given interest points and being t_{PPE}, the computational time required by PPE, the delay introduced by the parallel implementation of the matching algorithm is given by:

$$T_C = [2 \cdot [(2\alpha + 1)^2 \cdot [(2p + 1) - 2\alpha] + 2\alpha] + t_{PPE}] \cdot N \tag{3}$$

3 Conclusions

This paper proposes an FPGA implementation to solve the correspondence problem in a motion estimation algorithm. The matching algorithm is divided into two parts: an efficient array of processing elements for multiple data processing, and a post processing element in charge of the high-level computation of normalised correlation. An optimal implementation for the square root operation has been applied. The algorithm was implemented, simulated and tested using Quartus design software from Altera. A complex analysis of functionality, timing, required elements and dedicated multiplier was carried out.

References

1. P. Baglietto, M. Maresca, A. Migliaro, and M. Migliardi. Parallel implementation of the full search block matching algorithm for motion estimation. In *International Conference on Application Specific Array Processors*, pages 182–192, July 1995.
2. X. Cufí, R. Garcia, and R. Ridao. An approach to vision-based station keeping for an unmanned underwater vehicle. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 799–804, Lausanne, 2002.
3. V. Ila, R. Garcia, and F. Charot. Proposal of a parallel architecture for a motion detection algorithm. In *International Conference on Pattern Recognition*, Cambridge, Aug. 2004.
4. T. Komarek and P. Pirsch. Array architectures for block matching algorithms. *IEEE Transactions on Circuits and Systems*, 36:1301 –1308, 10 , Oct 1989.
5. W. Li and W. Chu. A new non-restoring square root algorithm and its vlsi implementations. In *1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 538 – 544, 7-9 Oct. 1996.
6. K.-M. Yang, M.-T. Sun, and L. Wu. A family of VLSI designs for the motion compensation block-matching algorithm. *IEEE Transactions on Circuits and Systems*, pages 1317 –1325, Oct. 1989.

Real-Time Detection of Moving Objects

Hiroaki Niitsuma and Tsutomu Maruyama

Institute of Engineering Mechanics and Systems, University of Tsukuba
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 JAPAN
niitsuma@darwin.esys.tsukuba.ac.jp

Abstract. In this paper, we describe a compact system for real-time detection of moving objects. In this system, we realized real-time computation of the optical flow and the stereo vision by area-based matching with one FPGA. By combining the optical flow and the stereo vision, we can detect moving objects and distances to the objects, which are very important for vision systems of autonomous vehicles. The system implemented on XC2V6000 runs at 68 MHz, which is fast enough to process more than 30 images (640×480 pixels) in one second.

1 Introduction

Compact real-time vision systems are very important for autonomous vehicles. FPGAs are ideal devices for the compact systems, because FPGAs can support many kinds of functions by reconfiguration depending on situations. In this paper, we describe a compact system for real-time detection of moving objects by the optical flow and the stereo vision. In order to accelerate the performance of the optical flow by hardware, many systems have been proposed to date[1][2][3][4]. In those systems, sizes of images are limited or only sparse vector fields are generated in order to achieve real-time processing. In our system, dense vector field (vectors for all pixels in images) for standard size image (640×480) can be generated at video-rate with only one FPGA. In order to achieve high performance with a small circuit, intermediate results in the computation along x axis are stored in the circuit and reused w times ($w \times w$ is the size of windows used for the area-based matching), while operations along y axis are re-executed w times.

2 Computation Method of the Optical Flow

In the optical flow and the stereo vision, the corresponding point to a given point in an image is searched in another image. Area-based (or correlation-based) algorithms match small windows centered at a given pixel to find corresponding points between the two images. We used the SAD (Sum of Absolute Difference) algorithm for the matching. In Figure 1, a small window centered at (x, y) (Figure 1(a)) is compared with all windows in its target area centered at (x, y) (Figure 1(b)(c)(d)). When the size of the target area is $(k+w-1) \times (k+w-1)$, there are $k \times k$ windows in the target area, and $k \times k$ SADs (Sum of Absolute Differences) are calculated. Then, the window which gives the minimum SAD is chosen, and its center point (x', y') is considered as the corresponding point to (x, y).

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1155–1157, 2004.

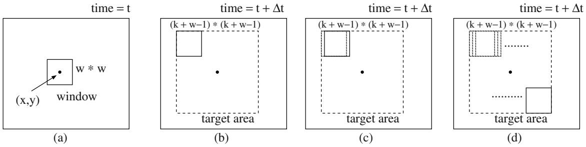


Fig. 1. Area-based Matching in the Optical Flow

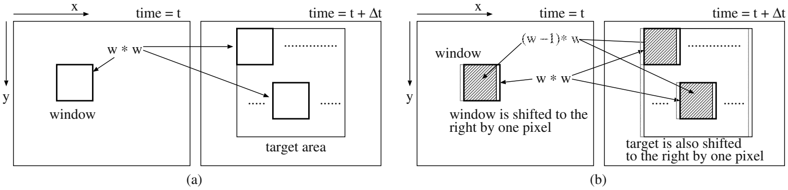


Fig. 2. Reuse of the Intermediate Results

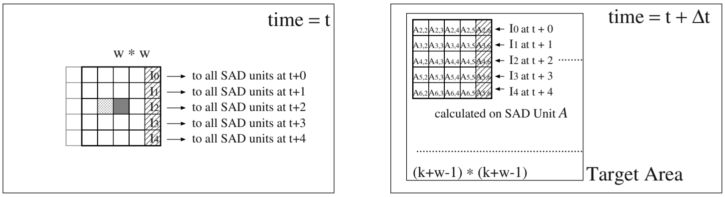


Fig. 3. An Implementation Method by Recalculation

In Figure 2(a), suppose that we have calculated $k \times k$ SADs and chosen the minimum of them to obtain one vector. Then, the window is shifted to the right by one pixel to obtain next vector (Figure 2(b)). At this point of time, pixels in a rectangle with slanting lines in the shifted window ($time = t$) are already compared with pixels in rectangles with slanting lines in its target area ($time = t + \Delta t$) during the computation of the previous vector. Therefore, by storing $k \times k \times (w - 1) \times w$ ADs (Absolute Differences) calculated in Figure 2(a), the number of new ADs to obtain the new vector can be reduced to $k \times k \times w$.

In our computation method, these $k \times k \times w$ ADs are calculated in parallel with $k \times k$ SAD units, while $k \times k \times (w - 1) \times w$ ADs are stored and reused. In Figure 3, I_0 is broadcasted to $k \times k$ SAD units first, and $k \times k$ ADs for I_0 ($|A_{2,6} - I_0|$ and so on) are calculated in the $k \times k$ SAD units in parallel. In the same way, ADs for $I_j(j = 1, 4)$ are calculated sequentially. These calculations takes w clock cycles in total. These ADs are, then, summed up (rectangles with slanting lines in Figure 3), and held on the shift registers. The sums held on the shift registers are used w times to calculate w SADs and discarded after shifted w times.

Figure 4 shows an array of SAD units on a register array to deliver pixel data in $time = t + \Delta t$ to the SAD units. In Figure 4, $k \times k$ ($k = 21$) rectangles in dark gray are SAD units, and each SAD unit calculates one SAD in w (w = 7) clock cycles. In order to reduce memory access operations, (1) one pixel (*d6*) is read into FPGA, and set on a register *r6*, (2) pixel data of six (w-1) rows stored in three Block RAMs (*Block RAM #0,#1 and #2*; two rows can be stored in one Block RAM) are also read out (*d0-d5*) and set on registers (*r0-r5*),

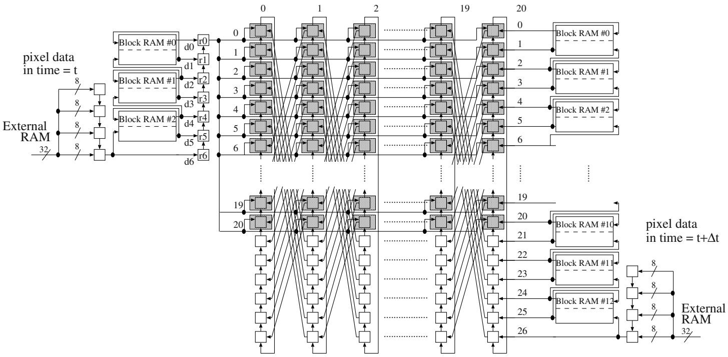


Fig. 4. An Array of SAD Units and a Register Array

and (3) six data (*d1-d6*) are also stored in the three Block RAMs at the same time. Then, data on the registers (*r0-r6*) are shifted 7 times, and broadcasted to all SAD units. With this implementation, pixel data are read out only once from the external memory bank.

3 Performance

This circuit was implemented on ADM-XRC-II by Alpha Data with one XC2V6000. Units for stereo vision was also implemented using almost same units as the optical flow. The circuit runs at 68 MHz, and 90% of slices in XC2V6000, 22 Block RAMs are used. This performance (30 images (640 × 480) per second) is fast enough to achieve real-time processing of the optical flow and the stereo vision.

4 Conclusions

In this paper, we described a compact system for real-time detection of moving objects. In our current implementation, we could find vectors for all pixels in a large size image (640 × 480) by comparing the window (7 × 7) centered at the pixel with its target area (27 × 27) in real-time (more than 30 images per second). The size of the target area is large enough to detect moving objects which are not moving extremely fast. We are now improving the system to work with an edge detection system to clearly distinguish borders of the moving objects.

References

1. Liu, H., Hong, T.H., Herman, M., Camus, T.A., Chellappa, R., “Accuracy vs. Efficiency Trade-Offs in Optical Flow Algorithms”, Computer Vision and Image Understanding, 72(3), 1998, pp. 271-286
2. P.C. Arribas, F.M.H. Macia, “FPGA Implementation of Camus Correlation Optical Flow Algorithm”, Vision Interface 2001
3. M. Fleury, A.F. Clark and A.C.Downton, “Evaluating optical-flow algorithms on a parallel machine”, Image and Vision Computing, 19(3), 2001, pp. 131-143.
4. Correia, M.V., Campilho, A.C., “Real-time implementation of an optical flow algorithm”, International Conference on Pattern Recognition 2002, pp. 247-250.

Real-Time Visual Motion Detection of Overtaking Cars for Driving Assistance Using FPGAs

S. Mota, E. Ros, J. Díaz, E.M. Ortigosa, R. Agis, and R. Carrillo

Department of Computer Architecture and Technology, University of Granada, Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain.
{smota, eros, jdiaz, eva, ragis, rcarrillo} @atc.ugr.es
http://atc.ugr.es

Abstract. Overtaking is one of the most dangerous operations in driving. The rear-view mirror is sometimes not consulted by the driver or is momentarily useless because of the blind spot. This paper describes a simple FPGA system based on motion detectors of the fly and rigid-body detection that is able to efficiently segment overtaking cars using a sparse map of features from the visual field of the rear-view mirror. FPGA implementation allows real-time image processing on an embedded system.

1 Introduction

Since 30 years ago machine vision systems have learned from biological systems that represent high efficient computing schemes. However, current vision models (based on the vertebrate visual system) are limited and require high computational resources.

Insects are a good example of a well known and simple visual motion detection model. We describe here the hardware implementation of an algorithm based on the motion detectors of the fly using a FPGA device. A FPGA device is a cheap option that allows: a real-time image processing, an easier change of the parameters to adapt the system to different conditions and a rapid prototyping.

The system described is applied to the overtaking problem, which is one of the most dangerous situations in driving because the rear-view mirror is sometimes not consulted by the driver or is momentarily useless because of the blind spot. The algorithm detects the moving vehicle behind and determines whether it is approaching in an overtaking trajectory or not.

2 Functional Description of the System

It is known that the detection and analysis of motion are achieved by neural operations in biological systems, starting with registration of local motion signals within restricted regions of the visual field, and continuing with the integration of those local motion features into global descriptions of the direction, speed and object motion.

This bottom-up strategy is adopted in the proposed system based on the processing stages [1]: **Edges extraction** (the edges allow to extracting scene structure); **Correla-**

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1158–1161, 2004.

tion process (the processing scheme is inspired on the motion-detection model of the fly proposed by Reichardt [2], where describes a multiplicative correlation detector); **Rigid body motion detection** (if we can detect a population of pixels in a window of the image which share the same velocity -rigid body-, they are likely to represent the overtaking car).

3 Hardware Implementation

The CCD camera provides 30 frames per second of 640 x 480 pixels and 256 gray levels. We use a RC-200 board from Celoxica [3] that includes a Xilinx Virtex-II FPGA (XC2V1000 device) [4] and 2 SRAM banks. The FPGA has 1 million system gates, distributed in 5120 slices, and 40 embedded memory blocks of a total of 720 Kbits.

Table 1. Hardware cost of the different stages of the described system. The global clock of the design is running at 31.5 MHz, although the tables include the maximum frequency allowed by each stage. The data of the table are provided by the ISE environment.

Pipeline stage	Equiv. gates	Number of Slices	% occupation	% on-chip memory	Image size	Max.Fclk (MHz)
Frame-Grabber	96,576	904	17	3	640 x 480	75.9
Edges	69,247	185	3	2	640 x 480	50.2
Correlation	47,322	2,296	44	0	640 x 480	45.7
Rigid Body	25,975	1,033	20	0	640 x 480	41.4
Total system	239,465	3,415	66	5	640 x 480	41.4

A frame-grabber collects the input image into a FIFO buffer, implemented on an embedded memory block. This FIFO structure is used to extract the spatial edges of the image by convolution with the Sobel 3x3 mask.

The Reichardt stage correlates a pixel with a population of stored pixels from the previous frame. The current velocity of an edge will be the maximum among the correlation values. To compute the maximum as fast as possible, we have used a micro-pipelined winner-takes-all scheme which compares by pairs, in successive clock cycles, first the correlation results, and then the winner results of the previous comparisons. Finally, the velocity estimation allows the global computation of one pixel per clock cycle, but with a latency that depends on the number of correlation values.

The rigid-body stage also follows this winner-takes-all structure to compute the maximum number of pixels that share speed.

The necessity of storing data in the external SRAM banks forces us to design a module that allows the writing and reading to/from the SRAM banks.

The pipeline structure consumes 1 cycle per stage due to the full potential parallelism and pipeline used; therefore the limitation of the system is in the external memory banks access. In spite of this, the system is able to process images of 640x480 pixels at the speed of 512 frames per second, when the global clock is running at 31.5 MHz.

Table 1 summarizes the main performance and hardware cost of the different parts of the system implemented. The hardware costs in table are rough estimations extracted from sub-designs compiled with the ISE environment. When the system is compiled as a whole many resources are shared.

The low number of equivalent gates enables the possibility of using other devices with less equivalent gates, computing higher resolution images with parallel processing units or employ them in other processing stages, for example in a tracking process that alerts the driver when an overtaking car is detected.

In the presented approach we do not use the multiplier blocks (Virtex-II has 40 multiplier blocks) this enables the whole Virtex family as possible target platform.

4 Conclusions

This contribution describes a motion processing embedded system to be used as blind spot monitor and driver assistance system to prevent possible distractions. It is designed to detect overtaking cars. The front-end of the system are Reichardt motion detectors. We define filters based on motion patterns of the image that seem to correspond to moving objects and help to segment the overtaking vehicle (if present). This filtering technique is a robust scheme because it is only based on a rigid body motion rule. The moving features are processed in a competitive manner, only patterns that activate a whole population of detectors with a similar velocity pass through this dynamic filter stage.

The hardware cost of the proposed system is low and allows the use of cheaper devices than other vision based algorithms. The described real-time computing scheme in embedded FPGA systems (portable) together with the very promising results obtained in the context of the posed problem (overtaking car detector) opens good application perspectives in diverse fields (robot vision, automobile industry, etc).

Acknowledgments. This work has been supported by the 5th EU research framework funds through the European Project ECOVISION (IST-2001-32114). We would like also to acknowledge Hella for providing us with the overtaking sequences.

References

1. Mota, S., Ros, E., Ortigosa, E.M., Pelayo, F. J.: Bio-Inspired motion detection for blind spot overtaking monitor. International Journal of Robotics and Automation (in press.).
2. Reichardt, W. Autocorrelation, a principle for the evaluation of sensory information by the central nervous system. In: W. A. Rosenblith (ed.): Sensory Communication. MIT Press, Cambridge, MA, (1961) 303-317
3. http://www.celoxica.com/products/boards/rc200.asp
4. http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Platform+FPGAs

Versatile Imaging Architecture Based on a System on Chip

Pierre Chalimbaud and François Berry

LASMEA, 24 avenue des Landais, 63177 Aubière, FRANCE

Abstract. In this paper, a novel architecture dedicated to image processing is presented. The most original aspect of the approach is the use of a System On Chip implemented in a FPGA. We show the main advantages of such a system based on a CMOS imaging device and a programmable chip. With its structure, the system proposes a high degree of versatility and allows the implementation of parallel image processing algorithms. As a result, a Window Of Interest (WOI) module is detailed.

Architecture overview. In this paper, an embedded imaging architecture based on a combination of CMOS imager and FPGA is proposed. Traditionally, the image is transduced by a sensor (camera), converted and stored (image grabber) and processed on a host computer[2]. In our approach, all early-vision[1] processing is performed in the sensor. This kind of sensor composed of a camera and an embedded processing unit, is currently called a smart sensor. The architecture presented in this paper can be considered as a development platform for a smart image sensor based on a System-On-Chip approach.

Based on the active vision concept, our approach consists in integrating the control of the imager in the perception loop, especially in the early vision processes. By integration of early processing and active vision mechanisms, close to the imager, a reactive sensor can be designed. The objective is to adapt sensor attitude to change in the environment and the current task to be performed. With such a smart sensor, it is possible to perform basic processing and selection of relevant features closed to the imager. This faculty is able to reduce sensor communication flow which is a significant problem of vision sensors [1]. But vision tasks are numerous and varied, and the choice of a versatile architecture based on a reprogramable chip becomes natural. In our case, the notion of SOC (System On Chip) describes the whole system.

It is well known that most vision applications are often focused on several small image areas and consequently acquisition of the whole image is not necessary. From this definition, it is evident that one of the main goals of an efficient vision sensor is to select windows of interest (WOI) in the image and concentrate processing resources on these. Indeed, the notion of local study is predominant. This notion is crucial for the choice of the imaging technology. CCD and CMOS are the two most common technologies used today in industrial digital cameras.

[1] The term "early vision" describes all processes except a high-level decision. It is different from binarization, convolution or other classical low-level processes

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1162–1164, 2004.

A particularity of CMOS imagers is to adopt a digital memory style readout, using row decoders and column amplifiers. Random access of pixel values becomes possible, allowing selective readout of windows of interest. In this way and in contrast to CCD imagers, it is possible to obtain a high speed imaging capability by addressing only a small region in the image. The main advantages of CMOS imagers are broad dynamic range, random access readout, easy integration, and low blooming.

Design adopted. The central idea in our sensor is that visual tasks can be broken down into a sequence of simpler subtasks. Consequently the principle consists in having a collection of routines that represent different kinds of basic image processing sub-functions. These can then be composed to subserve more elaborate goal-directed programs. In our approach, the sub-functions are implemented in a FPGA so that they can be changed easily. The global processing system is composed of SOPC by which an entire system of components is put on a single chip (FPGA).

The whole architecture is shown in Figure 1 and presents the different modules. The integration of the camera is done as shown in Figure 2.

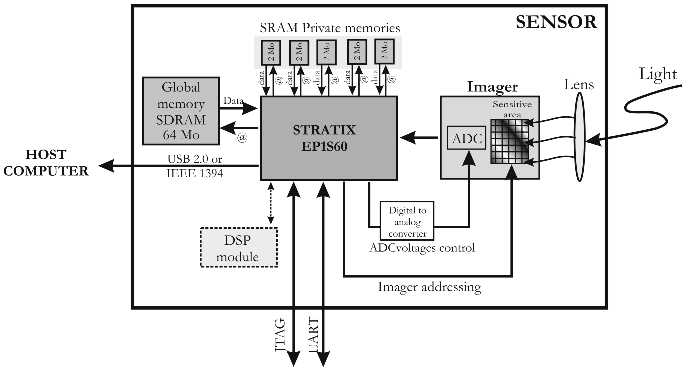


Fig. 1. Architecture of the sensor

The design objective presented in this paper was to create a flexible interface between the imaging device board and a host computer. This design exploits the advantages of the SOC cited above in order to allow software control of the acquisition chain. The design is based on a master entity which synchronizes and defines the control parameters of a set of modules. This set of modules acts on various points of the acquisition chain. The master entity is synthesized by a $NIOS^{©}$ soft core processor and its role is then easily defined by the software.

Application: Enhanced Windowing. This section presents the results on the WOI generation module. As explained below, this module generates the addresses of pixels in order to acquire only a predefined area. The advantage of

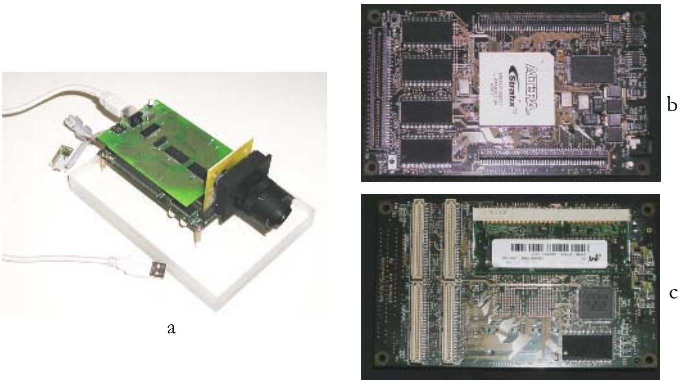
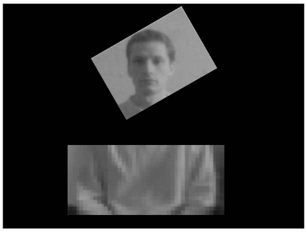


Fig. 2. a - Camera, b - top view of the main board, c - back view of the main board

enhanced windowing directly on a CMOS imager allows to set the acquisition times obtained for several acquisition windows (The acquisition time of a WOI 32×32 is 170 μs or 5800 fr/s). The main point of this technique is the sampling problem. In fact, the pixels of the CMOS imager do not have a 100% fill factor, so there is a blind zone between the sensitive areas. The consequence is that the distance between the sensitive areas varies as a function of the orientation of the window. One of the next tasks will be to interpolate the pixels in order to obtain a regular sampling step.

Full frame



Tilted and sub-sampled WOI

In conclusion, this paper proposes an alternative to the classical architecture with a highly versatile architecture dedicated to image processing.

References

1. Chalimbaud P. and Berry F. and Martinet P. The task ”template tracking” in a sensor dedicated to active vision. In *IEEE International Workshop on Computer Architecture for Machine Perception*, New Orleans, LA, USA, May 2003.
2. S. Fischer, N. Schibli, and F. Moscheni. Design and development of the smart machine vision sensor. In *SPIE EUROPTO Conference on Advanced Focal Plane Arrays and Electronic Camera*, volume 3410, Zurich, Switzerland, May 1998.

A Hardware Implementation of a Content Based Image Retrieval Algorithm

Constantinos Skarpathiotis and K.R. Dimond

Department of Electronics, University of Kent, Canterbury, Kent, CT2 7NT, England
{cs12,k.r.dimond}@kent.ac.uk

Abstract. The need for efficient content-based image retrieval has increased tremendously in many application areas such as biomedicine, military, commerce, education, and Web image classification and searching. We present a method where local and global features are extracted. As a global feature, we extract the colour histogram. As local features, we extract prominent regions from the image using a k-means variant and a labeling algorithm. For each region, colour and spatial locations are extracted. Because these algorithms are computationally intensive, a hardware implementation is presented that accelerates the processing of the images. The proposed design is well suited for implementation on an FPGA. The device can be used as an add-on to a Personal Computer (PC).

Keywords. Image Retrieval, FPGA, Component Labelling, Clustering

1 Introduction

As the memory of Personal Computers (PC) increased and high-bandwidth Internet connections emerged, it became easier the transfer of pictures and video sequences. New databases emerged which are called Multimedia Databases (MD). These databases except from text they can also include images and video sequences. It is clear that new tools are needed in order to search for pictures or video sequences on a multimedia database. The research area based on image indexing and retrieval is called Content Based Image Retrieval (CBIR). Our approach is based on region based image retrieval and the work presented in papers [1], [4] and [5]. The general method is to segment an image into distinct objects and then extract features for each object separately. The algorithms used for image indexing are computationally intensive and the databases usually include thousands of images. Indexing a database like that using a software program can be a lengthy procedure. The alternative is a hardware implementation. This paper describes the architecture of an indexing algorithm, which is well suited for implementation on an FPGA. The FPGA can be used as an add-on unit to PC.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1165–1167, 2004.

2 The Proposed Algorithm

The algorithm we propose extracts global and local features from images. The computed features can be used to search for similar images in an image database. As a global feature the colour histogram is computed. The local features represent the colours and location of regions after the application of a clustering algorithm. A block diagram of the proposed algorithm is presented in Fig. 1.

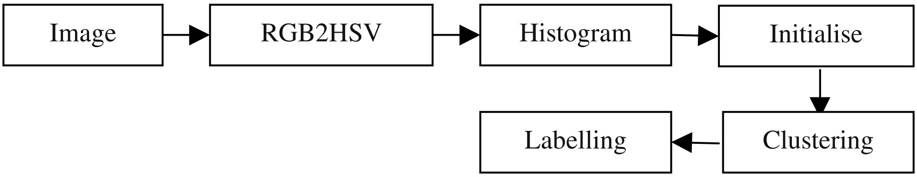


Fig. 1. Block diagram of the proposed algorithm

The first stage of the algorithm converts the RGB image into HSV colour space. From experiments, we concluded that a better clustering is achieved using the HSV colour space. The conversion of RGB images to HSV is computed using a Matlab function. Additionally, we cluster the colour space into 256 colours in order to compute the image histogram. For the colour clustering, we use 16 hues, 4 saturations and 4 values. Now that we have reduced the number of possible colours to 256, we compute the histogram. The histogram algorithm counts the number of times each colour was encountered in the image.

The next two stages present the Initialise and the Clustering algorithms. We introduce first the clustering algorithm and then we explain how it is initialised. The clustering algorithm is used to group pixels of similar colours. Consequently, important regions of the image will emerge. Forgy [2] proposed the following clustering algorithm:

1. Begin with a number of seed points.
2. Assign data units to the cluster with the nearest seed point. We continue this step until all data units have been assigned to clusters.
3. Compute the centroid of the clusters produced. The centroids are handled as new seed points
4. Alternate steps two and three until there is no change in centroids.

As data units, we use a feature vector, which consists of the colour components of each pixel. To compute the distance between a pixel and a centroid we use the Manhattan distance. Forgy's clustering algorithm has a major drawback. The number of clusters must be determined in advance. The solution we propose is to set a threshold on the histogram and accept as initial cluster centroids the colours with a count larger than a predefined threshold. This step is performed in the Initialise stage. When the clustering algorithm is finished, each pixel's colour information is replaced by the centroid of the nearest cluster. In the last step, we feed the clustered image to the connected component algorithm presented in [3]. The algorithm assigns a unique label to each connected component in the image while ensuring a different label for each distinct object. This algorithm is used for binary images but we altered the operation in order to apply for colour images.

3 Simulation Results

This section discusses the processing benefits in terms of speed when we use a hardware implementation. The algorithms already discussed were implemented using the Matlab software tool. We managed to operate the histogram and the Forgy's clustering algorithm at a clock speed of 35MHZ. For an image of 256*256 pixels, we achieve 530 passes per second while in Matlab we reached 33 passes per second. The labelling algorithm is using a 70MHZ clock. We calculated that for a 256*256 image we achieved 76 passes per second over the image. The Matlab implementation achieved 20 passes per second. We need to note for clustering and labelling algorithm that the number of passes needed to label or cluster an image differs from image to image.

4 Conclusions

In this paper, we presented a hardware implementation of an indexing system. The features we extracted were global and local. The colour histogram is the global feature we extracted and can be used in histogram queries. In addition, the location and colour of prominent regions were also extracted using a clustering and a labelling algorithm. The major advantage of our design is the speed up of processing compared to a software implementation.

References

1. M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, and B. Dom et al. Query by Image and Video Content: The QBIC System. IEEE Computer, 28(9), 1995.
2. E. Forgy**,** Cluster analysis of multivariate data: efficiency vs. interpretanility of classifications. Biometrics, 21, 1965
3. R,M. Haralick, Some neighbourhood operations, in real time/ parallel computing image analysis, plenum press, New York, 1981.
4. John R.Smith and Shih-Fu Chang, Tools and techniques for colour image retrieval, SPIE Proc. 2670, 1996.
5. James Z. Wang, Jia Li, Gio Wiederhold, ``SIMPLIcity: Semantics-sensitive Integrated Matching for Picture LIbraries," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 9, pp. 947-963, 2001.

Optimization Algorithms for Dynamic Reconfigurable Embedded Systems

Ali Ahmadinia

Department of Computer Science 12, University of Erlangen-Nuremberg, Germany

Reconfigurable hardware components such as FPGAs is used more and more in embedded systems, since such components offer a sufficient capacity for a complete SoC(System on a Chip) or even NoC(Network on a Chip). In order to use efficiently the dynamic reconfiguration possibility on such components, one needs a support in the form of operating systems to manage both software and reconfigurable hardware processes. For this support, suitable reconfigurable hardware model, and optimization methods are required. Our goal is the investigations of optimal strategies, methods, and architectures for controlling and use of the newest and future generations of reconfigurable hardware.

We target a Dynamic Reconfigurable System(DRS) which is made up of a soft core such as CPU(Central Processing Unit), and a Reconfigurable Processing Unit(RPU). In modelling of hardware resources, the characteristics like non-interruption of hardware tasks, reconfiguration overhead, communication model are considered. Task and resource managements on the DRS are on-line problems, that we investigate the optimal solutions for them.

Run-time space allocation, also known as temporal placement or on-line placement is a central part in reconfigurable computing system. In the on-line placement problems, for placing a new module, first we should identify the set of potential sites to place the new task. In most of the work, maximal empty rectangles will be stored for identifying the possible regions. But this method has a high complexity and modules should be placed always in the bottom left corner of an empty rectangle. Considering the fact that the set of empty rectangles grows much faster than the set of placed rectangles (tasks) leads us to manage the occupied space rather than the free space on the device. We identify the impossible regions, by computing the potential overlapping of the new module with other placed modules. In the second step in on-line placement, the best position to place the new module according to a set of given criteria should be selected. In contrast to existing methods, instead of using heuristic approaches such as best-fit or first-fit, we optimize the communication cost of the new module with the placed modules and input/output ports of the device. We have suggested two algorithms for this optimization, one in terms of Euclidean distance in the communication cost, and an optimal one by Manhattan distance.

To determine how the communications should be realized, optimal routing algorithms are needed. For on-line routing, there are two scenarios: packet routing and circuit routing. We are now implementing a packet routing approach on Xilinx FPGA for communication of placed modules, but our concentration is to develop optimal circuit-switching routing methods, which is more technology independent.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1168, 2004.

Low Power Reconfigurable Devices

Aman Gayasen

Dept. of Computer Science and Engineering
Pennsylvania State University

Advancements in VLSI technology have enabled the development of large reconfigurable platforms, which are being increasingly used in a wide variety of applications. Among the various reconfigurable devices currently in use, FPGAs have emerged as the most popular reconfigurable devices. While power optimization has been only of secondary importance in many FPGA applications, growing importance of leakage in FPGAs designed in 90nm and below makes it imperative to treat power optimization as a first class citizen.

My current research focuses on reducing the power consumption in FPGAs. In [1], we proposed a leakage-saving technique for FPGAs that involved dividing the FPGA fabric into small regions and switching on/off the power supply to each region using a sleep transistor in order to conserve leakage energy. Specifically, the regions not used by the placed design were supply-gated. It was observed that using small regions increases the area overhead of supply transistors, and hence, a new placement strategy was presented to maximize the number of regions that can be supply-gated while using large regions. This resulted in average leakage savings of 13% in Virtex-II FPGAs even with regions sizes as large as 128 slices, compared to 15% for region sizes of 4 slices.

In another technique, we propose a dual-V$_{DD}$ FPGA architecture, that uses two supply voltages to save power. Notice that normally a design consists of several non-critical paths that have additional timing slack as compared to the paths that determine the clock speed. Based on this observation, we propose two algorithms that assign lower supply voltages to the portions of the FPGA on to which the non-critical paths are mapped. Further, in a dual-V$_{DD}$ FPGA, a voltage level conversion is required if a low V$_{DD}$ block drives a high V$_{DD}$ block. Dedicated level converters, having area, delay, as well as power overheads, are required for this job. We experimented with two placements of level converters, one at the output pins of logic blocks (CLBs), and another at their input pins. Our experimental results show that reducing the supply voltage selectively to the non-critical paths helps to reduce dynamic power by 25% and leakage by 73% without affecting performance.

Modern high-end FPGAs have processors embedded in them (e.g. PowerPCs in Xilinx Virtex-II pro). This, coupled with the fact that these FPGAs can be dynamically reconfigured, calls for a comprehensive study of power issues in such FPGAs. Analyzing these issues forms the next step of my research. At a later stage, I also wish to work on novel non-silicon reconfigurable architectures.

References

1. A. Gayasen, Y. Tsai, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, and T. Tuan. "Reducing Leakage Energy in FPGAs using Region-Constrained Placement", *FPGA*, 2004.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1169, 2004.

Code Re-ordering for a Class of Reconfigurable Microprocessors

Brian F. Veale[1], John K. Antonio[1], and Monte P. Tull[2]

[1] School of Computer Science
[2] School of Electrical and Computer Engineering
University of Oklahoma, Norman, Oklahoma, USA 73019
{veale, antonio, tull}@ou.edu

A class of reconfigurable processors is introduced in which support for an instruction set is distributed among a collection of pre-defined configurations. For this new class of reconfigurable processors, there is assumed to be a pre-defined collection of configurations in which each configuration supports a subset of the overall instruction set. The union of all subsets of instructions, associated with the configurations, defines the instruction set supported by the reconfigurable processor. An objective for this class of reconfigurable processors is the support of popular commercial instruction set architectures with less hardware than required using existing static (i.e., non-reconfigurable) processors.

The basic problem considered is the following: Given a collection of configurations for a reconfigurable processor and given an executable program based on the entire instruction set, devise an algorithm to re-order the given program code so as to minimize the number of configuration switches required to execute the program on the reconfigurable processor. Our solution approach is based on a block-based analysis of the program. A greedy algorithm that works on a precedence DAG of a block of code is used to schedule the instructions to minimize the total number of configuration switches for each block.

An experimental study has been conducted based on a proposed partitioning of the PowerPC™ instruction set into two mutually exclusive subsets, one consisting of all floating-point instructions and the other consisting of all other instructions. The greedy algorithm is used to re-order machine code to minimize the number of reconfigurations required. The greedy algorithm reduces the number of configuration switches by around 50% in some cases. Additionally, the results of the experimental study highlight that a low percentage of blocks, about 20%, require configuration switching for the partitions assumed. In the study, static analysis of machine code was performed, i.e., the number of times a given block is performed during an actual program execution is not considered. The processor model is assumed to consist of one execution unit that can be reconfigured to implement different partitions of the instruction set.

Future investigations will involve the tracing of program executions so that the number of times each block is actually executed can be measured. Reduction in the number of reconfigurations for blocks that are executed multiple times provides greater improvement in overall performance than gains reported here. Another area of work includes investigating the application of clustering techniques to define near-optimal instruction partitions.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1170, 2004.

Design Space Exploration for Distributed Hardware Reconfigurable Systems[*]

Christian Haubelt

Hardware-Software-Co-Design
University of Erlangen-Nuremberg, D-91058 Erlangen, Germany

A recent trend towards networked and hardware reconfigurable systems can be identified in several areas like automotive, ambient intelligence, and many more. On the other hand, there is a lack of sophisticated design automation tools which support designers during the design process. With the increasing complexity of such modern embedded systems, an urgent need for design tools at higher levels of abstraction can be seen. The most challenging tasks in system-level design for networked reconfigurable systems is to guarantee the feasibility of solutions while exploring giant design spaces. To overcome these problems, a hierarchical graph-based model for networked hardware reconfigurable systems is introduced. The hierarchical system model consists of three parts: (i) A hierarchical process graph modeling the desired functionality of the system. Subgraphs associated with processes are used as alternative refinements for these processes. (ii) A hierarchical architecture graph representing the set of allocatable platforms. Subgraphs associated with hardware components are meant to be alternative configurations of hardware reconfigurable components. (iii) Hierarchical mapping edges relate processes with components of the architecture graph in a sense that these processes may be executed on the associated hardware components. The novelty of this model lies in the ability to support (i) Platform-based design: By optimizing a platform not only for a single static process graph but for a set of different refinements. (ii) Reconfigurable computing systems: The hierarchical architecture graph allows to exchange hardware functionality at runtime. (iii) Modeling of IP cores: Hierarchical mapping edges model partial implementations or subsystems with unknown properties. In contrast to existing solutions which mostly assume implicit communication between configurations and where configurations are constructed by structural temporal partitioning, our model is a real system-level model based on functional partitioning, since it supports explicit communication which let us detect hidden deadlocks in an implementation and configurations consist of system-level components like CPU, IP cores, buses, or even again FPGAs. In this Ph.D.-program, different methods for accelerating the task of design space exploration by exploiting the hierarchical structure of the underlying model have been proposed. All these approaches are based on Evolutionary Optimization. The most outstanding idea is called Pareto-Front Arithmetics where subsystems are optimized independently of each other. Later, the optimization results are combined regarding the hierarchical problem structure. Moreover, new formal methods based on SAT-techniques for testing the feasibility of an implementation have been developed. Furthermore, these SAT-techniques can be extended towards an analysis strategy to determine the degree of fault tolerance of a networked hardware reconfigurable system. The feasibility of this work, was shown by developing a networked hardware reconfigurable system, called ReCoNet.

[*] This work was supported in part by the German Science Foundation (DFG), SPP 1040.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1171, 2004.

TPR: Three-D Place and Route for FPGAs

Cristinel Ababei (Advisor: Professor Kia Bazargan)

Department of Electrical and Computer Engineering, University of Minnesota,
Minneapolis MN 55455, USA, ababei@ece.umn.edu

Due to technology scaling global wires dominate the delay and power budgets, and signal integrity, IR-drops, and process variations pose new design problems. Shrinking time-to-market windows and ever-increasing mask costs have reduced profits alarmingly. In response to the first category of problems, 3D integration can significantly reduce wire-lengths, boost yield, and can particularly be useful for FPGA fabrics because it can address problems related to routing congestion, limited I/O connections, and long wire delays. Practical application of 3D integrated circuits yet needs to gain momentum, partly due to a lack of efficient 3D CAD tools. We propose a new efficient timing-driven partitioning-based placement and routing tool for 3D FPGA integration [1]. The circuit is first divided into layers with limited number of inter-layer vias, and then placement is performed on individual layers, while minimizing the delay of critical paths. Finally, the circuit is routed using our 3D detailed routing algorithm. We show that 3D integration results in smaller circuit delays, provided that multi-via lengths are employed between layers or fully buffered routing resources are used. Simulations show on average a total decrease of 25% in wire-length and 35% in delay respectively, over traditional 2D chips, when 8 layers are used in 3D integration.

One can cope with the shorter time-to-market problem in two ways. First, one can develop more efficient design automation tools. We integrated an efficient terminal alignment heuristic for delay minimization into a new partitioning-based placement algorithm, which can achieve comparable circuit delays to those obtained with VPR at four times shorter run-times [3]. Second, one can adopt new design methodologies such as platform-based design, reconfigurable computing [2] or improve early design-metrics estimation in order for front-end design stages to be better aware of the later design optimization decisions. Optimization at lower levels of abstraction should be done in a constrained fashion such that early predictions shall actually turn to be accurate. In this way a constructive constrained optimization design methodology can be developed, which will lead to better predictability (hence, smaller number of design cycles).

References

1. Ababei, C., Bazargan, K.: Exploring Potential Benefits of 3D FPGA Integration. Submitted. (2004)
2. Ababei, C., Bazargan, K.: Non-Contiguous Linear Placement for Reconfigurable Fabrics. Reconfigurable Architectures Workshop (RAW). (2004)
3. Maidee, P., Ababei, C., Bazargan, K.: Fast Timing-driven Partitioning-based Placement for Island Style FPGAs. Proc. ACM/IEEE Design Automation Conference (DAC). (2003) 598–603

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1172, 2004.

Implementing Graphics Shaders Using FPGAs

David B. Thomas and Wayne Luk

Department of Computing, Imperial College, London SW7 2BZ, England

There are many similarities between modern 3D graphics chips and FPGAs: a shader based graphic chip can be viewed as a highly domain specific and very coarse-grained reconfigurable logic device. Our goal is to see how a fine-grained FPGA can be used to implement the pixel and vertex shader stages of a graphics pipeline, and how well the two methods compare in terms of speed, power, flexibility, and usability, initially by implementing Direct-X 9 pixel shader programs using Xilinx Virtex-II FPGAs.

Modern graphics cards use a customisable graphics pipeline split into two stages, which replace the classic fixed function graphics pipeline stages. The vertex shader is responsible for transforming scene geometry, such as manipulating vertex locations and perturbing vertex normals, while the pixel shader is responsible for performing per-pixel operations, such as specular lighting effects and environment mapping. Both shaders can be programmed by applications to achieve different effects using a simple assembler language, and different shader programs can be applied on a per-polygon basis.

As graphics chips have become more powerful, the capabilities and resources available to shader program writers have also increased, with facilities such as dynamic flow-control, floating-point registers, and more freedom in the ordering of instructions. This increase in flexibility and scope requires large amounts of logic to reach acceptable fill-rates, and the need to support the most complex possible shader programs means that for most applications there will be large amounts of logic left idle. Implementing shaders using FPGAs should increase logic utilisation while providing the flexibility for complex CGI-style shader trees.

Optimisations performed while compiling shaders to an FPGA design include: Compiling multiple shader programs into a single design, by transforming and interleaving instructions to maximise shareable logic; Altering the precision of fractional and fixed-point calculations to reduce logic requirements, while maintaining the mandated precision requirements; Reducing the texture memory bottleneck by using information gleaned from the shader to reduce the precision of and rearrange storage; Using block-memories to create caches with texture affinity and other program specific characteristics.

Experiments using our automatic compiler have so far achieved the implementation of six commonly used shaders on a single Xilinx Virtex-II 1000 device with a maximum clock rate of 90MHz, providing a fill-rate of 90MPel/s. While this can't yet compete with commercial graphics cards, which have a clock speed 3-4 times higher and a fill rate 10-20 times higher, the price of FPGAs is going down, whilst that of graphics chips is gradually increasing. Future work includes rapid generation of designs using JBits, run-time reconfiguration to alter shaders present on a device, and the implementation of vertex shaders.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1173, 2004.

Preemptive Hardware Task Management

Dirk Koch

Department of Computer Science 12, University of Erlangen-Nuremberg, Germany

In order to achieve real estate reconfigurable computing in prospectively embedded systems it is necessary to support the preemption of hardware tasks. This means we want to freeze a module, capture the module's state and restart the module at a later time. In the software world, the state is basically the context of the CPU registers. In hardware the state data is represented by the state register values and is typically an order of magnitude larger. In addition, this data is spread over the entire module and is not accessible by simple PUSH/POP mechanisms. Hardware task preemption plays a key role in building fault tolerant systems. It can be used for error detection (e. g. register out of range detection) as well as for the redistrubution of workload in the case of a failure. Furthermore, task preemption allows virtualizing hardware by time multiplexing. This is a challenging topic especially in real time systems where we have to prove that the exchange of a module is achieved below a certain time boundary.

There are two basic opportunities to capture the state of a running hardware task that is applicable with today's FPGAs. The first methodology is to read back the configuration bitstream. This is quite slow as the whole bitstream and not only values representing the state is read back. As the bitstream size can be in the megabit range, there is need to extract just the state information from the bitstream. The second methodology to capture the state of a running hardware module is to include a register scanpath into the module. Here, we started to examine the deterioration of the performance (resources and speed) stemmed from this approach. However, a scanpath allows fastest context switching and is applicable to reconfigurable architectures without dedicated readback modes. In addition, we want to use scanpaths for migrating modules containing a state from and to different reconfigurable architectures or even for hardware software migration.

In our system model we allow multi cycle IO transfers (DRAM access) and multiple clock domains leading to a non deterministic behaviour of the freeze and restart process. Furthermore, we consider combinatorial feedback loops and on-chip memory blocks by automatically including wrapper blocks into the module.

To exploit the advantages of both methodologies we aim at developing a reconfigurable architecture that takes the hardware preemption into account. We have currently a prototype FPGA architecture running on top of a host FPGA allowing fast context capturing. This is achieved by arranging the reconfigurable elements and therefore the bitstream in such a way that the state data is stored in the beginning of the bitstream. All other configuration information (e. g. routing data) is put at the end and is not further considered. In our approach, the reading back of the bitstream containing the state of an interrupted module can be done simultaneously with the reconfiguration of the new module.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1174, 2004.

Automated Speculation and Parallelism in High Performance Network Applications

Graham Schelle and Dirk Grunwald

University of Colorado at Boulder, Boulder, CO 80309, USA

Speculation and parallel processing can provide performance gains in many diverse applications. Compilers, grid computing, DSP, and bio-informatics are a representation of such areas where these concepts are utilized. In the field of network routers, packet processing can also use such speedups. As the line rate of packets increases with every new standard (Infiniband, 10-gigabit Ethernet), these speedups will become paramount for routers asked to do complicated tasks while still maintaining line speeds.

In order to facilitate a design platform for rapid prototyping of these high-speed router designs, we present CUSP (Click Utilizing Speculation and Parallelism). Click is a software modular router design framework that is similar to CUSP, but specifically built for a Linux platform and software routers. CUSP, while also having a modular design of reusable components, additionally provides automated speculation and parallelism. An accompanying scripting language, *CUSPED*, allows quick creation of these routers from existing components.

The FPGA platform provides the base to create such a system. Reprogrammable hardware allows for multiple data paths to execute simultaneously. In addition to that parallelism, speculation can occur on those data paths at no expense to other paths. This independence assumption across data paths is very applicable and feasible across networking applications. For example, a packet is classified when arriving at a network router. This classification leads to another level of processing depending on the type of packet. In CUSP, the packet is speculated to be EVERY type of packet the application can process, and speculatively executed in parallel as those types.

This speculation requires special attention to components that can change the state of the networking application. Specifically, memory accesses that are speculatively executed may have to be rolled back. The notion of predicate registers in compilers can play a role in this issue. By predicating writes and using a commit signal to validate those writes, speculation can occur on state changes as well. To demonstrate this speculative memory, a FIFO and Table memory constructs have been created.

We have designed an example network application that conforms to the ideas presented above. Using a Xilinx Virtex-II Pro FPGA, this application accepts IPv4, ARP, and ICMP packets from a gigabit link. Speculation and parallelism occur across all these packet types involving multiple memory accesses. From this working example, we have shown that speculation and parallelism can be automated through a modular design.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1175, 2004.

Automated Mapping of Coarse-Grain Pipelined Applications to FPGA Systems

Heidi E. Ziegler

USC Department of Computer Engineering Doctoral Thesis

Abstract. Configurable systems offer a unique opportunity to define application-specific architectures. These architectures offer performance advantages, where the use of customized pipelines exploits the inherent parallelism of the application. In this research, we describe a set of program analyses and an implementation that automatically map a sequential and un-annotated C program into a pipelined implementation targeted to an FPGA with multiple external memories. This research describes an automated approach to hardware design space exploration, through a collaboration between parallelizing compiler technology and high-level synthesis tools. In previous work, we described a compiler algorithm that optimizes individual loop nests, expressed in C, to derive an efficient FPGA implementation. In this research, we describe a global optimization strategy that maps multiple loop nests to a coarse-grain pipelined FPGA implementation.

We focus on the space-time tradeoffs associated with differing amounts of parallelism, communication granularities and custom data layouts. Highly optimized designs may be too large to fit within FPGA resource constraints, so we describe heuristics for reducing area requirements while minimizing the impact on global performance. We present a design space exploration algorithm, which demonstrates the potential of this approach, for automatically deriving pipelined designs from high-level sequential specifications.

The configurability of FPGA hardware and the advent of multi-FPGA platforms leads to new decision procedures for applying existing transformations. In this research, we investigate how techniques, borrowed and adapted from existing parallelizing compiler technology, can be combined with commercial synthesis tools, to automatically derive realizable and efficient designs on multiple FPGA-based architectures. In particular, the contributions are as follows:

– Communication Analysis and Pipelining. We define a set of compiler analyses and transformations required to automatically design the communication for application specific pipelines for FPGA-based architectures. We determine the best communication granularity, the corresponding communication placement points within the code, and the exact data that must be communicated between pipeline stages.
– Partition and Custom Data Layout. Our compiler algorithm finds a coarse-grain computation and data partition, along with a custom data layout. To combat the large search space, we employ several heuristics.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1176–1177, 2004.

- Implementation and Evaluation. We implement our analyses and present experimental results for a set of image-processing kernels.

With the growing number of available transistors on a single die, we anticipate the emergence of multiprocessor systems-on-a-chip and reconfigurable computing architectures with the ability to incorporate (through soft-cores) various coarse-grain computing elements such as microprocessor cores, and application specific engines (ASEs). Enabling pipelined execution, communication across computing cores, task level parallelism, and data distribution across banked memories will become increasingly important issues. Our analyses will allow the automated application mapping for these emerging infrastructures.

A Specific Scheduling Flow for Dynamically Reconfigurable Hardware

Javier Resano

Universidad Complutense de Madrid, Spain, javier1@dacya.ucm.es

Dynamically Reconfigurable Hardware (DRHW) presents the ideal features to cope with the highly dynamic and non-deterministic behaviour of current multimedia applications (such as digital video and 3D games) since it provides both high performance and run-time flexibility. However, in order to take advantage of the DRHW features dynamic task allocation support and scheduling support are needed. Working together, the Interconnection Network (ICN) model for DRHW [1] and the hybrid run-time/design-time scheduling Task Concurrency Management (TCM) scheduling methodology [2] provide the desirable support to use DRHW in embedded systems. First, the ICN model provides support not only for task allocation, but also for inter-task communication, and operating system primitives. Second, TCM provides run-time scheduling support and generates only a small run-time penalty due to its execution because most of the exploration and computation is done at design time.

However, the run-time flexibility of DRHW comes at the price of a very large reconfiguration overhead. For instance, reconfiguring one tenth of a Virtex XC2V6000 requires at least 4 ms, which is often unaffordable. In order to reduce this large reconfiguration overhead, specific support is needed for the DRHW resources during the scheduling flow. To provide this support I have developed two different techniques, namely the prefetch-scheduling technique and the replacement technique, and integrated them in the TCM scheduling flow [3]. First, the prefetch-scheduling technique receives as input a set of tasks that must be loaded and attempts to hide their loading latency by loading them in advance. The key idea of this technique is that if a task is loaded before the moment of time when it is going to be executed no time-overhead is generated by this load. Second, the replacement technique attempts to maximise the reuse of tasks applying a replacement policy that takes into account which tasks are more likely to be reused in the future. Thus, a task can be loaded once and executed multiple times as long as is not replaced by another task. In this context, the tasks loaded in the DRHW are conceptually similar to the memory pages in physical/virtual memories. In addition, this technique takes into account how critical the execution of a task is and it assigns more priority to those tasks that are more critical for the system performance. Applying these techniques to an actual set of multimedia applications they reduce the initial reconfiguration overhead by a 93% while generating a very small run-time penalty due to its execution.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1178–1179, 2004.

References

[1] T. Marescaux et al., "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", Proc. of FPL'02, pp. 795-805, 2002.

[2] P. Yang et al., "Energy-Aware Runtime Scheduling for Embedded-Multiprocessors SOCs", IEEE Journal on Design&Test of Computers, pp. 46-58, 2001.

[3] J. Resano et al., "Run-Time Minimization of Reconfiguration Overhead in Dynamically Reconfigurable Systems", Proc. of FPL´03, pp. 585-594, 2003.

[4] J. Resano et al, "A hybrid design-time/run-time scheduling flow to minimise the reconfiguration overhead of FPGAs". Elsevier Journal on Microprocessors and Microsystems, Vol. 28/5-6, pp. 291-301. 2004.

Design and Evaluation of an FPGA Architecture for Software Protection

Joseph Zambreno

Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA
zambro1@ece.northwestern.edu

Research into defense mechanisms for digital information has intensified in recent years, as malicious attacks on software systems have become a rapidly growing burden. The growing area of *software protection* attempts to address the problems of code understanding and code tampering along with related problems such as authorization. Our work focuses on the design and evaluation of an architecture for software protection that utilizes FPGAs as a run-time integrity enforcement engine.

Recent approaches to software protection tend to lie at two extremes of the security-performance spectrum. At one end are highly secure hardware approaches that require a substantial buy-in from hardware manufacturers [1]. The other end provides security by either mangling the code to make it less understandable or by burying checksums in unlikely places [2]. These extremes invite an approach that allows system designers to position themselves where they choose on the security-performance spectrum.

Our proposed method works as follows. The processor is supplemented with an FPGA-based secure hardware component that is capable of fast decryption and can also recognize strings of keys hidden in plaintext instructions. The FPGA is situated between the highest level of on-chip cache and main memory in order to directly handle the instruction cache miss requests. These instructions would then be translated and verified in some fashion before the FPGA satisfies the cache request. An advantage to this approach is that the compiler's knowledge of program structure allows for tuning of the security and performance of individual applications. Also, the use of FPGAs minimizes additional hardware design and is applicable to a large number of commercial processor platforms. Our initial results [3] demonstrate that the average performance penalty for this approach is not too severe for most applications.

References

1. D. Lie et al. Architectural support for copy and tamper resistant software. In *Proc. of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 168–177, 2000.
2. C. Collberg and C. Thomborson. Watermarking, tamper-proofing, obfuscation: tools for software protection. In *IEEE Transactions on Software Engineering*, pp. 735–746, 2002.
3. J. Zambreno et al. Flexible software protection using hardware/software codesign techniques. In *Proc. of Design, Automation, and Test in Europe*, pp. 636–641, 2004.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1180, 2004.

Scalable Defect Tolerance Beyond the SIA Roadmap

Mahim Mishra

Carnegie Mellon University, Pittsburgh, USA

As feature sizes approach the single-digit nanometer domain, manufacturing defects will become much more commonplace. Each computing fabric will have multiple defects, and physical and economic limits will make it impossible to eliminate them all. This will be true whether manufacturing is done using future-generation CMOS processes, or technologies such as Chemically Assembled Electronic Nanotechnology (CAEN). We will therefore have to find a way to use these fabrics inspite of the defects.

We can compute with defective fabrics by making them reconfigurable, and not use defective resources when mapping circuits onto the fabric. This requires two things: a scalable testing methodology that can locate all defects in the fabric, and layout tools that can place and route circuits onto the fabric while avoiding the defective parts.

Scalable testing methods: Current testing algorithms for reconfigurable fabrics such as FPGAs can deal with only a small number of defects. In particular, they cannot scale to situations where as many as 10% of the individual switches, configurable logic gates or wires may be defective, as is predicted for a number of proposed future technologies. We developed a testing algorithm which gave good results in simulations where the fabric had 10% or more defective components [2,1]. Our method implements LFSR-based test circuits on the reconfigurable fabric and analyzes test circuit outputs by a variety of methods to identify the correctly functioning components on the fabric. In simulations, our algorithm was able to identify upto 80% of the correctly functioning components when the fabric had upto 10% defects. We are now implementing these algorithms to test Xilinx VirtexII-Pro FPGAs with defective logic cells.

Layout algorithms: Once defect locations have been determined, layout algorithms should map circuit configurations around these defective components. This poses a new challenge for these algorithms: not only does this require *per-design* layout effort, but also *per-fabric* effort, since each fabric will have a unique set of defects. Also, if such fabrics are to be used for general purpose computation rather than only as ASIC replacements, the place-and-route steps will need to be very fast and efficient. We are currently exploring ways to partition layout algorithms into two parts, one to be performed per-design, and a second part to be performed per-fabric. The per-design component of the layout will be performed once by the application developer, who will ship out the resulting *soft* configuration to the users. Each user will run a quick final layout step, which will take into account each fabric's unique defect map to generate a *hard* configuration that can be loaded onto the reconfigurable fabric.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1181–1182, 2004.

References

1. Mahim Mishra and Seth C. Goldstein. Defect tolerance after the roadmap. In *Proceedings of the 10th International Test Synthesis Workshop (ITSW)*, Santa Barbara, CA, March 30–April 2 2003.
2. Mahim Mishra and Seth C. Goldstein. Defect tolerance at the end of the roadmap. In *Proceedings of the International Test Conference (ITC), 2003*, Charlotte, NC, Sep 30 – Oct 2 2003.

Run-Time Reconfiguration Management for Adaptive High-Performance Computing Systems

Mohamed Taher[1] and Tarek El-Ghazawi[2]

[1]Doctoral Candidate Student, [2] Thesis Advisor
The George Washington University, Washington DC, USA
{mtaher, tarek}@gwu.edu

Abstract. Adaptive Computing Systems (ACS) built from configurable logic processors such as FPGAs offer a very high performance in implementing a wide range of applications. However, their programming model is mostly a hardware paradigm, instead of software paradigm, which makes them difficult to program by application scientists. They also suffer from portability and scalability problems. Furthermore, the performance of these run-time reconfigurable systems is limited by the cost of reconfiguration overhead.

One of the major limitations of ACS is that some large applications require more hardware resources than are available, and the design cannot fit into the available FPGA chips. One solution to this problem is run-time reconfiguration (RTR). Run-time reconfiguration allows modular large applications to be implemented by reusing the same configurable processor. These configurations are uploaded onto the reconfigurable hardware as they are needed to implement the application. As configuration time could be significant, eliminating or reducing this configuration time overhead becomes a very critical issue for reconfigurable systems.

This thesis uses the processing locality characteristics to discover, at run time, the workload processing needs and reconfigure the hardware accordingly, thus providing a cache-memory-like resource management scheme. The cache management techniques are used to determine when configurations should be loaded and unloaded in order to exploit temporal and spatial processing locality in order to minimize the overall cost of reconfiguration and, thus, the overall execution time. To this end, we propose to integrate data mining techniques, such as Association Rule Mining (ARM) and clustering, to derive meaningful rules that can be useful in configuration-cache management, by configuring related functions together. These rules are used to determine the correlation between the reconfigurable functions, in order to know in advance which functions are tightly correlated to the current executing reconfigurable function, and thus pre-fetch the configuration of these functions immediately before they are needed.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1183, 2004.

Optimized Field Programmable Gate Array Based Function Evaluation

Nalin Sidahao
(Supervisors: George A. Constantinides, Peter Y.K. Cheung)

Department of Electrical & Electronic Engineering, Imperial College, London, UK,
nalin.sidahao@imperial.ac.uk

Abstract. A new family of architectures for multi-cycle area-efficient evaluation of elementary and composite functions is presented, and the design tradeoffs for implementation on FPGAs are explored. The method based on minimax polynomial approximation is exemplified with two common functions, sine and power-of-2. To test the performance of each design, we compare the proposed architecture to an established table-based method for several different input word-lengths and output precision requirements. FPGA-based results are presented, illustrating both the technology-independent and the technology-specific attributes of the tradeoff of area and speed between the proposed techniques.

We continue to work on a class of problem relating to the multiplication of a single number by several coefficients that, while not constant, are drawn from a finite set of constants that change with time. Such a situation arises commonly in synthesis due to resource sharing, for example in a folded implementation of a polynomial evaluation using Estrin's method and a FIR filter. To minimize the number of operations, we present the formulation as a form of common sub-expression elimination. The proposed scheme avoids the implementation of full multiplication. In addition, an efficient implemenation is presented targeting the Xilinx Virtex / Virtex-II family of FPGAs. We also introduce a novel use of Integer Linear Programming for finding solutions to the minimum-cost of such a multiplication problem. Using our formulation area saving of up to 25% has been achieved even for small benchmark problems.

While the work so far has been focused on area optimisation, future work will be extended to trading off speed and characteristics of the polynomial approximation such as order and precision used. Increasing the order of the polynomial and the word-lengths in the coefficients and the intermediate signals will increase accuracy at the expense of longer delay and lower throughput. Word-length optimization techniques will be explored to alleviate the speed penalty. Since the overall precision is limited by the polynomial approximation itself, increasing precision in the coefficients or signal representation could be superfluous and could incur unnecessary delays. Canonical Signed-Digital (CSD) number system will also be explored in order to achieve higher speed in the implementation of the multipliers and adders.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1184, 2004.

MemMap-pd: Performance Driven Technology Mapping Algorithm for FPGAs with Embedded Memory Blocks

R. Manimegalai, A. ManojKumar, B. Jayaram, and V. Kamakoti

Department of Computer Science and Engineering, IIT Madras, Chennai, India.

Abstract. Modern day Field Programmable Gate Arrays (FPGA) include in addition to Look-up Tables, reasonably big configurable Embedded Memory Blocks (EMB) to cater to the on-chip memory requirements of systems/applications mapped on them. While mapping applications on to such FPGAs, some of the EMBs may be left unused. This paper presents a methodology to utilize such unused EMBs as large look-up tables to map multi-output combinational sub-circuits of the application, with depth minimization as the main objective along with area minimization in terms of the number of LUTs used. This paper presents a new algorithm for technology mapping onto heterogeneous architectures containing LUTs and embedded memory blocks. For the first time, the concept of reconvergence is used in the field of FPGA mapping and is shown to be effective. The algorithm consists of four main stages, namely, Pre-Processing, Reconvergence Analysis, Memory Mapping and LUT Mapping. Experimental results show that our proposed methodology, when employed on popular benchmark circuits, leads to upto 14% reduction in depth compared with the DAGMap, along with comparable reduction in area.

Pre-Processing: In the first stage of the algorithm, the given circuit is converted into an equivalent two-input network. It has been shown that this conversion leads to better mapping of the circuit into LUTs by minimizing the overall depth of the decomposed circuit.

Reconvergence Analysis: In this stage, the circuit obtained from the preprocessing stage is analyzed for reconvergence and overlapping reconvergent regions are identified for mapping into embedded memories.

Memory Mapping: We use a 2-phase heuristic for selecting appropriate regions for memory mapping. In the first phase, the overlapping reconvergent regions that can be mapped to the memory blocks are expanded till they just satisfy the pin constraint imposed by the memory arrays. In the next phase, the best among the expanded regions are selected based on the potential depth reduction obtained by mapping the region onto embedded memory blocks.

LUT Mapping: This is the final phase of the algorithm in which the residual circuit left after mapping onto memory blocks is mapped into LUTs. The DAG-Map algorithm is used to implement this mapping.

Keywords: Embedded Memory Blocks, Reconvergence, Technology Mapping, Memory Mapping,Primary Stem Region, Dag Level, Slack.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1185, 2004.

A System on Chip Design Framework for Prime Number Validation Using Reconfigurable Hardware

Ray C.C. Cheung

Department of Computing, Imperial College London, United Kingdom

This paper presents a System on Chip (SoC) design framework for prime number validation which targets reconfigurable hardware. The primality test is crucial for most security systems using public-key schemes. It has been recognised that strong prime number generation is important, and prime validation is an intrinsic part of the generation. Our main contributions include: (1) A design method for mapping the Rabin-Miller Pseudoprime Test into hardware. (2) Parallel designs for Montgomery modular arithmetic operations. (3) A design generator for producing hardware prime number validators based on user-defined parameters. (4) An implementation of the proposed architectures in reconfigurable devices, with an evaluation of its effectiveness compared with other methods. (5) A scalable framework for parallelizing prime validations in reconfigurable hardware.

We define validation as the process of prime testing on a given number. This work has two parts: (I) a design flow from user-defined parameters to a synthesizeable core which includes contributions (1) to (4), and (II) a rapid-prototyping platform for integrating user cores and on-chip processor into an SoC design which includes contribution (5). This framework can easily be extended for developing embedded systems and cryptographic applications.

The Rabin-Miller test has been mapped into hardware. It makes use of efficient modular multipliers for computing Montgomery modular exponentiation to further speed up the validation and to reduce the hardware cost. The parallelism of this design has been explored for very large prime numbers. For instance, two parallel modular multipliers are used in Montgomery modular exponentiation, and the multiplier itself is optimised for parallel execution. A design generator has been developed to generate a variety of Montgomery modular multipliers, with different trade-offs in size and performance based on user-defined parameters. For instance, the number of small prime numbers used in the Rabin-Miller test determines the accuracy and performance of the system, and the user-specified bit-width determines the complexity of the modular operations in the hardware. We systematically implement the design for different bit-widths on reconfigurable devices in Celoxica RC200 and RC2000 platforms. The generated Handel-C designs are synthesized using Celoxica tools, and the FPGA is then configured as a prime number validator. Our work demonstrates the flexibility and trade-offs in using reconfigurable platforms. It shows that, for 512-bit prime validation, the design takes 14,176 slices and has a critical path delay of 81ns. A 1024-bit primality test can be completed in less than a second, which demonstrates that our prototype reconfigurable architecture can run more than

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 1186–1187, 2004.

15 times faster than a customised arithmetic crypto-processor in a smart card system.

We also investigate a general design framework for prime number validation that makes use of an embedded microprocessor, a fast Processor Local Bus (PLB) and programmable user-logic in reconfigurable hardware. In this framework, a divide and conquer technique is applied to prime number generation. The circuit in part (I) is used to validate one long prime number. The generated designs are first synthesized into VHDL and connected to the PLB bus through the predefined bus interface as programmable PLB slave bus modules. An embedded microprocessor such as the PowerPC is used to generate high quality random numbers and to interface between user and on-chip validators. The PLB bus provides a high performance interface between the microprocessor and the reconfigurable logic. We have chosen Xilinx ML300 as the prototyping platform which contains a Virtex-II Pro FPGA. Our result shows that the design is highly scalable and can accommodate up to 8 PLB slave modules for 256-bit prime generation in an XC2VP125 device operating at 18MHz.

On Computing Maximum Likelihood Phylogeny Using FPGA

Terrence S.T. Mak and K.P. Lam

Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong, Hong Kong SAR, China
{stmak, kplam}@se.cuhk.edu.hk

Abstract. Phylogenetic tree (or phylogeny) is a meaningful tree representation for the evolutionary history of different organisms that it has been shown useful in drug discovery, virus identification and functional genomic study [1]. The objective of this project is to develop efficient FPGA implementations for phylogenetic tree reconstruction algorithms. By taking advantage of hardware high-performance, we explore the possibilities of parallelization and system optimization to provide high-speed acceleration for the phylogeny inference. The *Maximum Likelihood* approach for inferring the phylogeny from molecular data has received much attention [2]. Although the optimal ML phylogenetic tree search problem is classified as NP-hard and it is difficult to find the optimal solution, the GAML algorithm (based on Genetic Algorithm and Maximum Likelihood) has been shown to find a good near-optimal solution in reasonable time [3]. In [4], we have shown that using HW/SW (Hardware/ Software) codesign for GAML implementation can provide significant speed-up when compared with software-only implementation. Our HW/SW system has good potential for handling large scale problems in real applications. In [5], an enhanced version of FPGA design with parallel and pipelined implementation for the likelihood evaluation is proposed. It has been shown 100 times faster than the single-CPU solution for the ML tree evaluation. To reduce precision loss attributed to truncation error in the FPGA, we are developing a dynamic floating-point alike structure based on the fixed-point architecture. We have also studied the implementation of phylogenetic tree reconstruction algorithm in the embedded platform (i.e. VirtexII-Pro Platform FPGA). Significant improvement in data transmission rate between hardware and software and higher clock frequency of FPGA have been realized [6].

References

1. Kishino H., et al.: Maximum Likelihood Inference of Protein Phylogeny and the Origin of Chloroplasts. J. Mol. Evol., 31:151-160, 1990
2. Felsenstein J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol., 17:368-376, 1981
3. P. Lewis: A Genetic Algorithm for Maximum Likelihood Phylogeny Inference Using Nucleotide Sequence Data. Mol. Biol. Evol. 15(3):277-283, 1998
4. Mak S. T. and K. P. Lam: High Speed GAML-based Phylogenetic Tree Reconstruction Using HW/SW Codesign. IEEE Computer Society Bioinformatics Conference 2003, 2003
5. Mak S. T. and K. P. Lam: FPGA-based Computation Maximum Likelihood Phylogenetic Tree Evaluation. Accepted in Field-Programmable Logic and Applications conference, 2004
6. Mak S. T. and K. P. Lam: Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA. Accepted in IEEE Computer Society Bioinformatics Conference 2004, 2004

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1188, 2004.

Minimising Reconfiguration Overheads in Embedded Applications (Abstract)

Usama Malik

University of New South Wales,
Sydney, Australia
umalik@cse.unsw.edu.au

This PhD project seeks to examine the reconfiguration of FPGAs at the architectural level in order to develop efficient techniques for supporting application-specific reconfiguration. This abstract presents a technique that addresses the problem of reducing the reconfiguration delay of an FPGA application at the time when configuration data is to be loaded onto the device. Let us consider an on-chip configuration c_1 and a new configuration c_2 that we want to load onto the device (c_1 and c_2 might not span the entire device). The amount of configuration data that we need to load can be reduced if we only write those parts of c_2 that are not present in c_1. In particular, a judicious placement of the two configurations can result in maximising the amount of *overlap* that we seek to exploit.

We consider the one-dimensional configuration placement problem. Our model is a partially reconfigurable FPGA. The device configurations span the height of the FPGA, reside in a contiguous portion of the memory, and can only be *linearly* shifted across the width of the device. The smallest unit of configuration is a *frame* that spans a column of FPGA resources. It is assumed that the device is homogeneous meaning that the same configuration configures the same circuit no matter where it is loaded.

Minimising the configuration overhead for a sequence of relocatable one-dimensional configurations can be shown to be NP-complete. A greedy algorithm was evaluated for a sequence of thirteen benchmark circuits targeted at Xilinx XCV1000 device. This algorithm places each configuration at a position that minimises the reconfiguration data between it and the on-chip configuration. It was found that the above technique reduces the reconfiguration delay by only 3%. However, it was also found that if the configuration interface supports smaller units of configurations, e.g. sub-frames as small as a single byte, then a reduction in reconfiguration data of up to 85% is possible.

In this work, we have shown the effect of the *frame granularity* and circuit location on configuration caching. Future work involves investigating the design of FPGA architectures that support reconfiguration of one-dimensional relocatable circuits. We are particularly interested in ensuring the correct operation of the system after reconfiguration, and enhancing the IO architecture of the device to support circuit relocation. We believe this technology will benefit the embedded systems domain, in which application characteristics are known a priori and static optimisations can therefore be made to achieve performance goals.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1189, 2004.

Application Specific Small-Scale Reconfigurability

Vinu Vijay Kumar

University of Virginia,
Charlottesville, VA-22903, USA
vv6v@virginia.edu

Fixed logic circuits are the platform of choice for implementing systems with exacting performance, area, and power requirements. However, their inability to accommodate design changes and modifications is particularly disadvantageous in today's world of emerging and rapidly evolving standards and applications. Hardware flexibility, enabling application-specific adaptation to tasks and dynamic adaptation to faults and other run-time events, can greatly increase the efficiency and useful life of the system.

Hardware flexibility is traditionally achieved with large-scale, general-purpose reconfigurable arrays such as FPGAs, which impose significant area, delay, and power penalties compared to fixed logic circuits. Small-scale reconfigurability (SSR) is a design technique that minimizes these penalties by inserting into a fixed-logic design only the flexibility that is required for a specific application. Reconfigurable logic and interconnect (SRAM-based LUTs, MUXes, one time programmable vias, etc.) are finely integrated with the fixed logic at a gate-level granularity. The fine integration and application-specific implementation allows SSR to essentially bridge the gap between the efficiency of ASICs and the flexibility of FPGAs.

This dissertation introduces SSR and methodologies for applying SSR to the implementation of flexible systems for three major applications: *flexible datapaths and controllers* for adaptable systems [1], *complexity-independent fine-grained heterogeneous redundancy* for online testing and fault tolerance [2,3,4], and *design flexibility for engineering change (EC) and yield enhancement*. Preliminary results show large area savings and reliability improvement in SSR-based adaptable systems compared to traditional fixed and fully flexible implementations.

References

1. Vijay Kumar, V., and Lach, J., "Designing, scheduling, and allocating flexible arithmetic components," *International Conference on Field Programmable Logic and Applications*, pp. 1166-9, 2003
2. Vijay Kumar, V., and Lach, J., "Heterogeneous redundancy for fault and defect tolerance with complexity independent area overhead," *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 571-8, 2003
3. Vijay Kumar, V., and Lach, J., "Flexible arithmetic components for area-efficient fault tolerance," *International Conference on Military and Aerospace Programmable Logic Devices*, D7:1-6, 2003
4. Vijay Kumar, V., et al., "A Markov reward model for fault tolerant synchronous dataflow system design," *International Conference on Dependable Systems and Networks*, 2004

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1190, 2004.

Efficient FPGA-Based Security Kernels

Zachary K. Baker[*]

University of Southern California, Los Angeles, CA, USA
zbaker@usc.edu

Current Research Interests: Implementing network security requires significant computational time and energy and becomes particularly challenging in mobile, battery-powered situations. Compared to traditional microprocessors, embedded processors, or DSPs, reconfigurable hardware can provide the computational resources required for these applications more effectively due to large on-chip bandwidth, hardware parallelism, and parameterized customization. The purpose of my work is to demonstrate novel uses of reconfigurable hardware toward real-time network security. Significant advances over state-of-the-art in intrusion detection will be realized by developing reconfigurable hardware architectures for on-the-fly threat and intrusion analysis, creating energy-efficient models and implementations for mobile and sensor networks, and developing efficient soft-IP (Intellectual Property) cores for application-specific reconfigurable fabrics.

Recent Work in Performance-Customized Design: Intrusion Detection Systems use sophisticated rules and pattern matching to detect potential malicious packets and prevent them from entering a network. This filtering requires significant computational resources and is difficult using naïve methods. We utilize FPGAs to provide the computation required. The performance of a design can be measured using the following performance metrics: ease of adaptivity/reconfiguration, throughput, and area/energy.

Modified KMP: The Knuth-Morris-Pratt (KMP) algorithm is a well-known, efficient string matching technique using a single comparator and a pre-computed transition table. We adapt KMP to achieve on-the-fly reconfiguration, high throughput, and moderate area by adding a second comparator and an input buffer. This guarantees the system will accept at least one character in each cycle. The use of the buffered KMP reduces the overall work done, and thereby reduce the area required. We proved that a unit of this architecture will terminate in a maximum of $n + k/2$ cycles for a k-length pattern and an n-character input stream, using a buffer of only $k/2$ elements.

Partitioning and Predecoding: In a separate but related work, a tool was developed for the automatic synthesis of highly area and time-efficient intrusion detection systems using a high-level, graph-based partitioning methodology. Automated system level design allows more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance. Through pre-processing, this tool-based methodology yields designs with competitive clock frequencies that are more area efficient than any other shift-and-compare architectures.

[*] Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by an equipment grant from the HP Corporation.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, p. 1191, 2004.

Author Index

Ababei, C. 874, 1172
Agis, R. 617, 1158
Ahmadinia, A. 847, 1032, 1168
Ahrens, C. 95
Akil, M. 1119
Alex, A. 13
Almenar, V. 494
Amano, H. 105, 464
Amira, A. 991
Anderson, J. 168
Andreu, J. 965
Ang, S.-S. 719
Antonio, J.K. 1170
Antunes, A. 1084
Arias, J. 996
Ariza, F. 1057
Arrais, J. 922
Astarloa, A. 965, 996, 1124
Athanas, P. 33, 414
Azorín López, J. 1104

Baeckler, G. 135
Baena, V. 1062
Bajracharya, S. 1001
Baker, Z.K. 311, 1191
Bala, P. 23
Ballester, F. 954
Bampi, S. 474, 1136
Bansal, N. 891
Baradaran, N. 1113
Bartic, T.A. 637
Batlle, J. 1152
Bazargan, K. 874
Becker, J. 115, 454, 1037
Belenguer, J. 954
Benavides, I. 986
Bensaali, F. 991
Berry, F. 1162
Bertels, K. 434
Berviller, Y. 1027
Bharath, A.A. 394
Bidarte, U. 965, 996, 1124
Bigot, A. 64
Bioul, G. 209
Blodget, B. 801, 1127
Bobda, C. 801, 847, 1032
Bochard, N. 555
Bogliolo, A. 1101
Borgosz, J. 1107
Boschetti, M.R. 474
Bouganis, C.-S. 394
Bourgeault, M. 135
Bouridane, A. 771
Braun, A. 811
Braun, L. 1037
Braunes, J. 781
Brebner, G. 404, 1017
Brião, E. 1042
Bulens, P. 231
Bungardean, C. 344

Calazans, N. 1042
Camozzato, D. 1042
Campi, F. 179
Campregher, N. 322
Canas Ferreira, J. 886
Canella, M. 1101
Canet, M.J. 494
Cantó, E. 1087
Canyellas, N. 1087
Cardells-Tormo, F. 279
Cardoso, J.M.P. 1022
Carrillo, R. 617, 1158
Carro, L. 970, 1136
Carvalho, E. 1042
Casado, F. 43
Castillo, E. 1139
Chalimbaud, P. 1162
Charot, F. 1152
Charpentier, F. 64
Chaudhary, K. 168
Chen, G. 158
Cheng, P. 168
Cheung, C.-C. 189
Cheung, P.Y.K. 54, 200, 322, 374, 394, 1047
Cheung, R.C.C. 1186
Chiesa, C. 179
Choudhary, A. 575
Ciccarelli, L. 179

Clarke, C. 1073
Colom, R. 954
Cong, J. 158
Constantinides, G.A. 200, 374, 1047
Conti, A.A. 1080
Court, T. Van 862, 1080
Crha, L. 1133
Cuadrado, C. 996
Cumplido-Parra, R. 943

Dagher, D. 606
Danilin, A. 852
Denning, D. 546
Deprettere, E. 690
Deschamps, J.-P. 209
Desmet, D. 637
Detrey, J. 221
Devlin, M. 546
Dias, F.M. 1084
Díaz, J. 617, 1158
Dickinson, M. 12
Dimond, K.R. 1165
Dinechin, F. de 221
Diniz, P.C. 1113
Domínguez, M.Ángel 1011
Donlin, A. 811, 1127
Donninger, C. 927
Drutarovský, M. 555, 1146
Duong, V. 270
Dutt, N. 891
Dydel, S. 23

Echanóve, M.A.A. 1062
El-Ghazawi, T. 1001, 1183
ElGindy, H. 906
Ellervee, P. 354
Entrena, L. 43, 751
Ewe, C.T. 200

Fabris, E.E. 1136
Fan, H. 189
Fekete, S.P. 847
Ferguson, M.I. 270
Ferizis, G. 906
Fernandez-León, A. 1062
Ferreira, P. 1084
Ferreira, R. 1022
Fischer, V. 555, 1146
Fons, F. 1087
Fons, M. 1087

Freschi, V. 1101
Frigo, J. 95
Fujii, T. 464
Fukushima, T. 105
Furtek, F. 125
Fuster Guilló, A. 1104

Gadea, R. 954
Gädke, H. 1006
Gaj, K. 1001
Galanis, M.D. 868
García, A. 384, 1139
Garcia, R. 1152
García Chamizo, J.M. 1104
Gayasen, A. 145, 1169
Giansante, R. 179
Glesner, M. 526, 761, 1090
Gokhale, M. 95
Gomez-Arribas, F.J. 1110
Gonzalez, I. 1110
Goutis, C.E. 868
Graf, J. 33
Grandpierre, T. 1119
Griese, B. 842
Grimm, B. 454
Grunwald, D. 1175
Gu, Y. 862
Gupta, R. 891
Gupta, S. 891

Hacker, R. 938
Handa, M. 444, 900
Haubelt, C. 1171
Hauck, S. 536, 857
Herbordt, M.C. 862, 1080
Heřmánek, A. 1149
Herms-Berenguer, A. 279
Hernández, Á. 1073
Hernandez-Ardieta, J.L. 1110
Hinkelbein, C. 791
Hiraki, K. 484
Ho, Q.-T. 959
Hogenauer, E. 125
Hogue, C. 13
Holland, M. 536
Hoppe, B. 933
Hormigo, J. 986
Horta, E.L. 975
Huang, R. 900
Huebner, M. 454, 801, 1037

Hunter, J. 414
Hutton, M. 135
Hwang, J. 424

Ienne, P. 596
Ila, V. 1152
Inaba, M. 484
Inuo, T. 464
Irvine, J. 546
Irwin, M.J. 145
Isserlin-Weinberger, R. 13
Ito, T. 911

Järvinen, K. 1098
James-Roxby, P. 1017
Janiaut, M. 1027
Janiszewski, I. 933
Jayaram, B. 1185
Jia, X. 836

Kalman, S. 168
Kamakoti, V. 1185
Kami, H. 464
Kandemir, M. 145
Kaptanoglu, S. 135
Kerins, T. 74
Keymeulen, D. 270
Khan, J. 669
Khomich, A. 791
Kienhuis, B. 690
Kikuchi, K. 516
Kim, H. 135
Klausmann, A. 1037
Kobayashi, F. 261
Kocan, F. 289
Koch, A. 679, 1006
Koch, D. 1032, 1174
Köhler, S. 781
Konagaya, A. 505
Koutroumpezis, G. 1116
Krasniewski, A. 1067
Krupnova, H. 64
Kugel, A. 791
Kumar, S. 586
Kumar, V.V. 1190
Kurugollu, F. 771

Lam, K.P. 1076, 1186
Lange, H. 679
Lange, S. 251
Lauwereins, R. 637
Layer, C. 648
Lázaro, J. 996, 1124
Lee, A. 135
Lee, D.-U. 364
Lee, K. 145
Lee, K.H. 526
Leong, P. 526
Lewis, D. 135
Lima, E.R. de 494
Liu, J. 189
Liu-Jimenez, J. 43
Lloris, A. 1139
Lockwood, J.W. 301, 975
Lodi, A. 179
López, M. 1087
López-Estrada, S. 943
Lopez-Ongil, C. 43
Lorenz, M.G. 751
Lorenz, U. 927
Ludewig, R. 526
Luk, W. 54, 364, 627, 719, 1047, 1173
Lysaght, P. 1127

Madabhushi, C. 168
Männer, R. 791
Maidee, P. 874
Majer, M. 1032
Mak, T.S.T. 1076, 1186
Malik, U. 1189
Manimegalai, R. 1185
Mannino, C. 1027
ManojKumar, A. 1185
Marek, T. 1133
Marescaux, T. 637
Marnane, W. 74
Martín, J.L. 965, 1124
Martinez, J. 1110
Maruyama, T. 505, 980, 1155
Masselos, K. 881
Massicotte, D. 959
Matos, G.M. 344
May, F. 761
McMillan, S. 424
Mecha, H. 241
Memik, G. 1093
Memik, S.O. 740
Mencer, O. 364
Mengibar, L. 751

Meurice de Dormale, G. 231
Meuth, H. 933
Meyer, J. 289
Meyer-Bäse, U. 384
Middendorf, M. 251
Miglioli, F. 1101
Mignolet, J.-Y. 637
Miller, J. 637
Minnich, R. 95
Mishra, M. 1181
Möller, L. 1042
Mohanty, S. 658
Moraes, F. 1042
Moscu Panainte, E. 434
Mota, S. 617, 1158
Moure, M.J. 1011
Mozos, D. 241
Müller, M. 791
Mukherjee, R. 740
Muñoz, F. 1062
Murgan, T. 761

Nag, S. 168
Nelson, B. 414, 948
Neto, H.C. 1022
Ng, J. 394
Nguyen, D. 575, 1093
Niang, P. 1119
Nicolau, A. 891
Niitsuma, H. 1155
Niyonkuru, A. 801
Novotný, M. 1133
Nurmi, J. 1130

Öers, S.B. 84
Oguri, K. 911
Olivares, J. 986
Ortigosa, E.M. 1158
Osana, Y. 105
Ou, J. 729
Ouaiss, I. 606

Paar, C. 586
Padalia, K. 135
Palmer, J. 948
Park, J. 1113
Parreira, A. 333
Parrilla, L. 1139
Patterson, C. 414
Paya, G. 954
Pearce, D.J. 364
Pedersen, B. 135
Peiró, M.M. 954
Peremans, H. 1073
Petrov, M. 761
Pfänder, O.A. 938
Pfleiderer, H.-J. 648, 938
Phillips, S. 857
Picatoste-Olloqui, E. 279
Pimentel, B. 922
Pionteck, T. 1090
Platzner, M. 831
Poetter, A. 414
Popovici, E. 74
Porrmann, M. 842
Pozzi, L. 596
Pramstaller, N. 565
Prasanna, V.K. 311, 658, 729
Preneel, B. 84
Preußer, T. 781
Pun, K.P. 526

Qiang, L. 1073
Qu, Y. 881
Quintáns, C. 1011
Quisquater, J.-J. 84, 231

Rabah, H. 1027
Raik, J. 354
Rajagopalan, J. 836
Ramírez, J. 384
Rao, S. 384
Ratchev, B. 135
Resano, J. 1178
Ribeiro, P. 1084
Riesgo, T. 1057
Rissa, T. 54
Ristimäki, T. 1130
Robert, F. 637
Roelandts, W. 1
Romero-Troncoso, R. deJ. 1142
Root, A. 424
Ros, E. 617, 1158
Rose, A. 811
Rose, J. 13
Rückert, U. 842
Ruiz, G.H. 1142

Saini, R. 135
Sanchez, L. 43

Sanchez-Pastor, J. 1110
Sanchez-Reillo, R. 43
Sans, I. 64
Santos, M.B. 333
Sawitzki, S. 852
Schelle, G. 1175
Scheuermann, J. 125
Schier, J. 1149
Schleicher, J. 135
Schoeberl, M. 917
Schuehler, D.V. 301
Schüler, E. 970
Sedcole, P. 1047
Segard, A. 710
Sempere-Agullo, J. 279
Septién, J. 241
Sharma, A. 857
Shi, C. 424
Shibata, Y. 911
Shimamoto, B. 2
Shu, C. 1001
Sidahao, N. 374, 1184
Silva, I.S. 474
Silva Matos, J. 886
Šimka, M. 555
Singh, V. 424
Siozios, K. 1116
Siripokarpirom, R. 700
Skarpathiotis, C. 1165
Skliarova, I. 922
Sklyarov, V. 922
Skyttä, J. 1098
Soffke, O. 526
Sorel, Y. 1119
Soudris, D. 868, 1116
Sousa, J.T. de 344
Spallek, R.G. 781
Staake, T.R. 1090
Standaert, F. 84
Steiner, N. 414
Stiefmeier, T. 1090
Stoica, A. 270
Sugawara, Y. 484
Sutter, G. 209
Suzuki, M. 464

Tabero, J. 241
Taher, M. 1183
Tahir, M.A. 771
Tammemäe, K. 354
Tanougast, C. 1027
Tatas, K. 1116
Tavares, C.J. 344
Teich, J. 847, 1032
Teixeira, J.P. 333
Thanailakis, A. 1116
Theodoridis, G. 868
Thomas, A. 115
Thomas, D.B. 1173
Tiensyrjä, K. 881
Tihhomirov, V. 354
Tiri, K. 1052
Todman, T. 627
Toma, M. 179
Tombs, J. 1062
Tommiska, M. 1098
Torralba, A. 1062
Torre, E. del 1057
Torres-Huitzil, C. 943
Torresen, J. 821
Tortosa, F. 1062
Tragoudas, S. 868
Tredennick, N. 2
Tripp, J.L. 95
Troeger, G. 1127
Tsoi, K.H. 526
Tuan, T. 145
Tull, M.P. 1170

Ullmann, M. 454, 1037

Valderas, M.G. 751, 1057
Valdés, M.D. 1011
Valls, J. 494
Vasilko, M. 322
Vassiliadis, S. 434
Veale, B.F. 1170
Veen, J.C. van der 847
Vemuri, R. 444, 669, 836, 900
Verbauwhede, I. 1052
Verdier, F. 710
Verkest, D. 637
Vernalde, S. 637
Vicedo, F. 494
Vijaykrishnan, N. 145
Villalba, J. 986
Vonnahme, E. 842
Vorbach, M. 761
Vuletić, M. 596

Wakabayashi, S. 516
Walder, H. 831
Watanabe, M. 261
Weber, S. 1027
Wilton, S.J.E. 719
Wolkerstorfer, J. 565
Wu, Y.-L. 189

Yamaguchi, Y. 505
Yoshimi, M. 105
Yuan, R. 135

Zabel, M. 781
Zambreno, J. 575, 1093, 1180
Zebulum, R. 270
Ziegler, H.E. 1175
Zipf, P. 526, 761
Zissulescu, C. 690
Zuloaga, A. 1124