University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations Theses, Dissertations, and Major Papers

1-1-2006

Hardware design and CAD for processor-based logic emulation systems.

Amir Ali Yazdanshenas
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Yazdanshenas, Amir Ali, "Hardware design and CAD for processor-based logic emulation systems." (2006). *Electronic Theses and Dissertations*. 7121.
https://scholar.uwindsor.ca/etd/7121

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

Hardware Design and CAD for Processor-based Logic Emulation Systems

by

Amir Ali Yazdanshenas

A Thesis
Submitted to the Faculty of Graduate Studies and Research through
Electrical and Computer Engineering in Partial Fulfillment
of the Requirements for the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2006

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-42312-7
Our file Notre référence
ISBN: 978-0-494-42312-7

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hardware Design and CAD for Processor-based Logic Emulation Systems

by

Amir Ali Yazdanshenas

APPROVED BY:

W. Altenhof, External Examiner
Mechanical Engineering

E. Abdel-Raheem, Departmental Examiner
Electrical and Computer Engineering

M. A. S Khalid, Advisor
Electrical and Computer Engineering

S. O'Leary, Chair of Defense
Electrical and Computer Engineering

December 6, 2006

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© 2006 Amir Ali Yazdanshenas

All Rights Reserved. No Part of this document may be reproduced, stored or otherwise retained in a retreival system or transmitted in any form, on any medium by any means without prior written permission of the author.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

It is fair to claim that the greatest challenge currently faced by IC designers is how they prove that their designs do not contain any functional errors before they actually send them away for fabrication. Given the fact that fabrication of a chip is not only a time-consuming process, but also very expensive, it would be financially devastating for IC manufacturing companies if any functional errors are detected after the chip is fabricated. Logic emulation systems are programmable hardware platforms that help IC designers to verify the correct functionality of their IC designs before they are sent for fabrication. Processor-based logic emulation systems belong to a class of logic emulators that are studied in details in this thesis.

In the first part of this research, a new hardware architecture for processor-based logic emulation system, which was implemented in Xilinx Virtex-II and Virtex 4 FPGAs, has been proposed. Efficiency of proposed architecture in terms of speed, area and other design constraints is compared with other studies. The new approach shows reasonable emulation speed ($200KHz$), better logic utilization ($\geq 67\%$) while reducing the hardware size and cost by orders of magnitude.

More importantly, based on the proposed architecture, a software CAD framework was created that allows automatic mapping of a gate-level netlist into a series of instructions, which can be executed in parallel by a collection of logic emulation processors. Two scheduling algorithms have been developed and implemented. The algorithms were evaluated using several popular benchmark circuits. Experimental results show that the algorithms achieved close to optimal average processor workload (83%) which results in fast emulation speed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To my mother who never stopped loving me.
To my father (Captain) who never stopped supporting me.
To my brothers who never stopped cheering me.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

There are so many people who have directly or indirectly influenced this work that I will remain thankful to all of them for the rest of my life because, without them, I would have never made it this far. First and foremost is Dr. Mohammed Khalid, my supervisor, to whom I would like to express my most sincere gratitude for his invaluable encouragement, guidance and support. To Dr. Esam Abdel-Raheem for his infinite kindness and patience, I would like to express my appreciation from the bottom of my heart. Next, I would like to thank Dr. William Altenhof, whose scientific and precise approach towards details has shed so much light into my work. Also, special thanks goes to Dr. O'Leary who kindly accepted to chair my defense session. I shall truly thank Dr. Majid Ahmadi for always believing in me and accepting me into this program. Also, I would like to thank Dr. Roberto Muscedere for answering technical questions I encountered in the RCIM lab and his professional help during the course of this research.

Added to these gentlemen, are my dearest colleagues and friends at the Department of Electrical and Computer Engineering. My best wishes go to Kevin Banović, Jason Tong, Raymond Lee, Harb Abdulhamid, Ian Anderson and Marwan Kanaan for creating such a wonderful and pleasant atmosphere to work at. To my friends Behdad Elahipanah, Nima Bayan, Mohammad Naserian, Amr Elkholy, with whom I spent such a wonderful time playing soccer, I wish them success in all aspects of their lives. And last, but not least, I have to express my thanks to Ms. Andria Turner and Ms. Shelby Merchand for being so generous and helpful to me throughout these years.

This research was funded by the National Science and Engineering Research Council (NSERC) and the University of Windsor, while the Canadian Microelectronics Corporation (CMC) has provided all our FPGA lab equipments, VLSI CAD software and technical support. Their contribution is also gratefully acknowledged.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Abstract iv

Dedication v

Acknowledgments vi

List of Figures x

List of Tables xiii

List of Abbreviations xiv

1 Introduction and Motivation 1

- 1.1 Thesis Overview 2
- 1.2 Thesis Organization 3

2 Background and Previous Work 4

- 2.1 History of Design Verification 5
  - 2.1.1 Formal Verification 6
  - 2.1.2 Simulation 7
  - 2.1.3 Hardware-Accelerated Simulation 8
  - 2.1.4 Rapid Prototyping 9
  - 2.1.5 Logic Emulation 9
- 2.2 Architecture of Logic Emulation Systems 9
  - 2.2.1 FPGA-Based Logic Emulation System (FBE) 11
    - 2.2.1.1 Introduction to Field-Programmable Gate Array 11
    - 2.2.1.2 Architecture of FPGA-Based Logic Emulation Systems 14

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A Mesh Interconnect 15
B Full Crossbar Interconnect 15
C Partial and Hierarchical Partial Crossbar 16
D Hybrid Complete Graph Partial Crossbar 18
E Virtual Wire Architecture 19
F Time-Multiplexed FPGA Architecture 21
2.2.1.3 Emulating Logic Designs on FBEs 22
2.2.2 Processor-Based Logic Emulation System (PBE) 23
2.2.2.1 Architecture of PBEs 24
A PBEs with Homogeneous Architecture 24
B PBEs with Heterogeneous Architecture 25
2.2.3 Logic Emulation Systems in Industry 25
2.3 CAD Flow for Logic Emulation Systems 26
2.3.1 Introduction 26
2.3.2 CAD Flow for FBEs 27
2.3.3 CAD Flow for PBEs 32

3 Architecture of Hybrid Emulation Processor (HEP) 35
3.1 Top-Level Organization the Emulation Engine 35
3.2 How a Logic Design is Emulated? 36
3.3 Structure of Hybrid Emulation Processor 37
3.4 Instruction Set Architecture of HEP 39
3.5 Central Control Unit of HEP 42
3.6 Control Memory of HEP 48
3.7 Data Memory of HEP 49
3.8 Input/Output Ports of HEP 50
3.9 HEP's Program Counter Register (Global Sequencer) 51
3.10 Additional Signal Pins of HEP 52

4 Implementation of Hybrid Emulation Processor on FPGA 54
4.1 Introduction 54
4.2 Design Specifications for HEP-based Emulation Engine 55
4.3 RTL Design of HEP-Based Emulation Engine 57
4.3.1 RTL Modeling of HEP Processor 57

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4 RTL Simulation Results 59
4.5 Synthesis Results 60
4.6 Comparison and Conclusion 66

5 A CAD Tool Suite for HEP-based Emulation System 69

5.1 Basic requirements for HEP-based CAD tool 69
5.2 Overall CAD Flow 71
5.2.1 Design Entry 71
5.2.2 Synthesis 72
5.2.3 Technology Mapping 73
5.2.4 Scheduling 75
5.2.4.1 Preliminaries 77
5.2.4.2 Levelization 79
5.2.4.3 Modified List Scheduling (MLS) 84
5.2.4.4 MLS+BFF Scheduling 88
5.2.4.5 Mathematical Formulation 89
5.2.4.6 Evaluation Metrics 93
5.2.4.7 Implementation of MLS/MLS+BFF Scheduling Tools 95
5.2.5 Experimental Results 95
5.2.6 Code Generation and Download 100
5.3 Comparison and Conclusion 100

6 Conclusions and Future Work 103

6.1 Future Work 104
6.1.1 Improvements in Hardware Architecture 104
6.1.2 Improvement in Design Compiler Tool 104

References 106

VITA AUCTORIS 111

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

2.1 Complexity/Productivity growth versus time in terms of number of transistors[66] . 5
2.2 General view of software simulation tools 7
2.3 General view of a Logic Emulation System 10
2.4 Logic emulation system connectivity 11
2.5 Internal view of a typical FPGA 12
2.6 Structure of a 3-input LUT ($k = 3$) 13
2.7 Internal structure of a generic logic element 14
2.8 A generic FPGA-based logic emulation system 14
2.9 Mesh architecture 15
2.10 Internal structure of a field programmable interconnect device (FPID) 16
2.11 Logical view of full crossbar interconnect (a). Block view (b). 17
2.12 Logical view of partial crossbar interconnect (a). Block view (b). 17
2.13 Example of two-level hierarchical partial crossbar architecture. 18
2.14 Hybrid complete graph partial crossbar architecture 19
2.15 A genric view of non-time-multiplexed signals among two partitions. 20
2.16 Generic view of Virtual Wire architecture 21
2.17 Time-multiplexed FPGA configuration model. 22
2.18 General view of one logic element in a time-multiplexed FPGA. 23
2.19 General view of a Homogeneous PBE system 25
2.20 General view of a Heterogeneous PBE system 26
2.21 CAD flow for FBEs. 31
2.22 CAD Flow for PBEs 34

3.1 Block diagram of an emulation module 36

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2 Internal structure of HEP Processor. 37
3.3 Example of implementing function F in a 4-input LUT. 38
3.4 Fields of LUTOP instruction 40
3.5 Fields of RAMREF instruction 40
3.6 Fields of ROMREF instruction 41
3.7 Fields of NOP instruction 42
3.8 HEP's Control Unit Finite State Machine 44
3.9 HEP's Control Memory structure. 49
3.10 HEP's Data Memory structure. 50
3.11 HEP's Input/Output structure. 51
3.12 HEP's Program Counter (Global Sequencer). 52
3.13 HEP's Pin-out Map. 53

4.1 Generic architecture of HEP-based emulation engine 55
4.2 Example of Signal Trap circuitry. 56
4.3 FPGA Design Flow 58
4.4 Hierarchy of VHDL design files for HEP Processor 59
4.5 Example of 4x4 Sequential Binary Multiplier 60
4.6 Simulated waveform view of Program Counter Submodule 61
4.7 Simulated waveform view of 4-input LUT 61
4.8 Simulated waveform view of 64-input interconnect switch 61
4.9 Simulated Read/Write cycles of IDR and LDR 61
4.10 Simulated Read/Write cycles of Left Control Memory 62
4.11 Simulated Read/Write cycles of Right Control Memory 62
4.12 Simulated functionality of Central Control Unit while executing a LUTOP instruction. 63
4.13 Simulation of emulation program being Downloaded/Executed on a processor 63

5.1 Design cycle versus Emulation Cycle in a generic DUT. 70
5.2 CAD Flow for HEP-based emulation system. 71
5.3 RTL view of binary multiplier produced by Synopsys Design Compiler 72
5.4 Gate-level view of binary multiplier generated by Synopsys Design Compiler. 73
5.5 Example of technology mapping for reducing area and delay. 75
5.6 Technology decomposition. (a) Balanced-tree, (b) Unbalanced-tree. 76
5.7 Modeling a DUT as a Mealy Machine. 78

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.8 DAG representation of netlist 79
5.9 Instruction memory map (IMM) of HEP-based emulation system. 80
5.10 ASAP Levelization 81
5.11 ASAP algorithm in pseudo code. 82
5.12 ALAP Levelization 82
5.13 ALAP algorithm in pseudo code. 83
5.14 Processor workload after levelizing "elliptic". 83
5.15 MLS algorithm 85
5.16 Examples of collapsing nodes during IMM allocation. 87
5.17 Prioritizing nodes with equal mobility with respect to their fan-out degree. 90
5.18 MLS+BFF algorithm 91
5.19 Executing program on a parallel platform 94
5.20 Example of output generated by GSchedule tool. 96
5.21 Resource utilization in HEP-based emulation system and FBEs. 102

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

4.1 Synthesis results of HEP processor and submodules. 64
4.2 Synthesis results of HEP-based emulation system. 65
4.3 Comparison of HEP with other Emulation systems 68

5.1 Ten biggest MCNC circuits. 74
5.2 Results of technology mapping. 76
5.3 Processor workload calculated after MLS scheduling. 97
5.4 Processor workload after MLS scheduling on multiplier. 97
5.5 Processor workload after MLS+BFF scheduling. 98
5.6 Emulation time and speed-up obtained by MLS scheduling. 99
5.7 Emulation time and speed-up obtained by MLS+BFF scheduling. 99
5.8 Comparing emulation time of HEP and VEGA 101

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Abbreviations

Abbreviation	Definition
C_L	Critical path length.
P	Number of HEP processors.
W	Number of instruction words per HEP.
ϕ	Processor workload.
$\overline{\phi}$	Average processor workload.
λ	Speed-up.
ALAP	As-Late-As-Possible (levelization).
ASAP	As-Soon-As-Possible (levelization).
ASIC	Application-specific integrated circuit.
BDD	Binary decision diagram.
BFT	Breadth-first traversal.
BLIF	Berkeley logic interchange format.
CAD	Computer aided design.
CMC	Canadian microelectronics corporation.
CPN	Critical path node.
CPS	Cycles per second.
DAG	Directed acyclic graph.
DFT	Depth-first traversal.
DMM	Download manager module.
DRC	Design rule checking.
DSP	Digital signal processing.
DUT	Design under test.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abbreviation	Definition
EDA	Electronic design automation.
FBE	FPGA-based emulation system.
FF	Flip-flop.
FPGA	Field-programmable gate array.
FPID	Field-programmable interconnection device.
FSM	Finite state machine.
HCGP	Hybrid Complete Graph Partial Crossbar.
HEP	Hybrid Emulation Processor.
HOL	Higher-order logic.
IC	Integrated circuit.
IDR	Input data RAM.
I/O	Input/output.
IEEE	Institute of electrical and electronics engineers.
IMM	Instruction memory map.
LDR	Local data RAM.
LE	Logic element.
LSB	Least significant bit.
LSI	Large scale integrated (circuit).
LUT	Look-up-table.
MCNC	Microelectronics center of North Carolina.
MFS	Multi-FPGA system.
MIMD	Multiple instruction multiple data.
MLS	Modified list scheduling.
MSB	Most significant bit.
MUX	Multiplexer.
PBE	Processor-based emulation system.
PE	Processing element.
RTL	Register transfer level.
TPG	Task precedence graph.
TTL	Transistor-Transistor Logic.
VHDL	Very high speed integrated circuit (VHSIC) hardware description language.
VLSI	Very large scale integrated (circuit).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction and Motivation

Ever since digital VLSI circuits came into existence engineers have been facing the constantly growing problem of verifying correct functionality of circuits before they are sent for fabrication. Once the chip is fabricated, which is a very expensive procedure, it would be impossible for designers to modify the hardware in case design errors were detected, unless they go through all the design steps again.

Several functional verification methodologies such as software simulation and hardware-accelerated simulation have been proposed so far. Each method has a number of advantages as well as disadvantages. A briefly review of all these methods is presented in future chapters. Traditional verification methods are not effective for very large IC designs. Consequently, finding faster, cost effective and more accurate solutions for design verification is a very important research issue.

The most effective method for performing functional verification of an IC design prior to fabrication is *Logic Emulation.* A logic emulation system (also known as logic emulator) is a field programmable system that can be programmed to emulate (i. e. imitate) the functionality of a digital circuit at speeds of millions of *cycles per second*(CPS).

During past few years many logic emulation systems have been proposed and implemented. The two main classes of logic emulation systems are *FPGA-based logic emulation* (FBE) and *processor-based logic emulation* (PBE) systems. Each of these systems have a number advantages as well as disadvantages. In most cases these systems might be so complex and expensive that it would be financially infeasible for small or medium size companies to afford. Currently, there is a demand for cheaper logic emulation systems that are fast and yet large enough to verify designs as big as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

multi-million gates.

More importantly, all logic emulation systems have an associated set of mapping CAD tools (called *design compilers*) that perform the task of *design compilation.* The design compiler reads the netlist of the *design under test*(DUT) and automatically converts it to a downloadable bit-stream that can be "programmed" into the logic emulation system. Once the logic emulation system is programmed, design engineers can verify the functionality of DUT by "running" it on the emulation system. Much work remains to be done in exploring new architectures and mapping CAD tools for logic emulation systems.

1.1 Thesis Overview

The main goals of this thesis are:

1. Investigate a cost effective architecture for processor-based logic emulation systems targeting FPGAs. The motivation is to combine the advantages of both FBEs and PBEs in a single system.
2. Create a CAD framework for automatic mapping of DUT netlist to a target processor-based logic emulation system.
3. Explore new scheduling algorithms for mapping design netlists into a collection of parallel processors.

In the first part of this research, a hardware architecture for processor-based logic emulation system has been proposed which was implemented in Xilinx Virtex-II and Virtex 4 FPGAs. Efficiency of proposed architecture in terms of speed, area and other design constraints is compared with other studies.

More importantly, based on the proposed architecture, a software CAD framework that can automatically map a gate-level netlist into a series of instructions, which can be executed in parallel on a collection of logic emulation processors, has been discussed. In addition to software CAD framework, two scheduling algorithms have been proposed and implemented. The algorithms were evaluated using several popular benchmark circuits and experimental results show that the algorithms achieved close to optimal average processor workload which results in fast emulation speed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2 Thesis Organization

This thesis is organized as follows:

In Chapter 2 the history and importance of functional verification is briefly reviewed and various hardware architectures for logic emulation systems are presented. Then the CAD flow and algorithms used in each class of logic emulation system is discussed. In Chapter 3, the hardware architecture proposed in this research is explained and later in Chapter 4 the implementation results of the proposed architecture are described. Chapter 5 covers the CAD framework for mapping design netlists on to the target logic emulation system. Also, two scheduling algorithms are introduced and explained in detail as to how they improve the emulation speed. The experimental results obtained by running the new algorithms on ten MCNC benchmark circuits are presented. Finally, Chapter 6 provides concluding remarks and a discussion of possible future work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Background and Previous Work

In 1965, Gordon Moore predicted that the number of transistors per unit area in a typical integrated circuit (e. g. microprocessor) will double roughly every 18 months [51]. This increase in the integration level is called *semiconductor productivity* [35], or better known as *Moore's law.* Another implication of semiconductor productivity is that greater functionality is being integrated into unit area of semiconductors, which results in a direct increase in design complexity. Therefore, some researchers refer to such trend in semiconductor productivity as *complexity growth.*

On the other hand, the term *design productivity* refers to the number of logic gates designed by single designer per day [35]. Statistics from real world show that although semiconductor productivity keeps increasing with the pace expected by the Moore's law, design productivity is not improving proportionally, resulting in what we would like to call *production gap* or, as it will be explained shortly, *verification gap* (Fig. 2.1). The existence of such a gap is due to two main reasons: first, increase in the number of circuit elements and their interconnection (i. e. design size). Second, increase in the number of test vectors to verify the correctness of all circuit elements. For example, if there are N circuit elements (such as logic gates or flip-flops) within the digital circuit under test and each element can assume a binary value (0 or 1), then we need at most 2^N test vectors to thoroughly verify the functionality of the circuit. It goes without saying that even for a very small circuit ($N < 100$) it is practically impossible to fully verify the correctness of the design as the number of test vectors (2^{100}) is almost infinite. To avoid design errors and possible expensive silicon re-spins, chip manufacturers are looking for solutions to functionally verify their

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

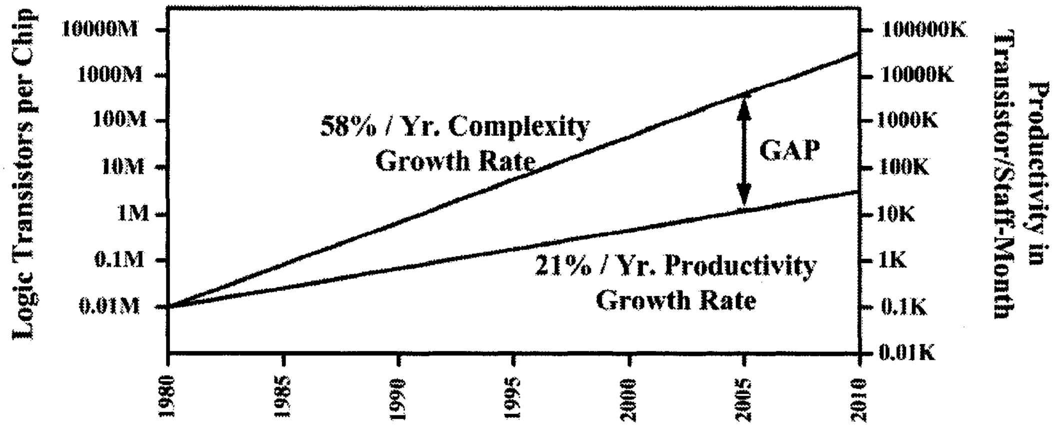


Figure 2.1: Complexity/Productivity growth versus time in terms of number of transistors[66]

designs before fabrication, often referred to as *design verification.* In fact, it would be fair to say that, design verification has become the most important bottleneck in the design process, requiring about 60-75% of design resources such as design time, computing resources and man-power [53][41]. Therefore, many researchers are targeting this area to narrow the verification gap or at least keep it from increasing as the design size grows.

2.1 History of Design Verification

There are many different ways for tackling the design verification problem, some of which have been around for a while. In general, there are five different methods used for design verification:

1. Formal Verification
2. Simulation
3. Hardware Accelerated Simulation
4. Rapid Prototyping
5. Logic Emulation

Each method has a number of advantages as well as drawbacks. In the semiconductor and electronic industries, some or all of these methods are used to verify designs, based on design complexity and verification requirements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.1 Formal Verification

Formal verification refers to a process through which a designer proves formally that a designed circuit satisfies the design specifications for all possible inputs [41]. The behavior of a hardware design is described formally and then the correctness of the design is proved by using a number of mathematical proof techniques [71] [27]. In formal verification, first the hardware is represented using, logic equations or *finite state machines* (FSMs), regardless of other design aspects such as timing or area constraints. Then, the designer studies the question of whether the designed circuit matches the specifications or not. The specifications are often written as a set of temporal logic formulas. For obvious reasons, some researchers believe that formal verification methods are simply parts of the design process and not a post-design process.

Two most common approaches for formal verification are *theorem proving* (algorithmic verification) and *model checking* (deductive verification). Model checking tools represent the design using *Binary Decision Diagrams* (BDDs) and the specifications by a set of temporal logic formulas [10][15]. The model checking tool then traverses the BDD by exploring all possible combinations of inputs/states/outputs to verify if the formulas are satisfied. On the contrary, in theorem proving techniques, both the hardware and its specifications are represented in some abstract logic such as *Higher-Order Logic* (HOL). Then, a mathematical proof within the rules of that logic is constructed that shows the design matches its specifications. Theorem proving tools automate the process of establishing the proof [23].

Since formal verification methods use mathematical approach to determine the correctness of a design, therefore all possible errors in the design will be detected and sound functionality of the design is guaranteed. However, they have a number of drawbacks which limit their usage for real world designs. For instance, formal verification methods are not easily scalable and they all suffer form state-space explosion. That is, if there are 250 memory cells within the circuit, then the circuit would have 2^{250} states [1] that need to be exhaustively searched. On the other hand, finding mathematical abstraction (model) for even a small design is a complicated and tedious task and requires lots of knowledge and experience. To overcome these problems, researchers have tried to combine different formal verification methods together [23], but the results are still not suitable for large designs.

---

[1] Just a bit more than the number of all particles in the universe!

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

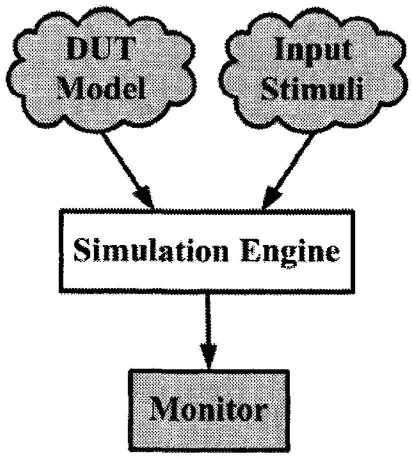


Figure 2.2: General view of software simulation tools

2.1.2 Simulation

By far the most popular verification method is *software simulation*, or simply, *simulation*. The inputs to a logic simulator are the design netlist file and input stimulus signals, often in the form of vector data files. The simulator computes how the design-under-test (DUT) would operate over time and generates required outputs, given those inputs [4][1]. It is then the designer's job to observe the outputs produced by the simulator and verify if the design is operating correctly. The comparison process can be automated by defining "monitors" for the simulation tools. It should be emphasized that, in the simulation technique, not only the input stimuli to the DUT are represented in software (e. g. vector data files) but also the DUT itself is represented in software. Therefore, it is obvious that the simulator is nothing but a software simulation "engine" that runs the models of a DUT against given input vector files (Fig. 2.2). In more recent design methodology, designers use hardware description languages (HDL), such as Verilog or VHDL, to not only describe the design, behaviorally or structurally, but also specify input stimuli and monitoring routines within the same embodiment, called *test bench* (shown by shaded blocks in Fig. 2.2) [56]. Software simulators have a number of advantages over other verification tools:

- They provide extensive capabilities for modifying and debugging the design which is due to the intrinsic flexibility in software.
- They are much easier to use.
- They are significantly cheaper than other tools.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The above benefits make simulators the most widely used verification tools. However, they do have limitations:

- As the size of logic designs doubles the amount of computing work to simulate them roughly quadruples. A rough estimation for such increase is that, an increase in the number of logic gates not only increases the number of cycles, but also it increases the computational work per cycle to get acceptable coverage [48]. Hence, software simulators are simply too slow to simulate designs with more than a million gates. Typically their simulation speed is around tens of *cycles-per-second*(CPS).
- Simulators do not provide the *in-circuit emulation*(ICE) capability.
- The accuracy of simulation results depends solely on how well the designer has modeled the DUT in software and the number of test vectors (input stimuli) provided. Therefore, user expertise is a key factor in simulation accuracy.

If we only use simulators for design verification, it is very likely that some design errors remain undetected. A notorious example of such an incident was the design bug in the floating point arithmetic unit of Intel's Pentium processor, reported in [54], which caused a financial loss of several million dollars to the company.

2.1.3 Hardware-Accelerated Simulation

To overcome the speed limitation of software simulators, simulation accelerators based on custom hardware were developed. These accelerators provided built-in test equipment (such as signal generators and logic analyzers). Instead of using computer workstations, designers could execute the simulation of their designs on a number of parallel processors which run orders of magnitudes faster than simulators [3][16][61].

Although, hardware-accelerated simulators provided good speedup for simulation, they still suffered from two major problems:

- It should be emphasized that hardware-accelerated simulators are still using software models of the design and not real hardware.
- Massively parallel processing platforms succeed in physical simulation such as fluid flow or structural analysis but they are not efficient enough in simulation of logic designs because logic designs have very irregular topologies [48].
- They do not provide in-circuit emulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.4 Rapid Prototyping

Another relatively less popular functional verification method is *rapid prototyping.* In this method designers quickly produce hardware models of the actual product that is fabricated by using fast prototyping platforms such as programmable logic technology. By examining the functionality of those models, designer can identify possible errors in their design before they send it for fabrication. Unfortunately, the feasibility of rapid prototyping technique depends highly on the type of the application and availability of tools. In one example, researchers have created a flexible environment to develop only *digital signal processing*(DSP) applications [33].

Since rapid prototyping requires building a hardware sample closest to the final product, the verification process will be fastest and detection of most design errors is likely. However, the main disadvantage is that once the prototype is built it can not be used for other applications and therefore it would be a throw-away effort.

2.1.5 Logic Emulation

The most recent verification tools are *logic emulation systems.* A hardware emulator is a completely programmable hardware system which can be programmed to imitate (i. e. emulate) the functionality of a large digital design (tens of million gates) at the speed of multi million cycles per second (CPS). In other words, a logic emulator is a programmable device that, once programmed, functions just like a prototype of the final chip before actually fabricating the chip itself.

Logic emulation systems have a number of advantages over other verification tools that have recently brought them into spotlight. In the upcoming sections we will be thoroughly investigating the hardware architecture and CAD tools for logic emulation systems.

2.2 Architecture of Logic Emulation Systems

So far a number of hardware architectures for logic emulation systems have been proposed, and some of these architectures have been implemented. Regardless of their architecture, they all share a number of basic features. Generally speaking, a typical logic emulation system consists of five major components which their connectivity is shown in Fig. 2.3.

1. Programmable hardware

2. CAD tools which automatically map design-under-test (DUT) into downloadable bit stream for the programmable hardware

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

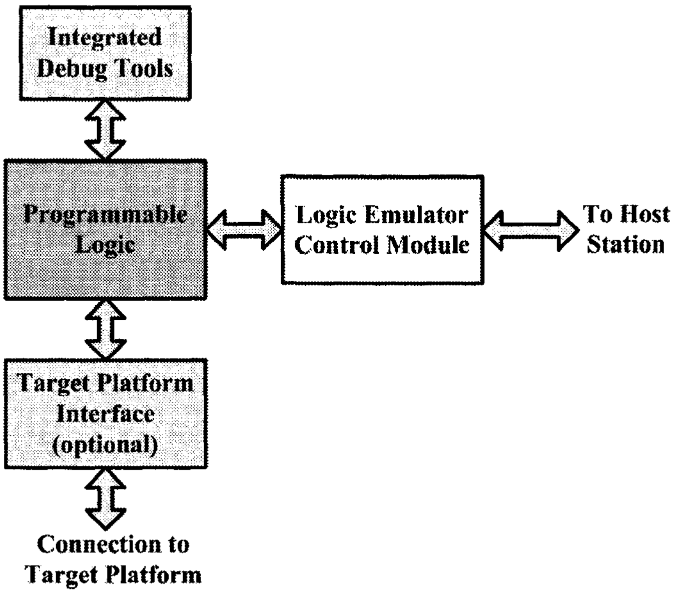

Integrated Debug Tools
Programmable Logic
Logic Emulator Control Module
To Host Station
Target Platform Interface (optional)
Connection to Target Platform

Figure 2.3: General view of a Logic Emulation System

3. Integrated instrumentation and debugging hardware such as integrated logic analyzers (ILA) or programmable signal generators
4. Integrated control hardware and software to support the run time environment of the emulated design
5. Target hardware interface circuitry

Figure 2.4 illustrates physical connectivity of a typical logic emulator in the real world environment. A logic emulator can be either connected directly to a single workstation or a collection of workstations through a network (e. g. LAN). A set of back-end and front-end CAD tools run on workstations. On the other end, a logic emulator can be connected to the target hardware, right in the socket where the to-be-emulated chip will be mounted in future.

Logic emulation systems are classified according to the architecture used in their programmable hardware. Although various companies and academic researchers have used different architectures, they can all fall into one of the following two categories:

1. FPGA-Based Emulators (FBE)
2. Processor-Based Emulators (PBE)

As it will be explained later the proposed architecture combines some of the properties of both classes of logic emulation systems. Thus the newly proposed emulation system will be referred to as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

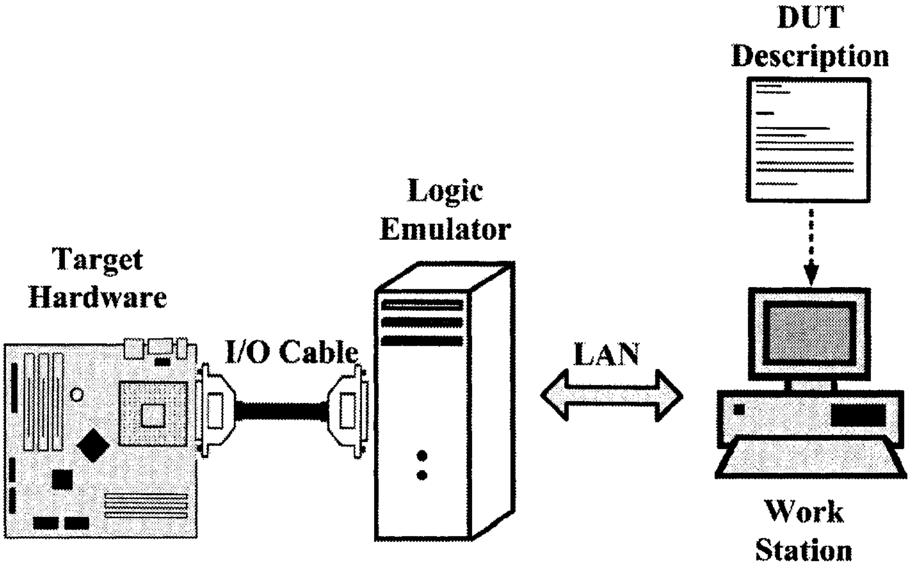


Figure 2.4: Logic emulation system connectivity

hybrid logic emulation system.

2.2.1 FPGA-Based Logic Emulation System (FBE)

Ever since *Field-Programmable Gate Arrays* (FPGAs) were introduced in late 80s, they have been extensively used in rapid prototyping and logic emulation platforms. Since FPGAs are fundamental building blocks of *FPGA-based emulation systems*(FBEs), first, we will briefly review the internal structure of a typical FPGA chip.

2.2.1.1 Introduction to Field-Programmable Gate Array

An FPGA is a flexible, completely re-programmable logic chip. While different FPGA manufacturers have introduced different architectures [55][8], the most popular FPGA architecture contains a two dimensional array of SRAM-based programmable *logic elements* (LE) (Fig. 2.5). The logic elements are interconnected through horizontal/vertical metal wires and SRAM-controlled interconnecting switches (shown at the bottom of Fig. 2.5).

Each logic element consists of two parts: a k-input *look-up table*(LUT) and a flip-flop. A k-input LUT consists of an array of $2^k \times 1$ SRAM-based memory cells. All k inputs to an LUT are address inputs to that memory array and the value read from a memory cell is the output of the LUT. A k-input LUT can realize any logic function of k inputs by programming the truth table values of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

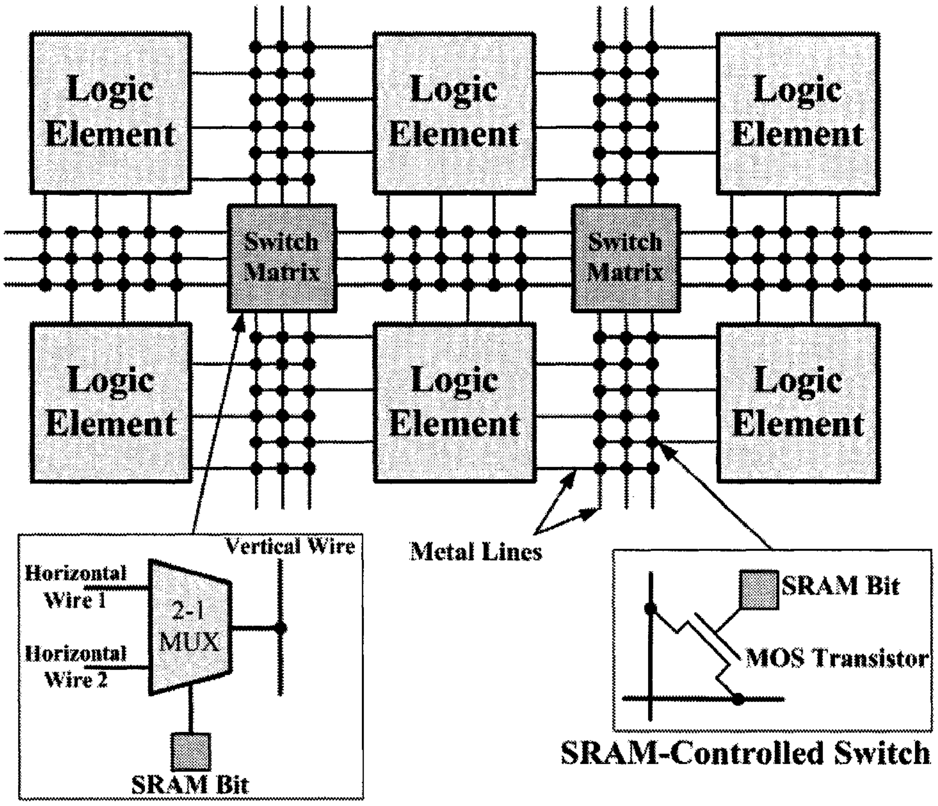


Figure 2.5: Internal view of a typical FPGA

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

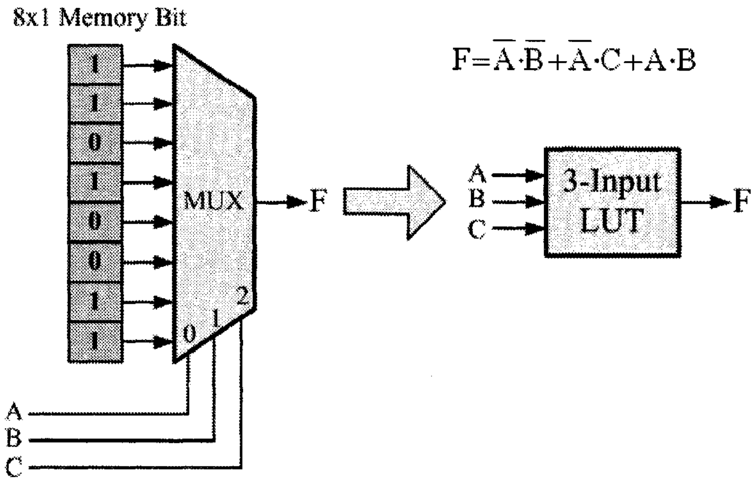


Figure 2.6: Structure of a 3-input LUT ($k = 3$)

logic function directly into the memory array. An example of a 3-input LUT is shown in Fig. 2.6 that implements Boolean function F.

A combination of a k-input LUT and a flip-flop is capable of producing all feasible combinational or sequential logic functions that can be built using k input signals. The option of choosing between the combinational or sequential output can be made by configuring the programmable bit connected to the output multiplexer shown in Fig. 2.7. Typical LUTs have three to six inputs ($3 \leq k \leq 6$), however it has been shown the best performance-versus-area is achieved by having $k = 4$ [60]. Along with the programmable logic described above, an FPGA includes a great number of SRAM-based programmable switches and interconnecting switch matrices (shown at the bottom of Fig. 2.5) which enables arbitrary interconnection among logic elements. The process of interconnecting logic elements together is called *routing.* At the perimeter of an FPGA chip, programmable I/O pins connect the FPGA's internal logic to the outside circuitry. Based on the above descriptions, it is obvious that an FPGA is a highly programmable device that can be configured (programmed) to implement any digital circuit.

It should be emphasized that commercially available FPGAs are much more complicated in architecture. They usually include embedded memory blocks, dedicated fast logic for arithmetic operations as well as complicated logic element architecture. Medium-size commercial FPGAs have a logic capacity of few thousands logic elements equivalent to few tens of thousands logic gates[20][39]. Although this capacity might sound large enough for some applications, it is not big enough for most logic design today. Therefore, FPGA manufacturers are periodically introducing newer FPGAs with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

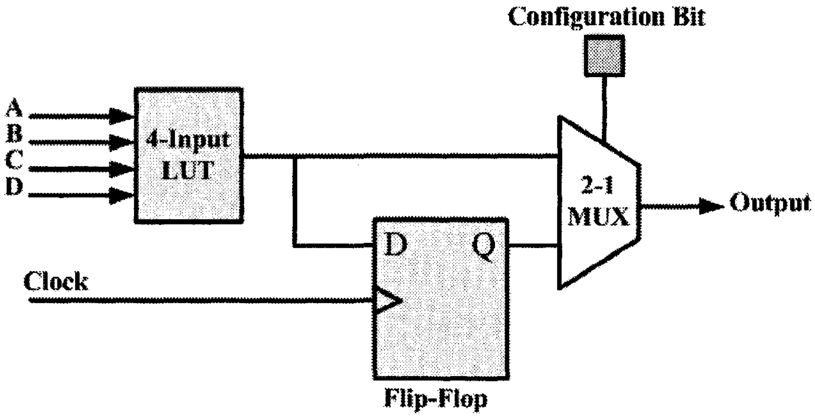


Figure 2.7: Internal structure of a generic logic element

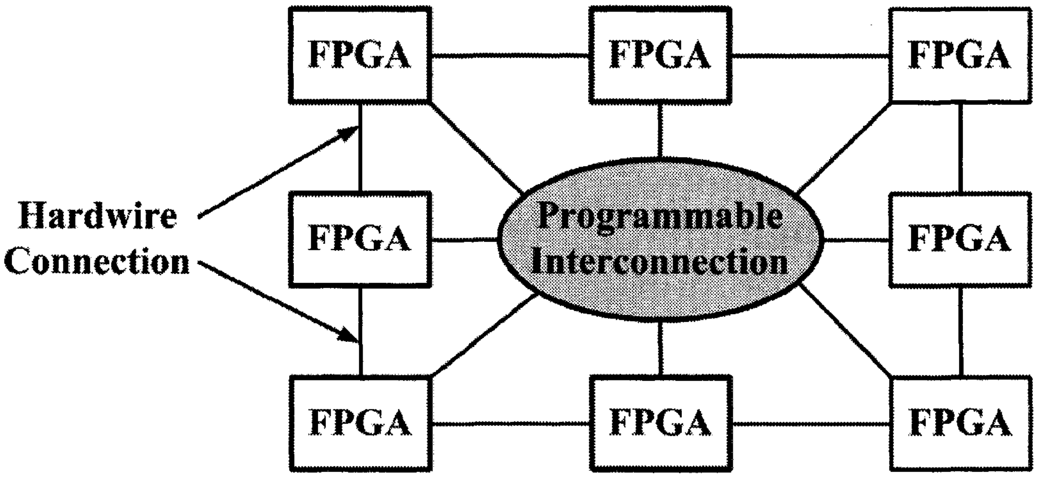


Figure 2.8: A generic FPGA-based logic emulation system

higher logic capacities.

2.2.1.2 Architecture of FPGA-Based Logic Emulation Systems

The programmable hardware section of FPGA-based emulators consists of a collection of FPGA modules interconnected through hardwires and/or *Field Programmable Interconnection Devices*(FPIDs) (Fig. 2.8) [67][11][65].

From the architecture point of view, programmable interconnection devices are quite similar to programmable routing resources inside FPGA chips. In other words, an FPID is a collection of programmable switches and switch matrices. Thus the combination of multiple FPGAs and FPIDs can create an extremely flexible and powerful platform for logic emulation and prototyping.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

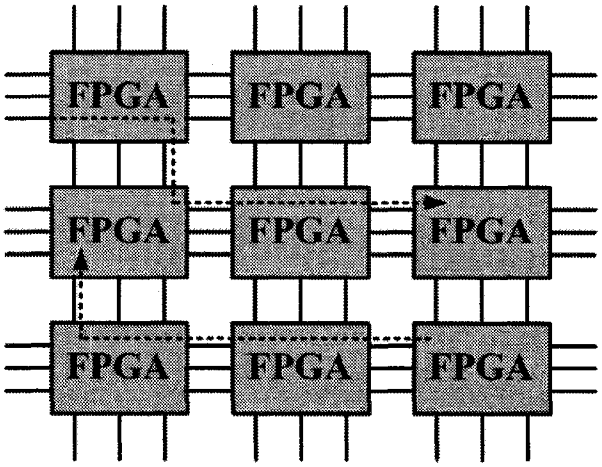


Figure 2.9: Mesh architecture

The “routing architecture” of an FBE is the way in which the FPGAs, fixed wires and FPIDs are connected. Previous research has shown that the routing architecture has a strong effect on the speed, cost and routability of emulation systems. This is because an inefficient routing architecture may require excessive logic and routing resources when implementing circuits and cause long routing delays. Increased routing delays will profoundly slow down the emulation speed.

Several routing architectures for FBEs have been proposed. The routing architecture in FBEs plays a key role in determining the cost and performance of these systems[70].

A Mesh Interconnect Early FBEs did not use any FPIDs. Instead the FPGAs were arranged in a two dimensional array and each FPGA was connected to its nearest neighboring FPGAs (mesh) using hardwired connections (Fig. 2.9) [34].

Although mesh architecture is simple, it has a number of limitations which has made it obsolete. In this architecture, FPGA I/O pins are not only used for connecting FPGA internal logic to outside world, but also for routing inter-FPGA signals. Therefore a large percentage of FPGA I/O pins will be used up for inter-FPGA routing purposes. Moreover, some nets might pass through many intermediate FPGAs in the mesh, which results in very long interconnect delays for some signals. Not only does this slow down the design emulation but also creates unbalanced propagation delays among signals that can induce incorrect or unwanted behavior in some time-sensitive signals, (e. g. set-up/hold time violations).

B Full Crossbar Interconnect An alternative to using FPGAs for routing is to use *field-programmable interconnection device*(FPID), which is a semiconductor device that can be pro-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

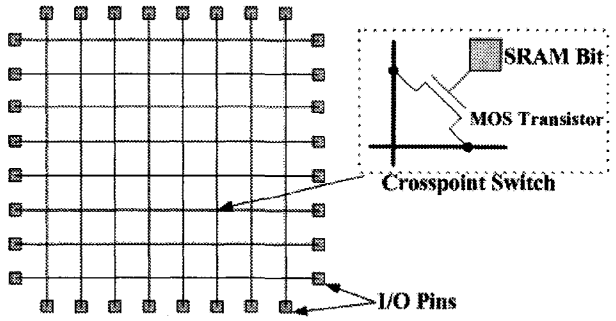


Figure 2.10: Internal structure of a field programmable interconnect device (FPID)

grammed (i. e. configured) to provide arbitrary connections between its I/O pins. It contains a two dimensional array of, usually SRAM-based, programmable switches (Fig. 2.10). Therefore it is capable of making any one-to-one or one-to-many connections between its I/O pins [21]. A typical FPID may have as many as 1000 I/O pins.

In most recent FBE systems FPIDs are being used for interconnecting signals among FPGA pins. The simplest architecture is *Full Crossbar* architecture. In this architecture each FPID is connected to "all" FPGAs on the emulation board (Fig. 2.11). Since a full crossbar is capable of connecting any two pins in the system it is logical to think of this architecture as a regular array of programmable crosspoint switches. Although a full crossbar architecture guarantees successful routability of all nets, it is utilized in small emulation systems with only a very few number of FPGAs. This is because the size (area) of FPID crossbar increases as the square of number of its I/O pins. Equation 2.1 shows the relation between the number of crosspoint switches "S" in a full crossbar that interconnects "N" FPGAs each with "P" I/O pins.

$$S = N(N-1)P^2/2 \tag{2.1}$$

For example, to interconnect 20 FPGAs (note that the number of FPGAs in a typical FBE system is far more than this), each with 200 I/O pins, we need a FPID module with 4000 I/O pins and a switch capacity of 7,600,000. Manufacturing such FPID would be impractical and expensive in terms of pin count and layout area.

C Partial and Hierarchical Partial Crossbar The partial crossbar architecture [65][42] overcomes the limitations of the full crossbar by using a set of smaller crossbars. This is due to the fact that in real designs only a tiny fraction of crosspoint switches would ever be used to route signals in the system. In this architecture the I/O pins of each FPGA are divided into subsets and each subset

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

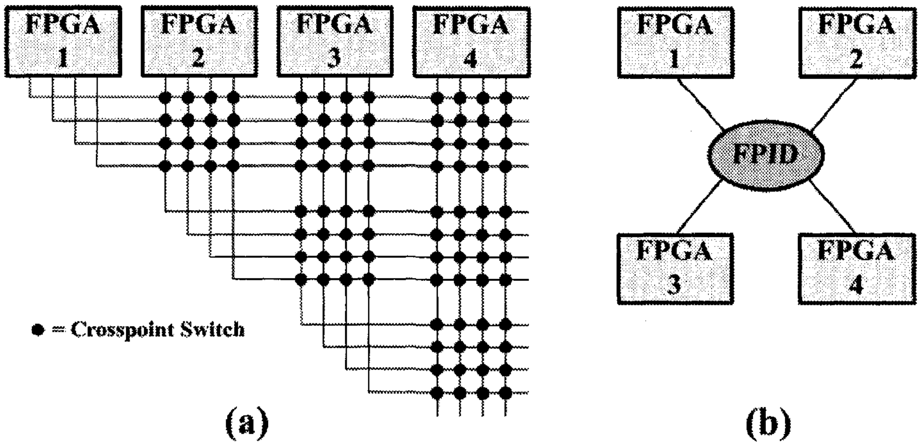


Figure 2.11: Logical view of full crossbar interconnect (a). Block view (b).

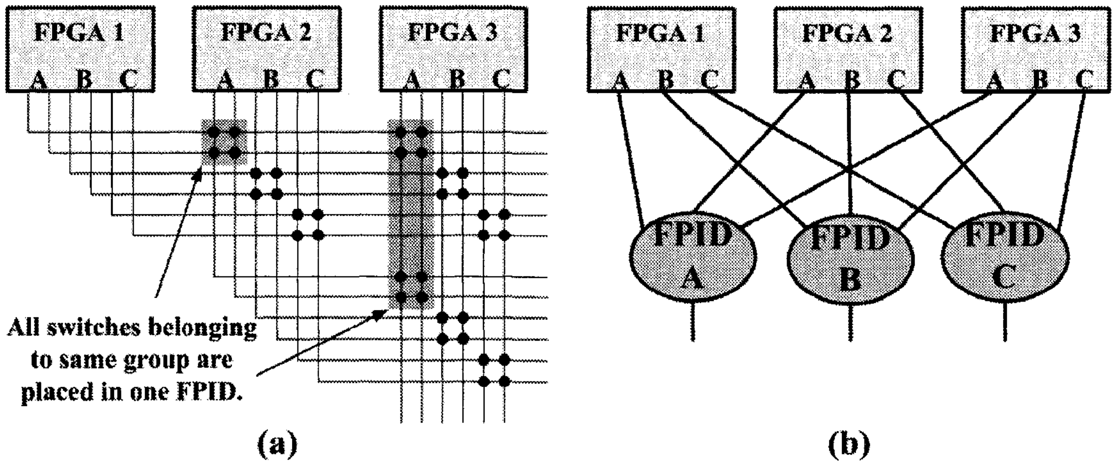


Figure 2.12: Logical view of partial crossbar interconnect (a). Block view (b).

is connected to a single FPID. Therefore the number of FPIDs in partial crossbar architecture is equal to the number of subsets (Fig. 2.12).

Partial crossbar architecture maximizes the use of the FPGA's logic capacity. The delay for any inter-FPGA connection is uniform and is equal to delay through one FPID. In this architecture, the size of FPIDs increases only linearly as a fraction of the number of FPGAs. Also, since this architecture is completely symmetrical, the mapping CAD tools can map a DUT into FBE in less time. Consequently, the partial crossbar interconnect is economical and fully scalable. However, it has some disadvantages too. First is the extra cost and size of multiple FPIDs. And second, the fact that direct connections between FPGAs for routing time critical signals are not available.

Large FBE systems (with hundreds of FPGAs) can not be interconnected through single layer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

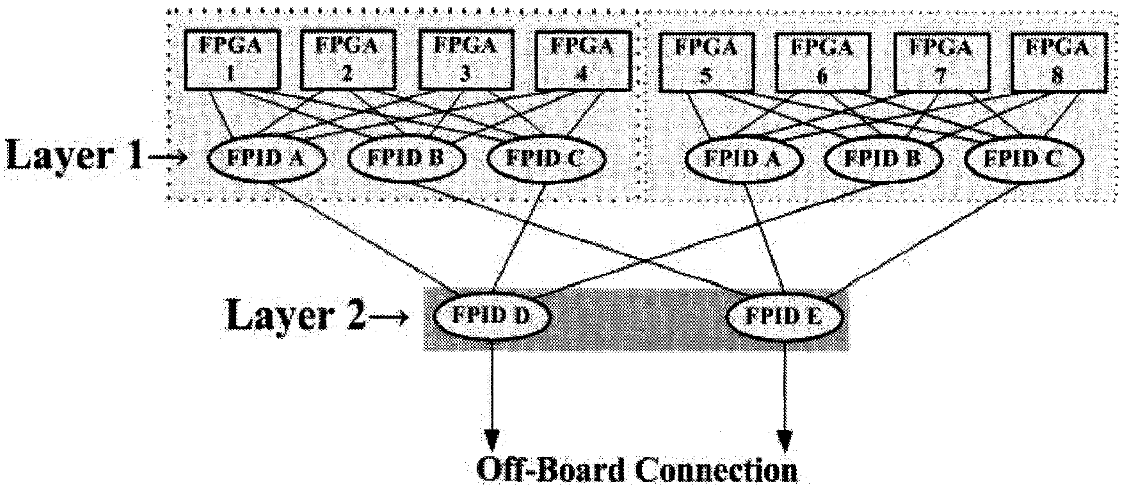


Figure 2.13: Example of two-level hierarchical partial crossbar architecture.

of partial crossbar. Instead, the partial crossbar architecture can be applied recursively, in a hierarchical manner. That is, each set of FPGAs and FPIDs, interconnected through partial crossbar architecture, could be virtually considered as a very large FPGA. A group of such "ultra-FPGAs" can be interconnected by a second level of FPIDs, as shown in Fig. 2.13.

In the example shown in Fig. 2.13, if there is a net that must be routed from "FPGA 2" to "FPGA 7", then that signal should pass through two FPIDs at "Layer 1" and one FPID at "Layer 2", imposing a total of 3 unit delays on that signal. This implies that the more hierarchy levels are used for interconnection, more delays would be induced on the nets. But this delay is acceptable because the size of flat partial crossbar cannot be scaled beyond a few tens of FPGAs.

D Hybrid Complete Graph Partial Crossbar The latest research shows that a mixture of hardwired and programmable connections among FPGAs provides a superior routing architecture for FBE systems. In this approach, a significant percentage of pins in each FPGA are connected using hardwired, the remainder are connected using programmable connections. The hardwire connections are usually used to route time critical nets, whereas other non-critical nets are routed through FPIDs (Fig. 2.14).

In *hybrid complete graph partial crossbar*(HCGP) architecture, the key parameter, which affects the degree of routability, is the percentage of programmable connections P_p with respect to the total number of interconnection (eq. 2.2-2.4). Results show that the ratio of 60 percent provides good routability and speed [42].

$$N_t = N_p + N_h \tag{2.2}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

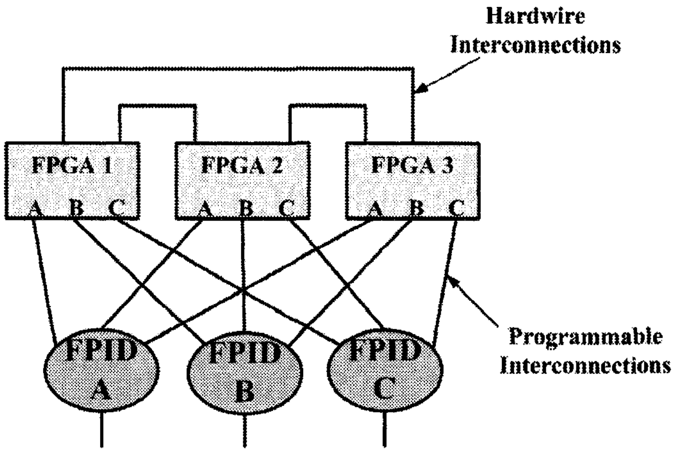


Figure 2.14: Hybrid complete graph partial crossbar architecture

$$P_p = N_p/N_t \tag{2.3}$$

$$P_p \approx 0.6 \tag{2.4}$$

Where,

N_p :Number of programmable connections

N_h :Number of hardwired connections

N_t :Total Number of Connections

E Virtual Wire Architecture The *logic capacity* (determined by the number of logic elements) of even the high end FPGA chips is not large enough to emulate even medium size digital IC designs. Hence, FPGA-based logic emulators must contain multiple FPGAs (tens to hundreds) so that they could emulate multi-million gate logic circuits. Obviously, for such circuits, the design netlist must be broken down in to smaller sub-circuits so that each sub-circuit could fit into single FPGA. The process of breaking down a circuit netlist into smaller sub-circuits is referred to as *partitioning.* Similarly, each sub-circuit is called a *partition.* After the circuit netlist is partitioned and mapped into FPGAs, they will be connected to each other through FPGA I/O pins. For each I/O signal belonging to a partition, one I/O pin will be utilized (Fig. 2.15). Since FPGAs have limited number of I/O pins, the sum of inputs and outputs of each partition can not exceed the number I/O pins in one FPGA. Therefore, while partitioning a circuit amongst multiple FPGAs, each partition should satisfy two constraints:

1. Logic capacity constraint:

   Number of logic elements in one partition$\leq$ (Total number of logic elements in one FPGA)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

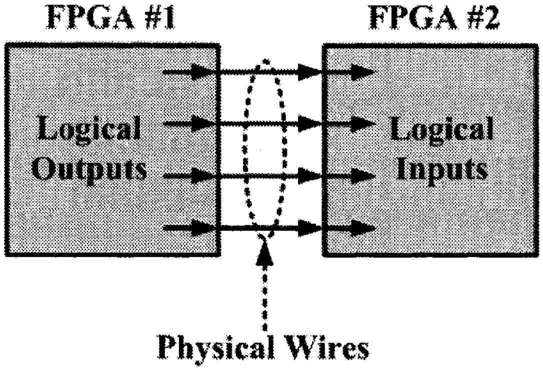


Figure 2.15: A genric view of non-time-multiplexed signals among two partitions.

2. Pin constraint:

   $N_i + N_o \leq P_t$ where,

   N_i :Number of Input signals to partition

   N_o :Number of Output signals from partition

   P_t :Total number of FPGA I/O pins

In a paper by Landman and Russo [46], it was empirically shown that the number of I/O pins in a partition is a function of number of logic elements in that partition. Such relation is shown in 2.5 and it is referred to as *"Rent's rule"*.

$$P_t = \bar{k} \times L^R \tag{2.5}$$

where,

L :Total number of logic elements

R :Rent's constant $(0.4 \leq R \leq 0.8)$

$\bar{k}$:average fan-in of logic elements

Empirical results show that, due to Rent's rule, a great percentage of FPGA logic capacities in conventional FBEs will remain underutilized. In worst cases it could be as high as 80%.

To overcome pin limitations (expressed by Rent's rule) and improve logic utilization in FPGAs, researchers at MIT proposed the idea of *Virtual Wires* [2]. Unlike traditional architectures where each interconnecting physical wire is assigned to one signal (net), in virtual wire architecture each physical wire will transfer multiple signal values at different time slots. In other words multiple signals will be "time-multiplexed" on the same physical wire (Fig. 2.16). Multiple "output" signals can be sampled and stored inside "micro-registers" at the "source" partition. The content of these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

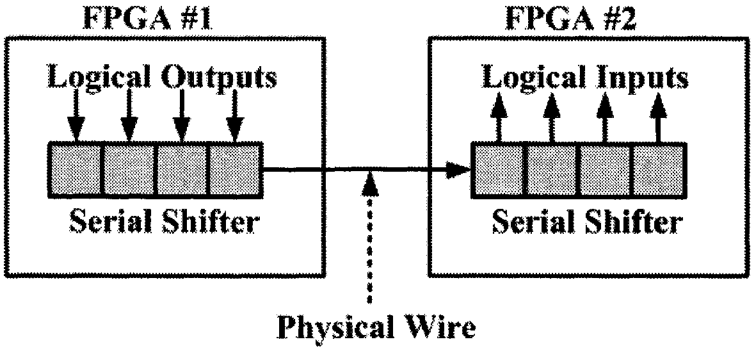


Figure 2.16: Generic view of Virtual Wire architecture

registers are then serially transferred to the "destination" partition. A single wire is used to transfer the serial values from the "source" partition into the "destination" partition. At the "destination" partition the signal values are De-multiplexed using a set of serial receivers and a serial-to-parallel converters. It should be mentioned that the sampling and transmission of signal values takes place during each design's clock cycle.

Virtual wire-based architecture has a number of advantages over other architectures such as:

- It significantly improves logic utilization in FPGAs (some cases more than 45%).
- Overcomes I/O pin constraints.
- Significantly reduces the number of FPGAs required in the FBE systems. Therefore virtual wire-based emulators are much smaller and cheaper.

On the other hand virtual wire-based emulators have a number of disadvantages too:

- Extra control circuitry inside each FPGA is needed to time multiplex/de-multiplex signals on a shared wire which imposes logic overhead in the circuit.
- Transferring signal values in time slots will cause delay in the signals. Therefore, emulation speed is reduced.

F Time-Multiplexed FPGA Architecture In a different approach to improve logic utilization in FPGAs, researchers have proposed a dynamically reconfigurable FPGA called *time-multiplexed FPGA* [64]. At any instance of time, a time-multiplexed FPGA has one "active" configuration and eight "inactive" configurations. The *configuration memory* (also referred to as configuration memory *plane*) is distributed over all logic elements and interconnecting switches within the FPGA

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

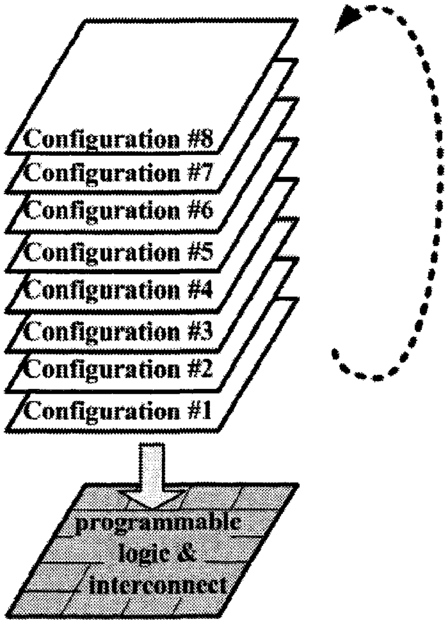


Figure 2.17: Time-multiplexed FPGA configuration model.

chip which might contain 100,000 memory cells. Each configuration memory cell is backed up by eight inactive configuration memory cells. Whenever the FPGA is reconfigured, all the logic elements and interconnecting switches are updated simultaneously through the contents of one configuration memory plane (Fig. 2.17). In practice, inactive configuration bit-streams might be stored in off-chip memory banks which increases the FPGA reconfiguration delay.

After each and every reconfiguration, the output of each logic element inside the FPGA is also stored in memory arrays called *micro-registers*. With 8 configuration planes, a micro-register should contain an array of 8×1 memory cells. A general structure of a logic element in a time-multiplexed FPGA is shown in Fig. 2.18.

In logic emulation mode, the time multiplexing capability of the FPGA is used to emulate a large design. The FPGA sequences through all configurations called *micro-cycles*. Partial results after each micro-cycle (i. e. after one configuration of the device) will be saved in micro-registers and passed to subsequent micro-cycles. One pass though all micro-cycles is equivalent to one DUT clock cycle (also known as *user cycle*).

2.2.1.3 Emulating Logic Designs on FBEs

So far we have explained different architectures used in the programmable hardware section of FBEs. Now we explain how a typical digital design can be emulated on a generic FBE. To emulate a logic

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

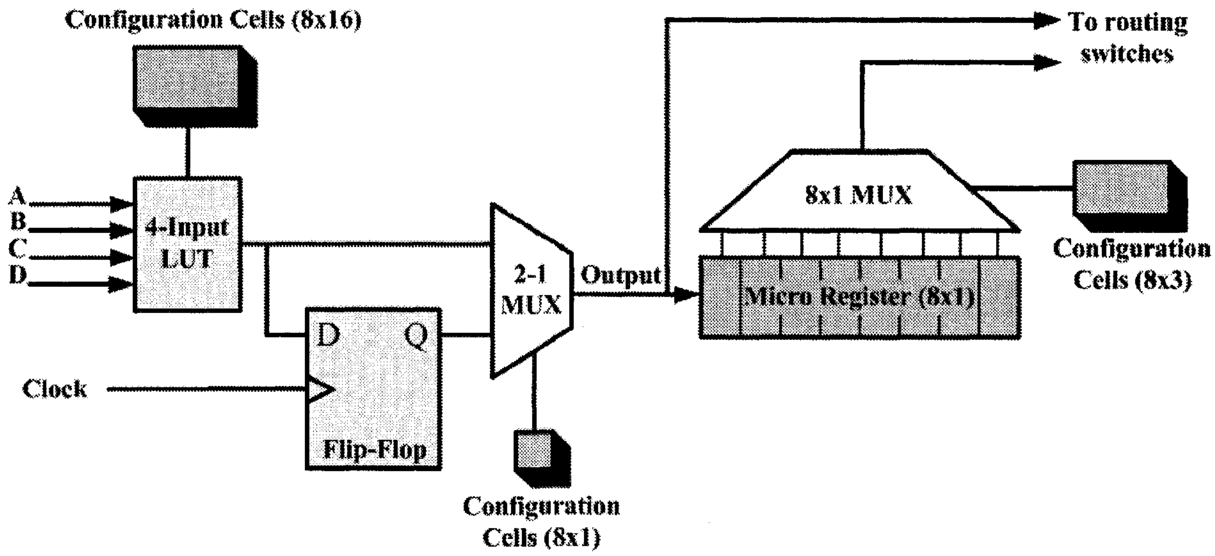


Figure 2.18: General view of one logic element in a time-multiplexed FPGA.

design on an FBE, first, the mapping CAD tools translate the design netlist into a set of configuration bit-streams that can be used to configure (i. e. program) the FPGAs and FPIDs. Then, programming bit streams are downloaded into all FPGAs and FPIDs. Once the FBE is configured it is ready to emulate the design. Through a set of run-time tools, designers can examine their designs and detect possible errors. We will explain the details of the steps involved in future sections.

2.2.2 Processor-Based Logic Emulation System (PBE)

The second class of logic emulators are *Processor-Based Emulator Systems* (PBEs)[70]. First generations of PBEs were introduced to the industry much before FBEs but they were only capable of performing simulation acceleration and not in-circuit emulation. After the invention of FPGAs, most companies preferred using FBEs for design verification. However, shortly later on, due to obvious disadvantages of FBEs as well as introduction of custom IC design, PBEs were brought back into spotlight. As of mid 90's (until now) major verification vendors have introduced large-scale high-end PBE systems to the market[24].

A general misconception does exist among few engineers that needs to be addressed here. Some people believe that PBE systems are just another kind of hardware-accelerated simulation engines which is not correct. Here are some fundamental differences between PBEs and hardware-accelerated simulators:

- PBEs contain a collection of application specific processors , called emulation processors,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which are optimized for emulating the functionality of logic circuits, as opposed to hardware-accelerated machines in which generic processors are utilized.

- Hardware-accelerated simulators use software models of DUT components to simulate the functionality of the whole design, whereas, in PBEs, the DUT netlist is directly mapped into hardware.
- Hardware-accelerated simulators can not be connected to target platform and their output appears, usually, in form of signal waveforms or data files, monitored on workstation screens, whereas, PBEs can actually be connected to the target hardware.

As it will be explained in forthcoming chapters, this research has introduced an easily implementable architecture for certain class of PBEs which has in fact created the required hardware platform for developing software CAD tools. But, before explaining the proposed architecture, we will investigate the generic architectures used in PBEs in this section.

2.2.2.1 Architecture of PBEs

In PBEs a collection of highly parallel hardware processors (e. g. tens to hundreds) are used to emulate the functional behavior of logic designs. The processors communicate with each other during run-time though an *interconnection network.* Depending on the *logic processors'* architecture, PBE systems could be very simple in structure or very complicated. However, roughly speaking, PBEs can be classified in two categories:

1. PBEs with Homogeneous Architecture
2. PBEs with Heterogeneous Architecture

A PBEs with Homogeneous Architecture In this architecture all logic processors (also known as *emulation processors*) are identical in architecture (Fig. 2.19). Conventionally, each logic processor is dedicated to emulating the functionality of a single gate in the DUT. However, because the processors are built using fast technologies, it is possible to use one processor to emulate multiple gates at different time slots. The control processor works as a bridge between the host processor and the emulation hardware. The I/O processor establishes *in-circuit* connection between the emulation system and the target hardware. During the emulation process, logic processors transfer signal values and other information to each other.

Various emulation systems used in industry are developed based on the homogeneous architecture. Examples of such systems can be found in [29].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

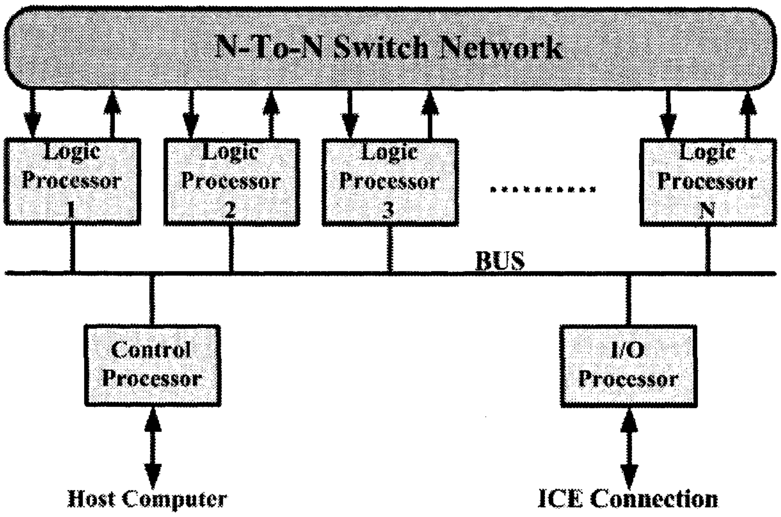


Figure 2.19: General view of a Homogeneous PBE system

B PBEs with Heterogeneous Architecture Unlike homogeneous architectures, *heterogeneous* PBEs consist of a collection of non-identical processors (Fig. 2.20). Instead, each processor is optimized to emulate specific tasks or functions[12]. For instance, some processors could be optimized for performing arithmetic operations such as multiplication/devision while another processor could emulate memory operations.

2.2.3 Logic Emulation Systems in Industry

We conclude this section by presenting examples of emulation systems used in industry that are currently helping design engineers to perform functional verification at early stages of IC design.

An example of commercially available FBE system is *VStationPRO* from *Mentor Graphics*$^{\text{TM}}$[22]. It is based on the virtual wires architecture that can emulate designs consisting of up to 120 million logic gates, at speeds ranging from 0.5-2MHz. *Palladium* system from *Cadence*$^{\text{TM}}$[24][38] is an example of a processor-based logic emulation system. It has a logic capacity of up to 256 million gates and emulation speeds ranging from 0.5 to 1MHz. It is not only a logic emulation system but also can be configured to function as a simulation acceleration platform for various design applications, offering simulation speed of 10000 times faster than software-based simulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

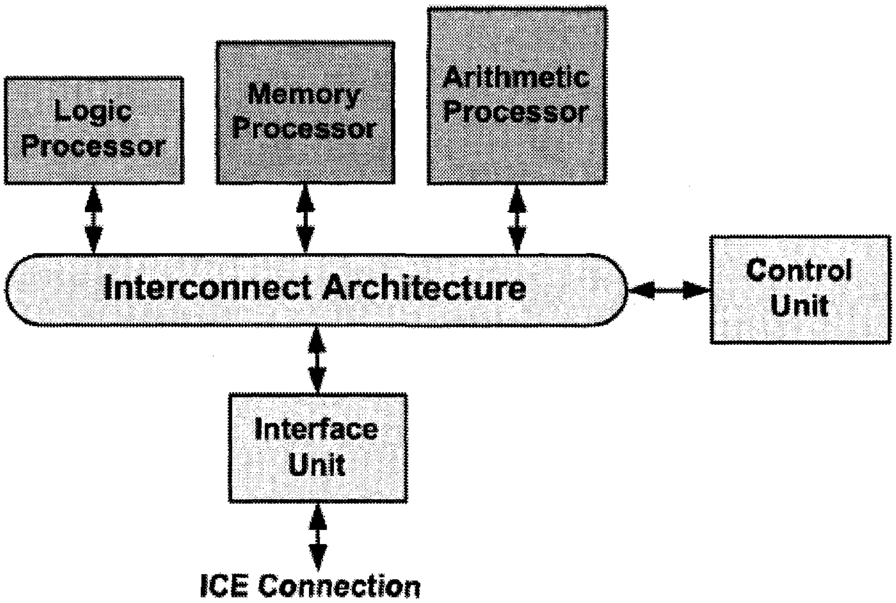


Figure 2.20: General view of a Heterogeneous PBE system

2.3 CAD Flow for Logic Emulation Systems

So far we have discussed different hardware architectures used in logic emulation systems. However, it goes without saying that without a useful computer-aided design (CAD) tool set, an emulation system would be a completely useless piece of hardware. In this section, we briefly review design mapping CAD tools used in logic emulation systems discussed so far to familiarize readers with basic ideas involved in designing CAD tools for a logic emulation systems.

2.3.1 Introduction

Recall from 2.2 that logic emulation systems are usually connected to a host workstation on which CAD tools are run. Generally speaking, an emulation CAD tool is responsible for two major tasks:

1. Mapping a logic design (DUT) into the logic emulation hardware, and
2. Controlling and supervising the operation of logic emulator during run-time.

Consequently, CAD tools for logic emulation systems consist of two major parts: design compiler and run-time support tools. The run-time support tools are a collection of different front-end software tools (such as graphical logic analyzer, waveform viewer, memory analyzer and etc.), which help the users in debugging DUT easily and efficiently during the emulation process. The run-time support

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tools may differ significantly from one manufacturer to another or even from one product to another. Due to such high degree of architectural dependency, the run-time support tool will not be out of the scope of this research. The main focus in this section will be the design compilation CAD tools.

By definition, an emulation design compiler is a *complex CAD system that efficiently translates huge structural representations of the design-under-test into the target emulator architecture.* In other words, the design compiler software takes the netlist of the DUT and translates it in a way that it could be mapped into the target emulation system, so that the functionality of the mapped netlist would accurately imitate the functionality of the original design. Given the fact that today's medium size designs would contain hundreds of thousand logic gates, the most important agenda would be the speed and accuracy of the design compiler CAD tool. Obviously, a well designed emulation CAD tool would be the one that translate the DUT netlist to the target emulation system more efficiently in less time.

The main focus of this section of thesis is to introduce an efficient set of tools that can take a large design netlist and map it to the proposed HEP-based logic emulation engine. But before that, we are going to briefly review the contributions made so far by other researchers in the field.

2.3.2 CAD Flow for FBEs

At first, we will be examining the CAD tool flow of FPGA-based logic emulation devices (FBEs). To map a logic design into an FBE, the design netlist has to pass through several steps of design compilation shown in Fig.2.21. The followings explain each step in further details:

- ***Design Entry***: The first step is design entry, where the compiler accepts input design file(s) specified in hardware description languages (HDLs), schematic netlists or any other proprietary design entry tool. At the end a "raw" design netlist will be generated by the design entry tool.
- ***Synthesis***: Design compilation begins by reading in the design file(s) and generating the gate level logic netlist, which involves the transformation of register-transfer level (RTL) specifications to gate level netlist [37][18]. This process usually results in a large hierarchical collection of netlists. The compiler combines them into a non-hierarchical single-level (flattened) design netlist file. If the design files are utilizing ASIC (Application-Specific Integrated Circuit) or cell libraries, the design compiler expands the library elements to the fully primitive level. At the end of this stage, a large flattened gate-level netlist of the design-under-test is generated. Also, at this stage nets which have to be connected to in-circuit cable pins, logic analyzer or pattern generator channels are identified and marked by the user.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- ***Technology Mapping***: At this stage the technology mapping tools translate logic primitives in the design file into FPGA's *logic elements* [17][31]. For instance, if the FPGA's logic elements only support 4-input LUTs then those logic gates inside the netlist with fan-in degree greater than four are broken down into smaller logic gates supportable by FPGA's logic element architecture. Similarly, small logic gates with fan-in degree less than 4 will be grouped together so that they could fit into logic elements. Also, technology mapping can automatically generate the FPGA logic block to emulate particular memory configuration in the design netlist.

- ***Partitioning***: Next, the huge gate-level netlist has to be broken down into smaller chunks of logic netlists so that each chunk could fit into one FPGA chip on the emulation board. This step is essentially needed for those FBEs which contain multiple low-capacity FPGA chips[2]. This process is referred to as *spatial partitioning*, or simply, *partitioning*. The partitions are evaluated and optimized according to different criteria like FPGA logic capacity (size), number of I/O pins on FPGAs and timing/speed constraints. The goal of partitioning is to minimize the number of utilized FPGAs, while observing the above constraints.

  Almost all partitioners will take "multi-level-multi-way" partitioning approach to perform partitioning on the design netlist. Through this process, first, the design netlist is partitioned into a number of logic modules (LMs) that are usually equal to the number of boards available in the emulator. Then each LM is partitioned into minimum possible number of FPGA chips. To perform multi-level-multi-way partitioning, two classes of solutions have been introduced: top-down techniques and bottom-up techniques. Two algorithms, *min-cut* [30][36] and *ratio-cut* [68], belong to the top-down category that cut the whole design netlist recursively into smaller and smaller partitions. Clustering techniques are used for bottom-up approach through which partitions are built up out of tightly interconnected logic primitives [19][52]. Commercial partitioning tools use combination of both techniques alternatively to build the partitions. Once the partitions are created, each partition is assigned to a single FPGA in the FBE.

  On the other hand, those FBE systems in which time-multiplexed FPGAs or virtual wire technology is used, hardware resources (such as FPGA logic elements or I/O pins) are shared over time. In such systems, the DUT netlist has to be partitioned not only spatially but also *temporally*. The temporal partitioning algorithms perform the partitioning operation on the netlist so that delay overhead of sharing resources is minimized. In virtual wire-based emulation systems, where FPGA I/O pins are shared throughout time, the temporal partitioning

---

[2] Such systems are also referred to as Multi-FPGA Systems (MFS)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithms bundle up source-sink pairs in the netlist and assign unique time slot to each signal value. The algorithms try to minimize the time delay in all signals to obtain greater emulation speed [62]. In time-multiplexed FPGA-based emulation systems, the temporal partitioning algorithm will partition a technology-mapped netlist based on the precedence of logic elements in netlist, so that those closest to the input signals are emulated earliest and those closest to outputs are emulated last. The algorithms guarantee that no signal is emulated earlier than its *fan-in* signals while keeping the number of FPGA reconfiguration minimum [63].

- ***Board Level Placement***: Once the design is partitioned each partition must be assigned to an FPGA among numerous FPGAs on the emulation hardware. The complexity of this step is totally dependent on the interconnection architecture employed in the emulation hardware. For instance, those emulators in which partial crossbar architecture is used, the interconnection architecture is totally symmetrical. Consequently, any random board level placement is acceptable. However, when the mesh architecture is used, placement becomes highly critical for maintaining the inter-FPGA connections as short as possible.

  The most well-known placement algorithm is *simulated annealing* [43][58] which imitates the annealing process in molten metals. Starting with a high-temperature the simulated annealing algorithm generates a number of random placements of partitions among multiple FPGAs. As long as the new placements decrease the cost function(s) (i. e. routing cost, delay) the new placements would be accepted as valid placements. If the new placements increase the cost function the algorithm still accepts them, but in a probabilistic manner. If the new temperature gets below the "threshold temperature" then the algorithm will stop and will accept the last placement configuration which generated the least cost value. This way the algorithm avoids getting trapped in the local minima. It is worth emphasizing that, just like partitioning, there are no optimum solutions for placement problem achievable in polynomial time.

- ***Inter-FPGA Routing***: The inter-FPGA router determines the routing path for each inter-FPGA net. The router could use direct connections between each FPGA pairs or it may use intermediate FPGAs and FPIDs, depending upon the routing architecture used and wire availability. The router tries to avoid or minimize the number of intermediate FPGAs/FPIDs used so that usage of routing resources as well as delay is minimized. It also tries to balance the usage of routing resources to ensure routing completion.

- ***Intra-FPGA Placement and Routing***: At the next step, the compiler has to place each logic partition into the assigned FPGA and perform routing of internal nets using internal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

routing resources of FPGA chips. The placement and routing tools for this purpose are usually provided by FPGA vendors and may vary significantly depending on the internal architecture of FPGAs [13]. However, the following four steps are common among all of them:

- Assigning each logic block in design netlist to a specific logic block in the target FPGA (placement). The goal is to minimize total wiring length and critical path delays.
- Various FPGA placement algorithms have been proposed such as [50][47][5] [49].
- Finding topological path of wires of each net in the chip. This process is referred to as *global routing.* Global routing is performed based on graph search techniques guided by channel or switch block density [9][13][6].
- Defining routing regions by breaking the areas around FPGA logic elements into channels and switch boxes. Performing *detailed routing* (also known as *channel routing*) for each region, one region at a time [9][6].

In most algorithms mentioned above the main objectives are reducing wiring length as well as reducing signal delays in the mapped netlist.

- Configuration Bit-Stream Generation: The last step in the design compilation flow is the generation of the configuration bit stream for each FPGA which would be eventually downloaded into FPGAs.

Once the configuration bit-stream is downloaded into the FBE hardware, the DUT is ready to be emulated.

It is worth emphasizing that, despite the fact that the CAD flow is presented sequentially, in the real world, CAD tools might iterate several times through different steps to obtain near optimal results. Also, for the sake of simplicity, some intermediate steps such as *design rule checking* (DRC) and *clock tree analysis* are not illustrated here. Commercial CAD tools might run the CAD tool on multi-processor platforms to reduce the compilation time.

Most importantly, partitioning placement and routing are well known examples of *NP-hard* problems, for which there are no algorithms available that can produce optimal results in polynomial time [59]. Instead heuristic techniques are used, which usually provide acceptable near-optimal solutions within a reasonable amount of time. However, the design compilation time is quite dependent on the size of design netlist. Consequently, in comparison with PBE CAD tools, design compilation under FBE CAD tools is relatively more time consuming and less predictable[3].

---

[3] Hence, it takes significantly more time to make "what-if" changes in DUT, if it were emulated using FBE.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

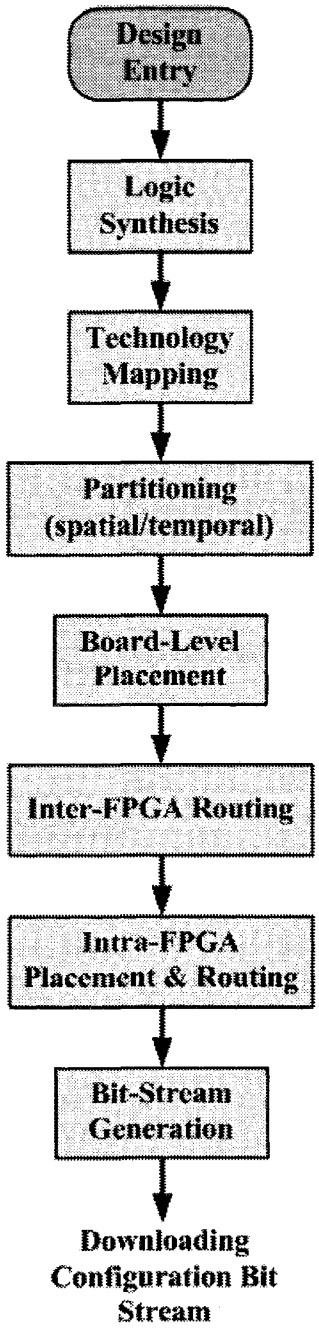


Figure 2.21: CAD flow for FBEs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3.3 CAD Flow for PBEs

As it was mentioned earlier, a typical PBE system contains a collection of highly parallel processors that, together, they emulate the functionality of DUT. Just like FBE systems, PBEs should be accompanied by a set of CAD tools that automatically "translate" the DUT's netlist into PBE hardware for emulation purpose. PBE's CAD tools take the design netlist through a series of steps to compile. At the end of compilation a set of executable binary codes will be generated for each emulation processor in the target PBE hardware. Once executable codes are generated, they will be downloaded into the "program memory" associated with each processor. Each processor will execute a unique emulation program.

The design compilation flow for PBE systems is similar to that of FBE system, with some minor differences. In fact, the algorithms involved in design compilation for PBE systems are relatively simpler and less complicated. A typical design compilation flow for PBEs is shown in Fig. 2.22. The detail of activities at each step is as follows:

- Design Entry and Synthesis: these two steps are more or less identical to those in FBE CAD tool. At the design entry step, the compiler accepts input design file(s) specified in hardware description languages (HDLs), schematic netlists or any other proprietary design entry tool. The synthesis tool will generate a large flattened gate-level netlist of the design-under-test.

- Technology Mapping: Next, the gate-level netlist is mapped into logic primitives which are recognizable by the emulation processors architecture. Hence the result of this step may vary significantly from one PBE to another.

- Spatial and temporal partitioning: At this stage, DUT netlist is divided into smaller sections so that once an emulation program is generated for each partition, the program could fit into the "control memory" of the associated emulation processor. The PBE partitioning tool will perform the partitioning process based on the processing capacity of each emulation processor within the network, or in other words processor's *granularity*[4]. The partitions are then temporally arranged based on their *precedence* in the circuit. Such process may also be referred to as *scheduling*. Temporal partitioning tools determines the sequence of execution for each emulation program. The objective of scheduling algorithm is to balance the processors' workload by evenly distributing tasks among different processors and maximizing emulation speed by profiling inter-processor connection.

---

[4] As opposed to PBEs, in FBE CAD tools the main constraint for partitioning is FPGA logic capacity versus available I/O pins while minimizing delay.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Emulation Program Generation: The last stage is to generate the instruction words (i. e. opcode) for each processor. The instructions will be eventually downloaded to the control memories of processors. After downloading the control programs the emulation hardware is ready to emulate the DUT.

Few notes about the CAD tool flow mentioned above would be in order: First, it should be emphasized that design compilation steps listed above may not appear in all PBE systems because these systems are quite diverse with respect to their architecture. In some cases more/less steps for design compilation might be needed. Second, technology mapping tools in PBEs might be very complex based on the granularity of emulation processor. For example in, heterogeneous architectures (see B) the technology mapping tool has to be able to automatically identify functionality of each submodule (such as adders/multipliers, memories, counters/shift registers etc.) in the netlist and then assign/map each submodule to its corresponding emulation processor. Such capability might require technology mapping tools to contain comprehensive set of libraries for all functional submodules or have profiling capabilities to identify each submodule in the DUT's netlist. Obviously, this increases the complexity of CAD tool quite extensively. Examples of such tools can be found in [29] although the authors have not explained details of their CAD tools. Third, in somc cases the order of spatial and temporal partitioning might be reversed where seemed appropriate. Based on the above facts, it is evident that in order to prove the efficiency of the proposed HEP-based logic emulation engine, we need to introduce the accompanying set of CAD tools that automatically translate the DUT netlist to the target emulation engine. In the next chapter of this thesis we are going to introduce the proposed set of tools as well as their sequential flow.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

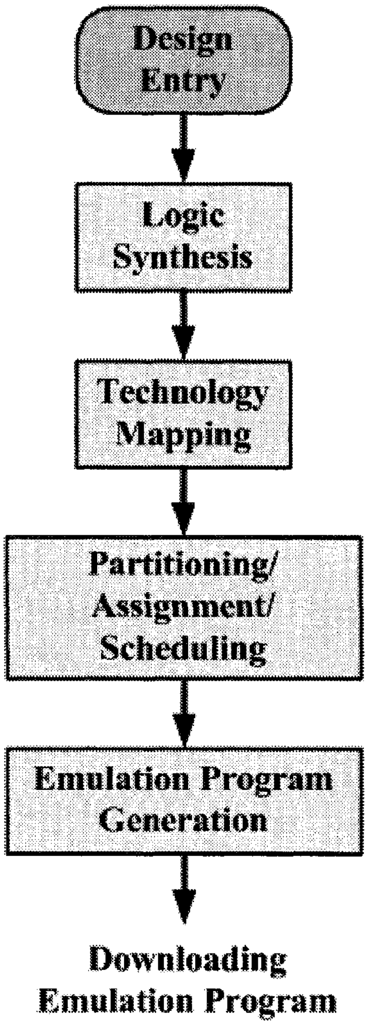

Design Entry
Logic Synthesis
Technology Mapping
Partitioning/ Assignment/ Scheduling
Emulation Program Generation
Downloading Emulation Program

Figure 2.22: CAD Flow for PBEs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Architecture of Hybrid Emulation Processor (HEP)

This chapter introduces a new class of processor-based logic emulations systems (PBE) that are easily implementable in FPGAs. The new emulation system is referred to as *hybrid emulation system.* The basic building blocks of the proposed architecture are *Hybrid Emulation Processors* (HEP) which is described in details in this chapter. The architecture of the HEP processor has few similarities with the architecture explained in [29]. However there are fundamental differences that will be explain when appropriate. The information presented in this chapter will also help readers to understand the software considerations for mapping CAD tools presented in future chapters.

3.1 Top-Level Organization the Emulation Engine

The proposed logic emulation system consists of an array of 64 identical processors referred to as *Hybrid Emulation Processor*(HEP). The processors can transmit and receive data through an *interconnection network.* All the processors execute their local program in *parallel.* A *global sequencer* (or *Program Counter*), whose value is shared by all 64 processors, causes the processors to step through their emulation program in synchronism. Such embodiment consisting of processors, interconnect network and global sequencer is called an *emulation module.* The block diagram of an emulation module is shown in Fig. 3.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

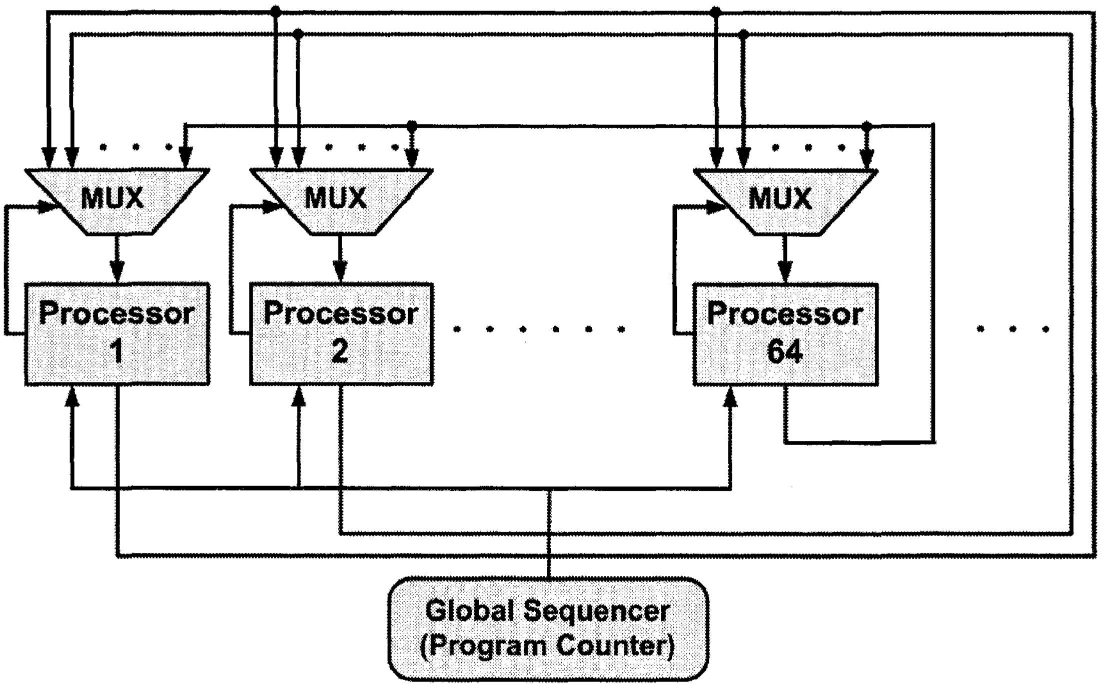


Figure 3.1: Block diagram of an emulation module

3.2 How a Logic Design is Emulated?

Before moving on to details about the internal architecture of the proposed emulation system, it is appropriate to give the big picture on how a typical logic design can be emulated on this engine.

Before emulation starts an emulation CAD tool translates, maps and partitions the design-under-test into logical clusters. For each cluster, a control program consisting of a set of control words is constructed for a specific emulation processor. Individual emulation control programs are then loaded into embedded control memory associated with each processor prior to emulation. During emulation, the emulation processors execute control words from their respective control programs in synchronism via step values provided by the program counter. A complete sequence of steps corresponds to traversing all logic paths starting from the inputs towards the outputs within the DUT. It should be emphasized that each processor executes its unique program to emulate its assigned logic cluster. Due to the fact that the logic within clusters should be able to interact with each other, therefore the processors need to have the ability to transmit and receive data to/from each other. The communication among the processors is provided through the non-blocking interconnection network consisting of sixty four 64-input multiplexers (MUX).

In the following sections the internal structure of each part in the emulation engine is described

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

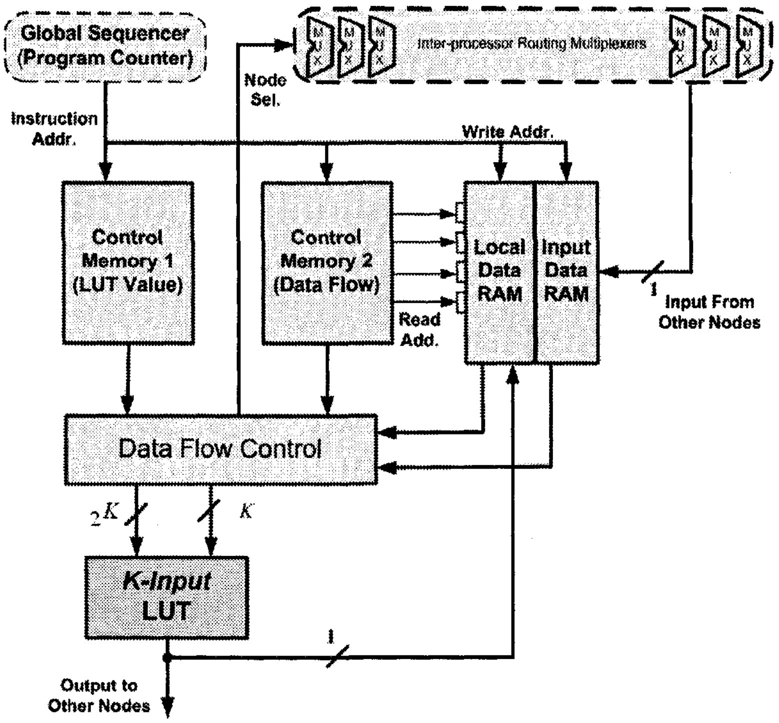


Figure 3.2: Internal structure of HEP Processor.

as well as their functionality.

3.3 Structure of Hybrid Emulation Processor

The emulation engine contains 64 identical HEP processors. The hybrid emulation processor (HEP) is a basic building block of the emulation engine. The internal structure of the processor is shown in Figure 3.2. At the heart of each processor there is a reconfigurable 4-input look-up table (LUT) that can implement any logic function of four inputs. A k-input LUT, can implement 2^{2^k} logic functions. Given the fact that in this architecture $k = 4$, HEP processor can implement any of 65536 possible logic functions at each step [1]. The processor's primary function is to execute 4-input logical function and produce a "function bit-out" during each step of the sequencer. Figure 3.3 exemplifies how the logic function (F) of four inputs (A,B,C and D) is implemented using a 4-input LUT. Presence of LUT in the emulation processor certainly enables the processor to emulate any combinational logic consisting of 1-4 inputs. On the other hand, to enable a processor to emulate sequential logic, two memory arrays are implemented to store logic values: *Local Data RAM*(LDR) and *Input Data*

---

[1] As opposed to [29] in which $k = 3$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

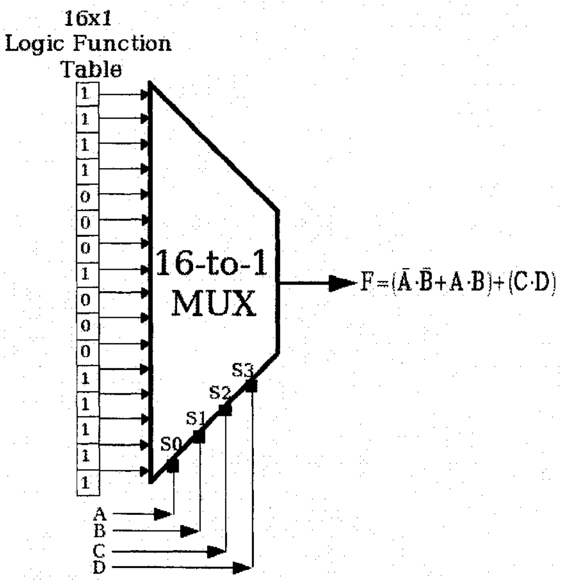


Figure 3.3: Example of implementing function F in a 4-input LUT.

RAM)(IDR). To implement a logic function, the "select" inputs to the 4-input LUT can receive any value from either of two memory arrays. Hence, an alternative to processors' logic function is a memory operation that stores/retrieves binary values to/from these memory arrays.

Embedding memory modules within each processor has created an architectural superiority over other emulation engines as well. Given the fact that, most of the today's logic circuits have some sort of built-in "memory", that stores binary information for processing (e. g. System-On-Chip modules have various memories, registers and buffers), embedded memory modules within each processor can be used to emulate various memory-related operations in DUT.

Each processor can produce one-bit output at each step. Based on the above scheme the resulting function bit out may correspond to:

- a combinatorial logic output corresponding to a combinatorial logic cluster in the DUT
- register output in the DUT
- single-bit value read from a cell in a memory array

Additional common operations performed by the processor during the emulation steps include storing the function bit out for subsequent use by the processor inside the Local Data Ram (LDR) and capturing and storing external (to the processor) data from other processors inside Input Data Ram(IDR).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each processor contains two sets of "program" memories referred to as *Right Control Memory* and *Left Control Memory.* The left and right control memories hold a unique program created by the emulation CAD tool for each processor. The LDR and IDR hold data previously generated and are addressed by fields in a corresponding right control word to locate (fetch) four binary bits for input to the LUT.

All the processors step through their program memories, while all share the value in the program counter (sequencer register). During each step of the sequencer an HEP processor emulates either a four input logic function, a memory array access or simply nothing (i. e. No-Operation) according to the instruction read from the program memories. Different fields in the left and right control words determine the type of operation as well as controlling the "data flow" within the processor.

3.4 Instruction Set Architecture of HEP

Unlike generic processors that usually have a large set of instructions, the HEP processor realizes only four instructions[2]. The combination of these four instructions constitute emulation programs which control the hardware emulation process on each HEP processor. As it is depicted in Figure 3.2 each instruction consists of two control words which are stored in Left and Right Control Memories respectively. The HEP instructions are:

1. **LUTOP**: Refers to "LUT Operation". The LUTOP instruction emulates a combinatorial logic functions of 1-4 inputs. Different fields of this instruction is shown in Fig. 3.4. The two most significant bits (MSB) (i. e. bits 17:16) of the left control word identifies the op-code (in this case ="01"). The remaining 16 bits in the left control word (i. e. bits 15:0) is the value which is loaded into the logic function table inside the 4-input LUT. The logic function is emulated by forming an address from four data bits retrieved from LDR and/or IDR. The location of these four bits inside the LDR and IDR memory spaces are specified in the right control word. Each address is 7 bits long which in Fig. 3.4 are labeled as "Operand Address A"(bits 6:0), "Operand Address B"(bits 7:13), "Operand Address C"(bits 14:20) and "Operand Address D"(bits 21:27). Four bits within the "source" field in right control word (bits 28:31) are used to configure the data path within the HEP processor to select between LDR and IDR as the source for fetching four operands. For instance, if bit 28 is "0" then operand "A" is fetched from LDR otherwise the value is fetched from IDR. The six bits of the "Node Address" in the right control word (bits 32:37) are used to select the single bit input to HEP processor from

---

[2] Instruction set of processors in [29] consists of only two instructions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

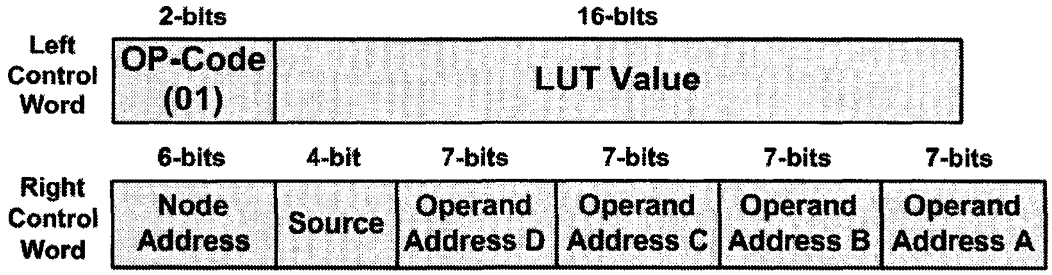


Figure 3.4: Fields of LUTOP instruction

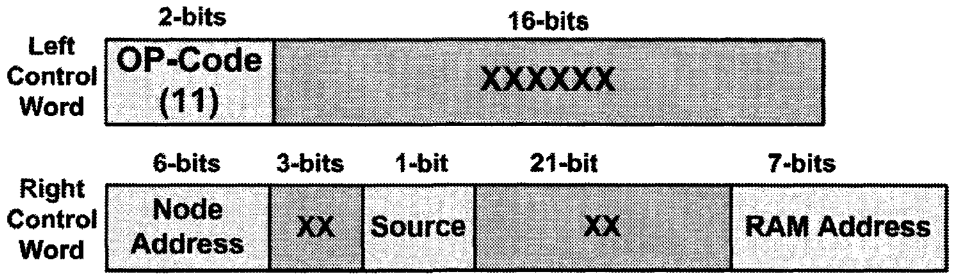


Figure 3.5: Fields of RAMREF instruction

any of the 64 processors in the emulation engine. This address is applied to the associated 64-input multiplexer (switch) to select a “bit-out” from one of the 64 processors in the engine. The selected processor bit-out is received as a processor bit-in signal and is stored in the IDR.

2. **RAMREF**: Refers to “RAM Referencing”. The RAMREF instruction performs a memory access operation on either LDR or IDR. The instruction will read single bit value from RAM memories and presents it as the processor's bit-out. Figure 3.5 shows different fields of this instruction. The two most significant bits in the left control word (bits 16:17) indicates the opcode (=“11”). The 7-bit address of the value that has to be fetched from LDR or IDR is presented in the least significant bits of the right control word (bits 0:6). A single “source” bit in the right control word (bit 28) specifies whether the value should be fetched from LDR or IDR (if the source bit =“1” then the value is fetched from IDR). The six most significant bits in the right control word (bits 32:37) specify the “Node Address” which was discussed in “LUTOP” instruction.

3. **ROMREF**: Refers to “ROM Referencing”. The ROMREF instruction reads one bit value from the “Right Control Memory” and presents it as the processor's output (i. e. bit-out). This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

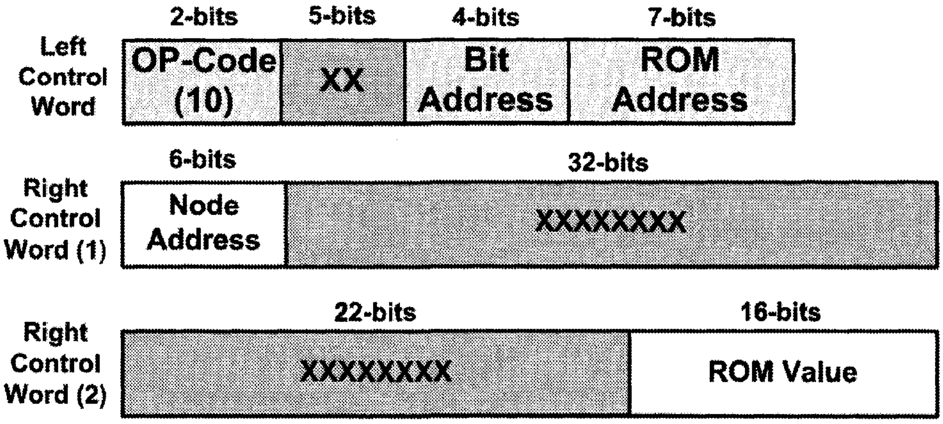


Figure 3.6: Fields of ROMREF instruction

instruction is mainly used when static binary values are needed for the emulation process. It is worth mentioning that, since the content of both Left and Right Control memories is loaded only once during the initialization of emulation engine, the binary values stored in these memories can be used to represent static data. Figure3.6 shows different fields of this instruction. The two most significant bits in the left control word (bits 16:17) represent the opcode (="10"). Seven least significant bits in the left control word (bits 0:6) contain the address of the location in the right control memory where the value must be read from. The value that is read from the right control memory is a 16-bit binary value from which only one bit is desirable. The 16-bit value fetched from the right control word is high-lighted in Figure3.6 as the lower 16 bits in the "Right Control Word(2)". To address a single bit among 16 bits, a 4-bit "bit-address" field in the left control word (bits 7:10) is used. Six most significant bits in the right control word (1) (bits 32:37) constitute the "Node Address" field. For further information about this field please refer to descriptions of LUTOP instruction.

4. **NOP**: Refers to "No-Operation". The NOP instruction does exactly what is says so: it does nothing at all. Such instruction causes the processor to slack (stall) for the duration of one instruction, during which it stores necessary data received from other processors. Such instruction is usually needed when one processor requires multiple inputs produced by other processors all the same time. In that case the processor should "wait" for other processors to produce the input values. Different fields of NOP instruction is shown in Figure 3.7. The two most significant bits in the left control word (bits 16:17) indicate the op-code value for this instruction (="00"). Six most significant bits of the right control word (bits 32:37) contain

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

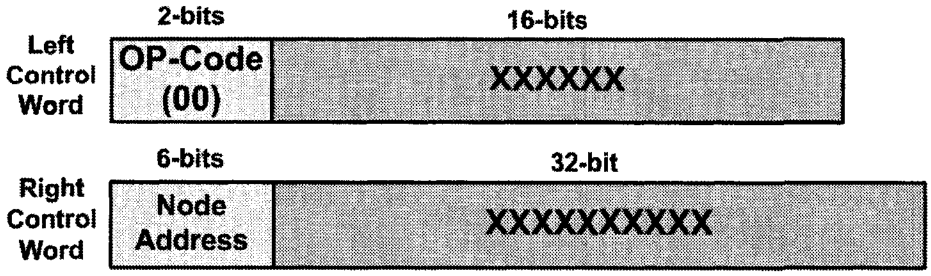


Figure 3.7: Fields of NOP instruction

the "Node Address" (see descriptions of "LUTOP" instruction for further details about "Node Address"). It is worth emphasizing that although the NOP instruction has no functional significance except for the fact that it still uses "Node Address" to select one of the 64 processors in the engine and latch the in-coming data from the selected processor.

It is worth emphasizing that an HEP processor, unlike other processors, does not recognize any type of "jump" or "conditional statement" instructions. The processor simply executes all the instructions one by one until it is halted by the *emulation supervisory unit.*

3.5 Central Control Unit of HEP

From the mathematical point of view a digital processor, in this case HEP, is a *Turing* machine with finite number of "states". Hence, all digital processors contain a central *control unit* that implements a *Finite State Machine*(FSM) that takes the processor, step-by-step, through a series of activities or states. Being no exception to this rule, the HEP processor contains a central control unit that traverses a finite state machine, symbolically shown in Figure 3.8. By traversing the FSM, the control unit supervises the flow of data inside the processor. In other words the FSM determines what kind of activities or events take place inside the processor during an *instruction cycle.*

Due to the fact that an HEP processor has only four types of instructions, the instruction cycle is less sophisticated than those in general purpose processors. In nutshell, during one instruction cycle, the processor fetches one instruction word from both Left and Right Control memories, where the "Program Counter Register" is pointing at. The instruction is then decoded and executed. The output produced by a processor is a single bit value which appears on the processor's "Node Bit-out" pin[3]. A copy of the output value is also stored in the Local Data RAM (LDR) memory within the processor for future references. The location where the output value is stored inside LDR is again

---

[3] The only exception to this rule is the "NOP" instruction which does not produce a new output.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

provided by the program counter register. Also, during each instruction, a processor will receive a single bit input from one of the sixty four processors inside the emulation engine. The received input is automatically stored inside *Input Data RAM*(IDR), where Program Counter points to.

In Figure3.8 each state has been assigned a unique two-digit state number which appears inside each state box.Details of activities taking place at each state of instruction cycle is explained below.

1. **State "00"**: This state initiates the fetching of instruction words from Left and Right Control Memories. The "Read" signals to both memories are asserted (active "High"). The address of the instruction is provided by the global sequencer (Program Counter Register) and is placed on the address bus. The control words read from both control memories are stored into processor's *Left Control Register* and *Right Control Register* respectively. Once the control words are read into the registers, the instruction is immediately decoded. Based on the type of the instruction, the control unit may jump to one of four possible states (i. e. "State 11", "State21", "State 31" or "State 41") in the next HEP clock cycle.

2. **State "11"**: By this state, the processor has identified (decoded) that the instruction to be executed is a LUTOP instruction. The six-bit "Node Address" is extracted from the right control word (bits 32:37) and applied to the 64-input MUX to select the single input bit to the processor among 64 inputs (see 3.4). The logic function table of the 4-input LUT is updated with a 16 bit value stored in the left control word. The location address of the first operand to the 4-input LUT is extracted from the right control word ("Operand Address A") and applied to the address busses of both LDR and IDR. The respective "Read" signals to LDR and IDR are asserted. Bit 28 of the left control word selects either LDR or IDR as the source for "Operand A". Consequently, at the end of this state the first input to the 4-input LUT is fetched from the memory.

3. **State "12"**: At this state the location address for the second input to the 4-input LUT (i. e. "Operand Address B") is extracted from left control word and placed on LDR and IDR address busses. Also, bit 29 of the left control word selects either LDR or IDR as the source for "Operand B". At the end of this state the value of "Operand B" is fetched from either of the memories.

4. **State "13"**: At this state the location address for the third input to the 4-input LUT (i. e. "Operand Address C") is extracted from left control word and placed on LDR and IDR address busses. Also, bit 30 of the left control word selects either LDR or IDR as the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

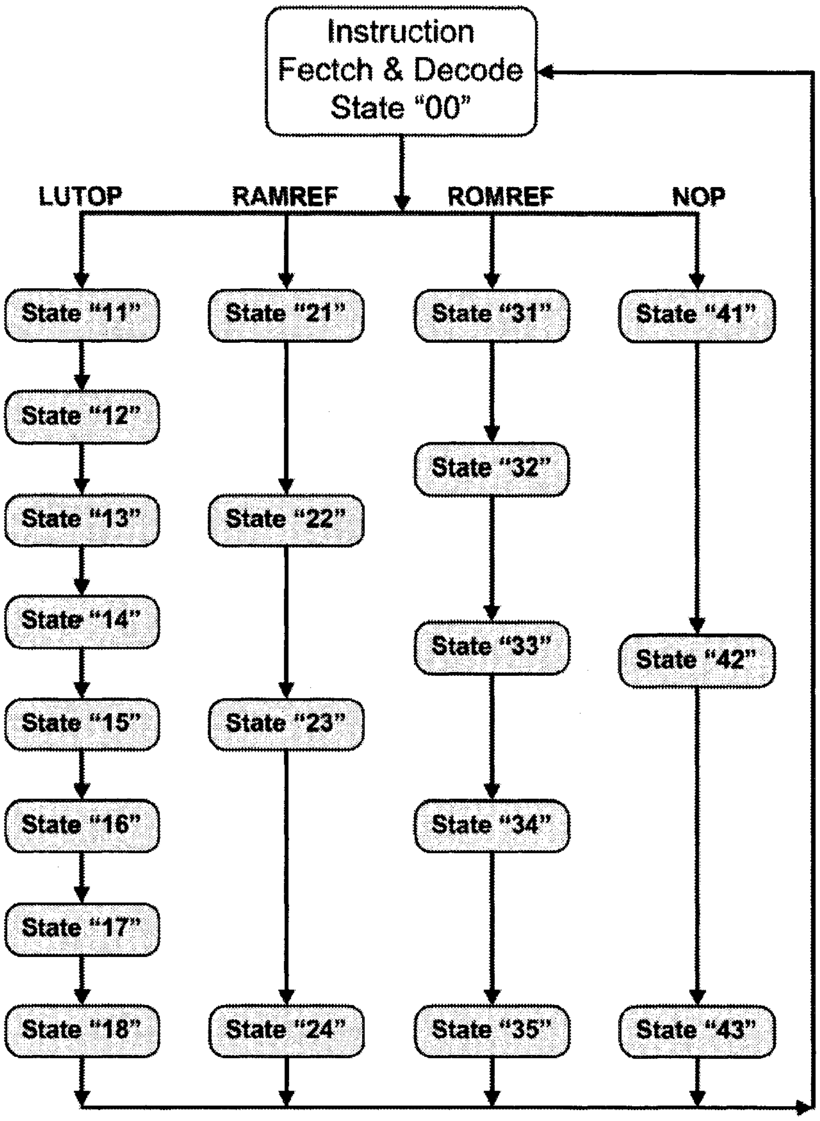


Figure 3.8: HEP's Control Unit Finite State Machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

source for “Operand C”. At the end of this state the value of “Operand C” is fetched from either of the memories.

5. **State “14”**: At this state the location address for the fourth input to the 4-input LUT (i. e. “Operand Address D”) is extracted from left control word and placed on LDR and IDR address busses. Also, bit 31 of the left control word selects either LDR or IDR as the source for “Operand D”. At the end of this state the value of “Operand D” is fetched from either of the memories.

6. **State “15”**: By the end of “State 14” all four operands to the 4-input LUT are fetched from data memories. These four operands construct a 4-bit address to the 4-input LUT (see Fig. 3.3). Hence, at the beginning of State 15, the “Read” signals to data memories are disactivated, marking the end of the operand read cycle. During State 15 the 4-input LUT generates one-bit value as an output. The output of the LUT is stored in a temporary buffer within the HEP processor and will be stored in LDR later at “State 17”. Also, each HEP processor will receive one input bit from one of the 64 processors. The received bit must be stored in the IDR memory. The location inside IDR where the input bit must be stored at is addressed by the current value of Program Counter Register. Therefore, at this state the value of program counter register is placed on the address bus of IDR. Also, the “Write” signal to IDR memory is activated. At the end of this state processor’s “bit-in” is latched (written) into IDR.

7. **State “16”**: At this state, write cycle to IDR is terminated. The output of the LUT is transferred from the temporary storage to the internal data bus of the processor so that, on the next state, it would be stored inside the LDR memory.

8. **State“17”**: At this state, the output of LUT appears on the “Node Bit Out” pin of the processor. This value must also be stored inside LDR memory where value of Program Counter Register is pointing to. Hence, the content of program counter is placed on LDR’s address bus and the memory’s “Write” signal is activated. At the end of this state, the output of LUT is stored in LDR. Also, Program Counter Register is automatically incremented by one.

9. **State “18”**: At this state the, LDR’s write cycle is terminated which, in fact, marks the end of execution cycle of one LUTOP instruction. At the end of this state, the processor jumps back to State “00” which initiates fetching of the next instruction in control memories.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

10. **State “21”**: The controller jumps to this state if the new instruction happens to be a “RAMREF” instruction (see 3.4). The function of RAMREF instruction is to retrieve one bit value from either LDR or IDR memory arrays. Seven bits within the right control word (bits 0:6) provide the address of the location where the desired value is stored. Hence, this address is applied to the address bus of both data memories (LDR and IDR). Then “Read” signals to both memories are asserted (activated). The “source” bit in the right control word (bit 28) specifies the LDR or IDR as the supplier. At the end of this state a single bit value is fetched from one of the data memories. Also, at this state, the six-bit “Node Address” is extracted from the right control word (bits 32:37) and applied to the 64-input MUX to select the input bit to the processor among 64 inputs.

11. **State “22”**: At this state the value that was fetched from either of data memories during State “21”, is latched within a temporary storage inside the processor.

12. **State “23”**: At this state the “Read” signals to both data memories are disactivated which marks the end of memory read cycle. Also, the input bit to the processor which was selected during State “21” has to be latched inside IDR. Hence, the address where the input bit has to be stored inside IDR is provided by Program Counter Register and applied to IDR’s address bus. Then the “Write” signal to IDR is activated and input bit to the processor is stored inside IDR. At the end of this state the Program Counter Register will be automatically incremented by one.

13. **State “24”**: At this state the “Write” signal to IDR memory is disactivated to mark the end of the data memory write cycle. Also, the single-bit value that was previously fetched from either of data memories (LDR/IDR) during State “21” is transferred to the output pin of the processor (i. e. “Node Bit Out”). This value would be the output value of the processor at the end of the RAMREF instruction. At the end of this state the controller will jump back to State “00” to initiate fetching of the next instruction.

14. **State “31”**: The controller jumps to this state if the new instruction happens to be “ROMREF” instruction (see 3.4). The function of ROMREF instruction is to retrieve one bit static value from right control memory. To perform this operation, ROMREF instruction will need to fetch a second word from the right control memory. Therefore, at the beginning of this state, the address of location where the second word is stored, will be extracted from the seven least significant bits of the left control word and applied to the address bus of the right control memory. At the end of this state a 16 bit value is fetched from the right control memory.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

15. **State "32"**: Among the 16 bits fetched from the right control memory at State "31", only one bit is desirable. To extract the bit value, the 16 bit value is loaded into the logic function table of the 4-input LUT. Four bits in the left control word (bits 7:10), also referred to as "bit address" are used as the input address to the 4-input LUT. Once the four bit address is applied, the LUT will extract one bit among the 16 bit value. At the end of this state the extracted bit value will be stored in a temporary register inside the processor.
16. **State "33"**: In this state the processor will select one input among all 64 inputs to the processor. The processor input must be stored in IDR memory. Therefore, at this state the location where the input bit has to be stored at inside the IDR will be provided by Program Counter Register. The "Write" signal to IDR is also activated. At the end of this state the "Node Bit-in" is stored inside IDR memory.
17. **State "34"**: At this state the single bit value, which was extracted from the right control memory during the state "32", will be transferred to output pin of the processor ("Node Bit-out"). Also, a copy of that bit has to be stored inside LDR memory for future references. Hence, the address of the location where that value has to be stored is provided by the Program Counter register and placed on the address bus of the LDR. Subsequently, the "Write" signal to LDR is activated. At the end of this state the single bit value retrieved by the ROMREF instruction is stored in LDR memory.
18. **State "35"**: This state marks the end of the processor's write cycle. The Program Counter Register is incremented by one. At the end of this state the processor will jump back to State "00" to initiate the fetching if the next instruction.
19. **State "41"**: The controller jumps to this state if the new instruction happens to be "NOP" instruction (see 3.4). The NOP instruction performs no significant function. It causes the processor to delay for one instruction cycle. The only activity that takes place during this instruction is that the processor receives a single input bit from one of the 64 processors and stores the value inside the IDR memory. To perform that, six-bit "Node Address" is extracted from the right control word (bits 32:37) and applied to the 64-input MUX to select the input bit to the processor.
20. **State "42"**: The location where the input bit has to be stored inside IDR memory is provided by Program Counter register and is applied to address bus of IDR memory. The "Write" signal to IDR is activated at this state. By the end of this state the input value is stored inside the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IDR memory.

21. **State "43"**: This state marks the end of memory's write cycle as well as the processor's instruction cycle. At the end of this state the processor returns to State "00" to initiate another instruction cycle.

Although there are total number of 21 states shown in control unit's FSM, we have managed to "combine" all the states in to only 9 states during implementation. Also, we have used "one-hot" encoding technique to further simplify the structure of HEP processor. Consequently, the longest path, from the start of the FSM towards the end, consists of total of 9 states. Given the fact that each state takes one clock cycle to finish, maximum instruction execution time in an HEP processor is $9 \times T_{clock}$, where T_{clock} is the period of processor's clock signal.

3.6 Control Memory of HEP

An HEP processor contains two memory arrays which, together, store the emulation program assigned to each processor. These memories are referred to as *Left* and *Right Control* memories (Fig. 3.9). Each control memory stores 128 control words, executed sequentially and repetitively under the control of program counter (global sequencer) register. Each revolution of the program counter from zero to a predetermined maximum value($\leq$ 127) corresponds to one design path clock cycle in DUT. A left control word and a right control word in the control memories are simultaneously selected during each instruction cycle.

Each instruction word in the left control memory consists of 18 bits. The two most significant bits in the left control word always (bits 16:17) indicate the instruction op-code (for details about each field of left control word please refer to3.4). The functionality of remaining bits in the left control word (bits 0:15) depends on the type of the instruction. The left control memory is always addressed directly by the step value inside the program counter register. Each instruction word in the right control memory consists of 38 bits that, depending on the instruction type, might be interpreted differently (for details about each field of right control word please refer to 3.4). The right control memory is usually addressed by the step value inside the program counter register unless the "ROMREF" instruction is being executed. In such case, contents of the right control memory are interpreted as static data in the emulated memory array and is addressed by the value extracted from left control word. Accordingly, any of the right control words may be addressed during any step of the sequencer and only the left control words are sequentially addressed by the program counter register.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

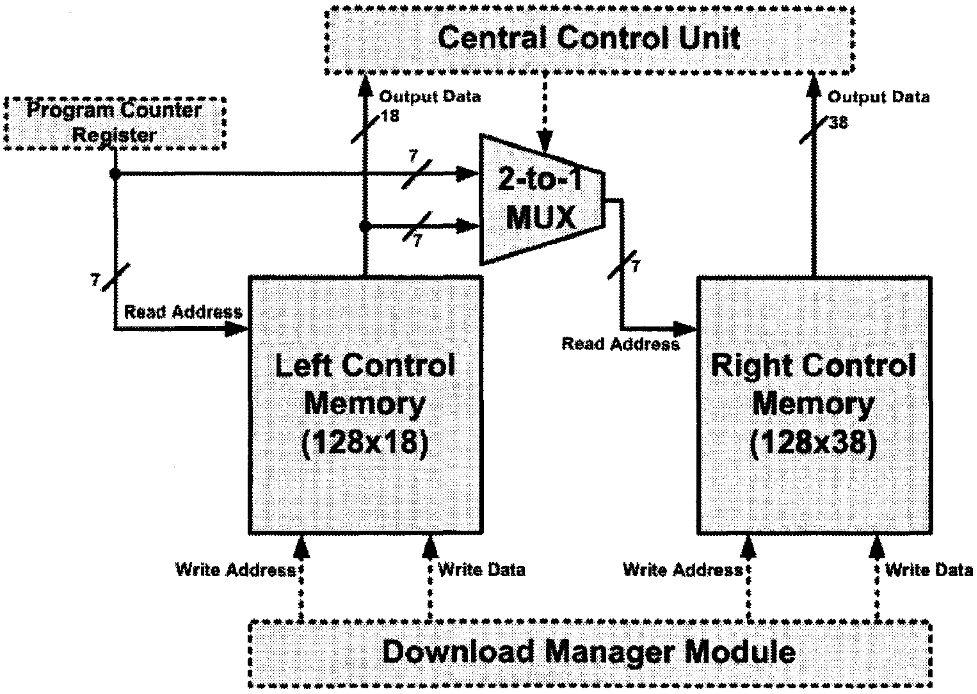


Figure 3.9: HEP's Control Memory structure.

The contents of both Left and Right Control memories are uploaded only once during the initialization of the emulation engine. During this time all the processors will be halted and no operation will take place. Therefore, both control memories have additional address and data busses for downloading binary information in to them. These ports are managed by the an external "Download Manager Module" within the emulation engine. Once downloading bitstreams into control memories is finished, the download manager reset all the HEP processors in the emulation engine and the processors start the emulation process synchronously.

3.7 Data Memory of HEP

Each HEP processor has two 128-by-1 bit memories for data storage. These data memories are referred to as *Local Data RAM*(LDR) and *Input Data RAM*(IDR). The LDR memory stores a copy of the the output bit generated by the processor after executing each instruction. The IDR memory, on the other hand, stores the single bit value that a processor receives from one of the 64 processors in the emulation engine during each and every instruction execution. The write address to both data memories is always provided by the step value stored inside the program counter register (global

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

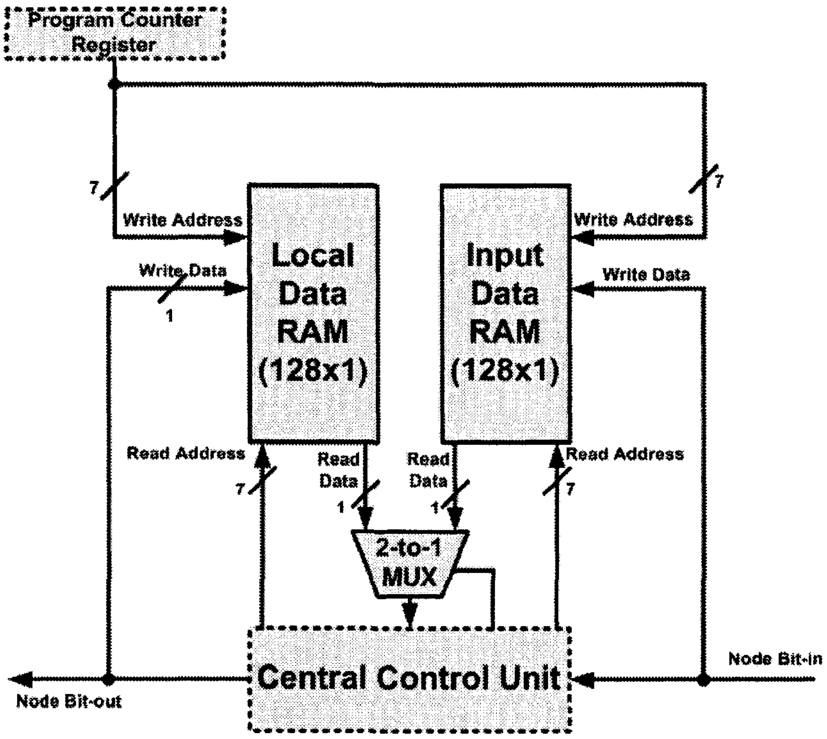


Figure 3.10: HEP's Data Memory structure.

sequencer). The read address to data memories is provided by fields inside the right control word of each instruction. Figure 3.10 shows the block diagram of data memories inside each processor. It is worth mentioning that, the IDR memory is written to during every instruction cycle. The LDR memory is written to during ever instruction cycle, except for "RAMREF" instruction.

3.8 Input/Output Ports of HEP

An HEP processor generates a single bit output after executing each instruction. The processor's output appears on the "Node Bit-out" pin which is connected to the emulation engine interconnect network. An emulation engine contains sixty four HEP processors. An output pin of one processor is connected to the input of all other 63 processors inside the emulation engine. Evidently, such interconnection network would enable each processor to receive its input, one bit at a time, from any other processor inside the emulation engine[4]. As it is shown in Figure 3.11 all 64 inputs to one

---

[4] In reality, the output of one processor is also provided as the sixty fourth input to the same processor to make the architecture more symmetric. That means, that each processor can also accept an input from itself as well. However, in this embodiment such functionality is never used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

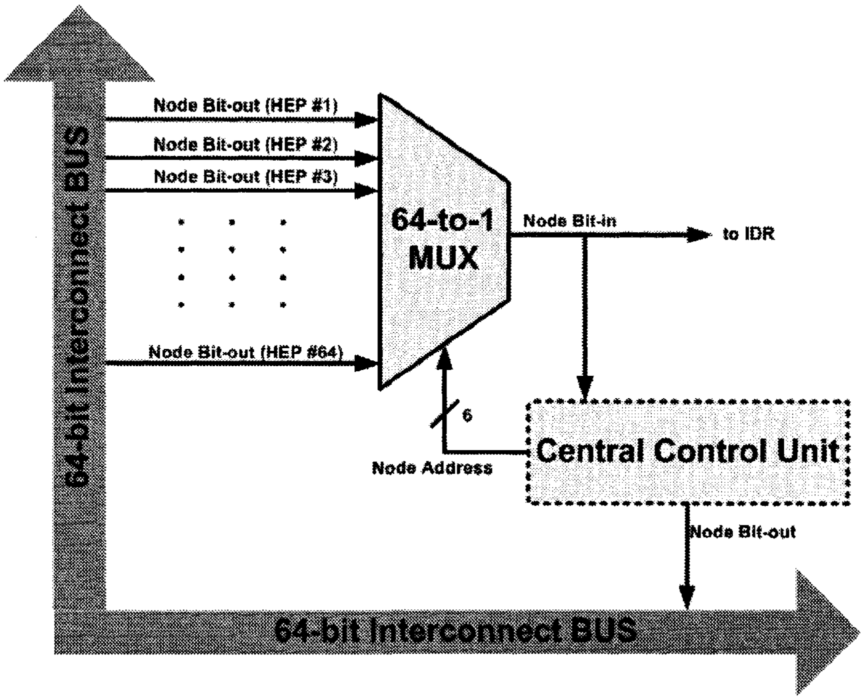


Figure 3.11: HEP's Input/Output structure.

processor are connected to a 64-by-1 multiplexer. The single input bit to one processor (i. e. "Node Bit-in") has to be selected by the same processor among all 64 inputs[5]. To do that, the processor needs a six-bit address. Six most significant bits of the instruction's right control word (bits 32:37) provides such address to the 64-by-1 MUX (for further details please refer to section 3.4). It should be emphasized that the input bit to a processor is always stored inside the IDR memory during every instruction cycle.

3.9 HEP's Program Counter Register (Global Sequencer)

As it was mentioned earlier in this chapter, all sixty four processors inside the emulation engine, although they execute their unique emulation program, they all step through their emulation program in synchronism. Consequently, an emulation engine should contain a *Global Sequencer* that helps all the processors to step through their program. The step value provided by the global sequencer is identical to all the processors. This value could be between zero and 127 (total of 128 steps).

---

[5] The processors described in [29] are connected to 3 adjacent processors through a mesh interconnect. Hence, each processor can receive 3 inputs simultaneously.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

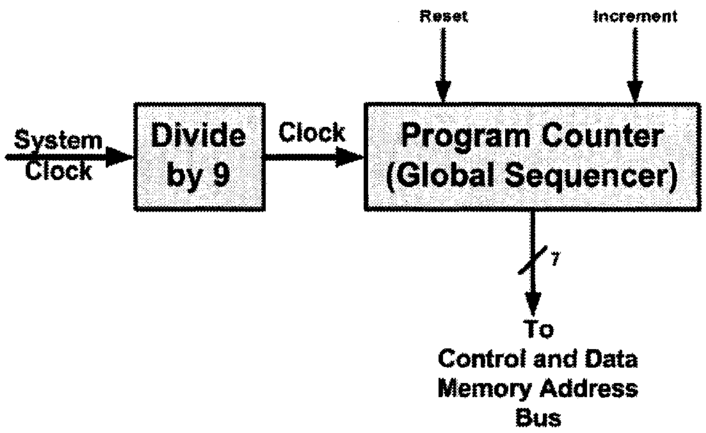

Reset
Increment
System Clock
Divide by 9
Clock
Program Counter (Global Sequencer)
7
To Control and Data Memory Address Bus

Figure 3.12: HEP's Program Counter (Global Sequencer).

However, in reality, due to the fact that the global sequencer's output has to be *fanned out* to 64 processors, we decided to "localize" the global sequencer inside each HEP processor. Therefore a global sequencer has now become the *Program Counter Register* within an HEP processor. But it has to be emphasized that at each instant of time during the emulation, the values stored in all program counter registers are equal. Since each processor can only execute total of 128 instructions, the Program Counter Register is a seven-bit long. The program counter is incremented every 9 clock pulses of system clock (Figure 3.12). The reset signal causes the program counter to initialize to zero.

3.10 Additional Signal Pins of HEP

The physical pin-out mapping of an HEP processor is shown in Figure 3.13. Each processor, being a synchronous machine, has an input *Clock* signal. The clock signal is identical to all HEP processors in the engine and, as we will see in future chapter, is referred to as *system clock*. The *Reset* signal to each processor is activated only once at the beginning of the emulation operation. Upon the activation of reset signal the program counter register is reset to "0" and all the processors will start executing instructions starting at address zero.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

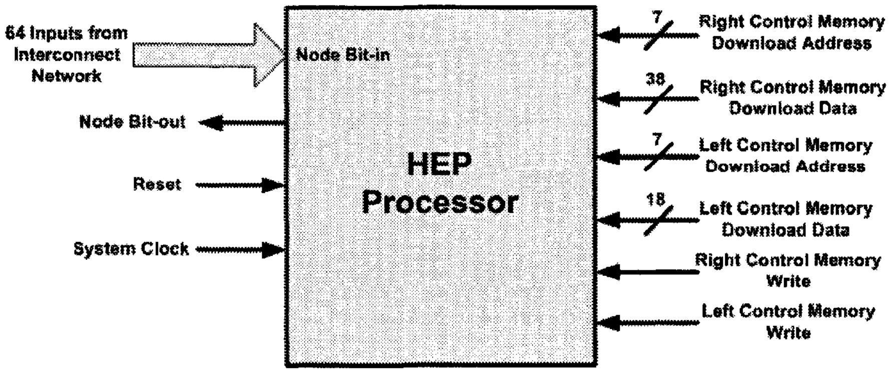

64 Inputs from Interconnect Network
Node Bit-in
Node Bit-out
Reset
System Clock
HEP Processor
7
Right Control Memory Download Address
38
Right Control Memory Download Data
7
Left Control Memory Download Address
18
Left Control Memory Download Data
Right Control Memory Write
Left Control Memory Write

Figure 3.13: HEP's Pin-out Map.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Implementation of Hybrid Emulation Processor on FPGA

In section the architecture of HEP emulation processor was described. In this chapter we present the results of simulation, synthesis and implementation of HEP-base emulation engine on Xilinx^{TM}Virtex-II and Virtex-4 FPGA devices. Also, a brief overview of other processor-based emulation systems that are being used in academia is presented. Finally, we compare the proposed architecture with other emulation systems based on size, logic capacity, speed and implementation platform.

4.1 Introduction

Until the mid 1990s, large scale digital circuits were functionally verified using software simulators and implemented using Application-Specific Integrated Circuits (ASIC). However, with the introduction of large capacity FPGAs, there has been a shift towards *reconfigurable computing* for verification and implementation. The fine-grained parallelism in FPGAs coupled with the inherent data parallelism found in many circuit simulation applications, have made reconfigurable computing an encouraging alternative that offers a compromise between performance of fixed-functionality hardware and flexibility of software-programmable devices. As opposed to general purpose processors, FPGAs allow non-standard word-length sizes and fully parallel processing, which can significantly improve throughput (e. g. one to four orders of magnitude) with only a reasonable penalty in terms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

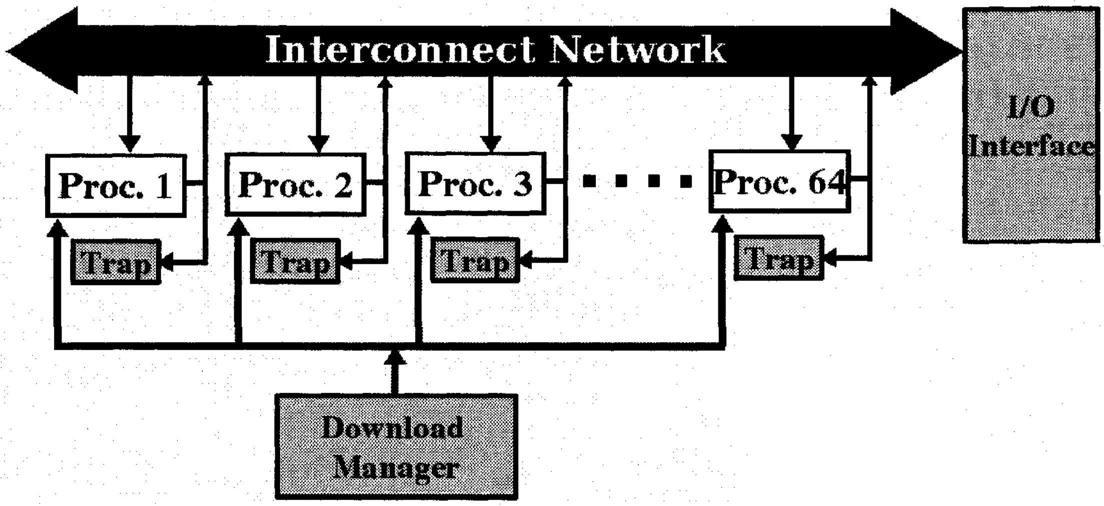


Figure 4.1: Generic architecture of HEP-based emulation engine

of implementation area (3 – 4×) [44]. Additionally, using FPGAs can offer rapid prototyping of emulation engines in much less time. Using FPGAs for rapid prototyping usually reduces the development time by half. Also, unlike ASICs, FPGAs provide relatively flexible visibility into the design-under-development. Last, but certainly not least, is the price factor. The logic emulation systems that use proprietary ASIC emulation processors could be much more expensive than those using off-the-shelf FPGA modules. Based on the above facts, FPGA devices were selected as the target platform to implement the proposed HEP-based emulation engine.

4.2 Design Specifications for HEP-based Emulation Engine

The generic architecture of the proposed emulation engine is shown in Figure4.1. The engine consists of the following modules:

1. Sixty four HEP processors and the interconnection network
2. Target System I/O Interface
3. Download Manager Module (DMM)
4. Signal Trap Module

The heart of the engine consists of 64 HEP processors that communicate through a time-multiplexed interconnection network. This module is in fact the target platform for the developed CAD tool, which will be discussed in later chapters.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

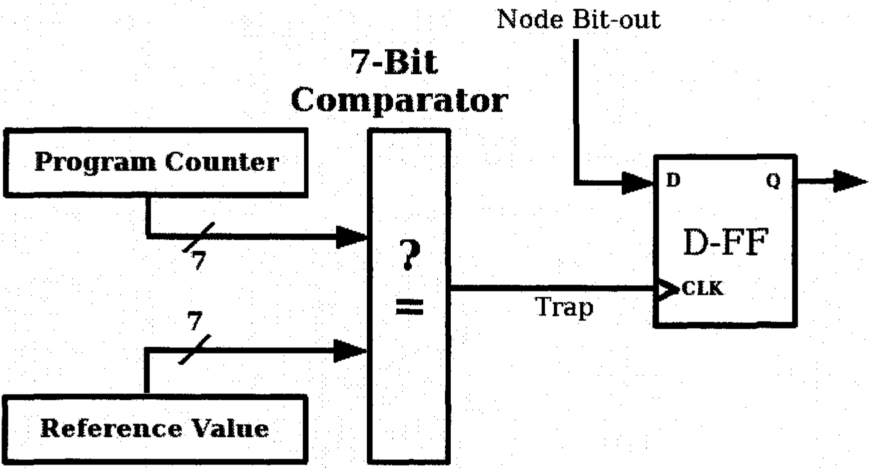


Figure 4.2: Example of Signal Trap circuitry.

The "target system I/O interface" module connects the emulation engine to the target system where the DUT will be eventually mounted. The main task of I/O interface module is to acquire signal samples from the target system and assign these inputs to emulation processors in appropriate emulation cycles. The *Download Manager Module* (DMM) performs two main tasks: Before the emulation starts, it downloads the emulation program bitstream into Left and Right Control memories of all 64 HEP processors inside the engine. Once the downloading process is finished, the DMM signals all processors simultaneously to start the emulation by activating their "Reset" signal.

Signal Trap module helps the emulation engine to "trap" (i. e. latch) a signal value during emulation runtime. This module is programmable by user, who determines which signal at what time should be monitored. Each signal trap module is associated with one processor which creates a flexible signal monitoring capability. It is worth emphasizing that signal trap modules can be very simple or very sophisticated with respect to their structure or functionality. In the simplest form, a signal trap module consists of an "n-bit" digital comparator and a D-FlipFlop (Fig. 4.2). The comparator compares the value of Program Counter Register (Global Sequencer) with a predefined value (determined by the user). If these two values become equal (i. e. Program Counter reaches certain emulation cycle) the processor's output ("Node Bit-out") will be stored (trapped) in D-FlipFlop. Later on, any "monitoring" mechanism can extract and echo the trapped value to the user. This way users can trace or monitor virtually all the events in DUT during run-time. It should be emphasized that the main focus of this research was the evaluation and implementation feasibility of the HEP-based emulation core and the study of other submodules such as I/O interface, DMM and monitoring are left for future research.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3 RTL Design of HEP-Based Emulation Engine

Conventionally, FPGA design and implementation involves a top-down design flow, illustrated in Fig. 4.3 which was applied in implementation of the proposed emulation engine as well. The first step in the design process involved identifying hardware specification and general functionality of emulation engine. Based on the specifications, the *register-transfer-level*(RTL) models and test benches for each individual submodule in the engine were developed using VHDL language. RTL design refers to the methodology of modeling a sequential circuit as a set of registers and a set of transfer functions which describe the flow of data between the registers. Each submodule, is developed in VHDL using both *behavioral* modeling, to describe the functionality of the submodule, as well as *structural* modeling to instantiate and bind comprising submodules together. The design was simulated at the RTL level by running the testbenches using ModelSim®. We chose a sequential 4 × 4-bit binary multiplier as an example of DUT and performed "sanity checking" on the emulation engine to confirm the correct functionality of the proposed engine. However, timing and FPGA resource usage remains unknown until logic synthesis is performed. FPGA logic synthesis is performed to create an optimized gate-level netlist which is based on design constraint such as timing (speed), area, I/O pin and power. Once the gate-level netlist is generated and mapped to the target FPGA's logic-elements, the design (i. e. Emulation Engine) is placed and routed inside the FPGA(s). The synthesis constraint also affect the effort required for placement and routing. If the design is over-constrained it is very likely that routing failure will occur since routing resources are fixed in FPGAs. The last step in the design flow is the generation of configuration bitstream file that can be downloaded into FPGA.

It has to be emphasized that some intermediate steps in the FPGA design flow are not shown in Fig. 4.3. In practice some of the steps might be executed iteratively. There are a number FPGA *electronic design automation*(EDA) tools that are provided by both FPGA and third party manufacturers. Complete design environments are offered by Xilinx ISE[39] and Altera Quartus II[20]. Since, Xilinx Virtex-II and Virtex-4 FPGA device family are selected as the target platform for implementing the proposed HEP-based emulation engine, we used Xilinx ISE (v7.1) as the desired FPGA EDA development tool.

4.3.1 RTL Modeling of HEP Processor

The HEP processor is described using VHDL language and IEEE_std_logic_1164 library while adopting a bottom-up approach. The RTL models of all submodules along with their functionality is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

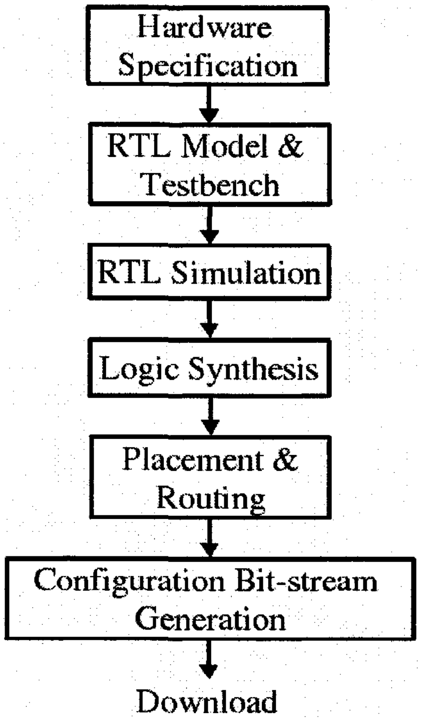


Figure 4.3: FPGA Design Flow

described behaviorally in each design file. Later, each submodule is instantiated and binded to top-level modules using VHDL structural description. The hierarchy of VHDL design files is shown in Fig. 4.4, where "EP_Top_Module. vhd" is the HEP processor top module[1]. Each design file has an associated VHDL testbench file as well [2], which are used by ModelSim to perform RTL simulation. The functionality of each design file is described below.

1. "EP_PACKAGE. vhd": Includes global constants shared by all the VHDL bodies (not shown in the figure).
2. "EP_PROGRAM_COUNTER. vhd": Describes the functionality of Program Counter Register (Global Sequencer) of HEP processor.
3. "EP_RECONFIGURABLE_4LUT. vhd": Describes the functionality of the 4-input LUT.
4. "EP_INPUT_SWITCH. vhd": Describes the functionality of the 64-input reconfigurable multiplexer that helps the processor to select the input "Node Bit-in".

---

[1] The listing of VHDL design files are presented in the accompanying CD with this thesis.

[2] Testbench filenames are similar to design files except that they are followed by "_TestBench.vhd".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

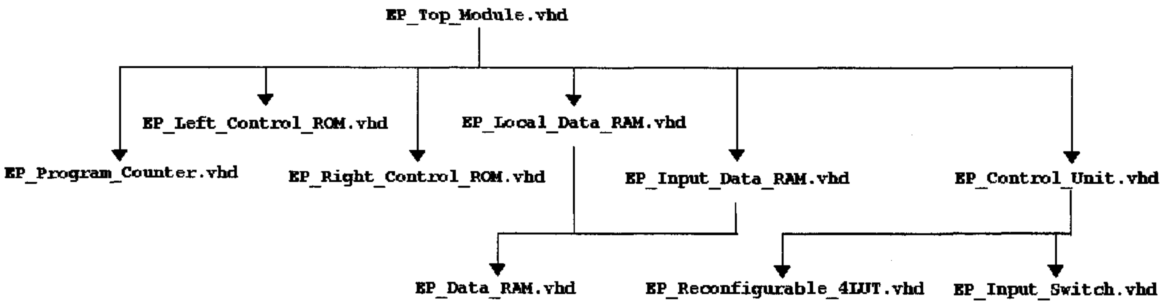


Figure 4.4: Hierarchy of VHDL design files for HEP Processor

5. "EP_RIGHT_CONTROL_ROM. vhd": Describes the structure of Right Control memory of each HEP processor.
6. "EP_LEFT_CONTROL_ROM. vhd": Describes the structure of Left Control memory of each HEP processor.
7. "EP_DATA_RAM. vhd": Describes the structure and functionality of both data RAM modules (IDR and LDR) within each processor.
8. "EP_CONTROL_UNIT. vhd": Describes the functionality of central control unit of the HEP processor. It explains how the controller's FSM actually works.
9. "EP_TOP_MODULE. vhd": This is the wrap-up module that instantiates and binds all the submodules together to build an HEP processor.

4.4 RTL Simulation Results

To investigate correct operation of HEP processor and its submodules as well as the emulation engine, we performed software simulation using ModelSim tool. A 4 × 4-bit sequential binary multiplier (Fig. 4.5) was selected as a design example to be emulated on HEP-based emulation engine. Figures 4.6 to 4.13 illustrate the simulation results.

Figure 4.6 illustrates the functional behavior of the program counter after initiating the reset signal to the emulation engine. The program counter is incremented by one during every instruction cycle.

Figure 4.7 illustrates the functionality of the reconfigurable 4-input LUT during the execution of two consecutive LUTOP instructions. "LUT_Input_x" represent the select signals to the 4-LUT module and "Input_value" is LUT value extracted from left control words.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

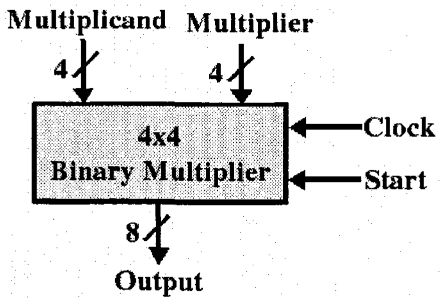


Figure 4.5: Example of 4x4 Sequential Binary Multiplier

Figure 4.8 shows the operation of 64-bit input switch of HEP processor during execution. The "address" represents the address of the processor within the module whose output is read during the instruction cycle. "Bus value" represent hexa-decimal equivalent of the value currently present on the interconnect network.

Figure 4.9 depicts read and write cycles of the input and local data RAMs. During the first write cycle a node bit-in is latched into IDR which is fetched by a RAMREF instruction during cycle 3.

Similarly, figures 4.10 and 4.11 illustrate read and write cycles of Left and Right control memories respectively. The write cycles show the process of downloading emulation programs into control memories. In the figure, the write cycles are marked by asserting "Write" signal (=1). The read cycles, however, show the instructions are fetched from program memories and are marked by asserting "Read_data" signal to high. The address of the instruction if provided through the "Read_Address" bus. The read/write cycles are synchronized with respect to system clock signal.

Figure 4.12 illustrates the functionality of HEP's central control unit while executing a LUTOP instruction. The transition through states of FSM is clearly shown in the figure ("FSM_State" signal). The value presented at the "Node_Bit_Out" represents the output value of the HEP processor.

Finally, figure 4.13 illustrates the functionality of an HEP processor after downloading 3 instructions (e. g. Two NOP and one LUTOP instruction) into control memories and initiating the start of emulation by disactivating the processor's "reset" signal. The output of the processor appears on the "node_bit_out" pin after executing the third instruction (i. e. LUTOP).

4.5 Synthesis Results

Once the proper functionality of all submodules were determined, a gate-level netlist of each submodule as well as the whole processor was generated and mapped using Xilinx ISE®(7.1) design environment. The HEP processor was synthesized targeting Xilinx Virtex II (XC2V8000) and Virtex-4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

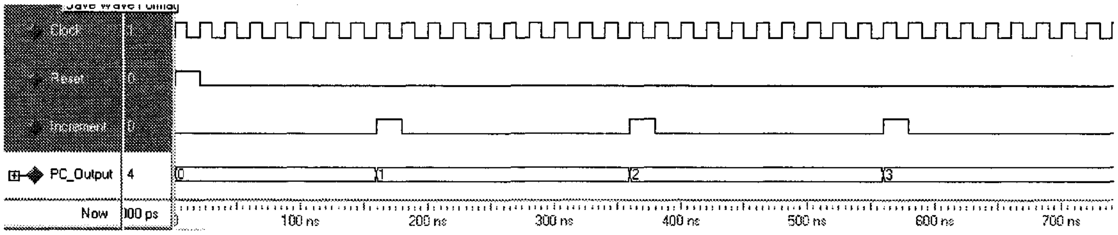


Figure 4.6: Simulated waveform view of Program Counter Submodule

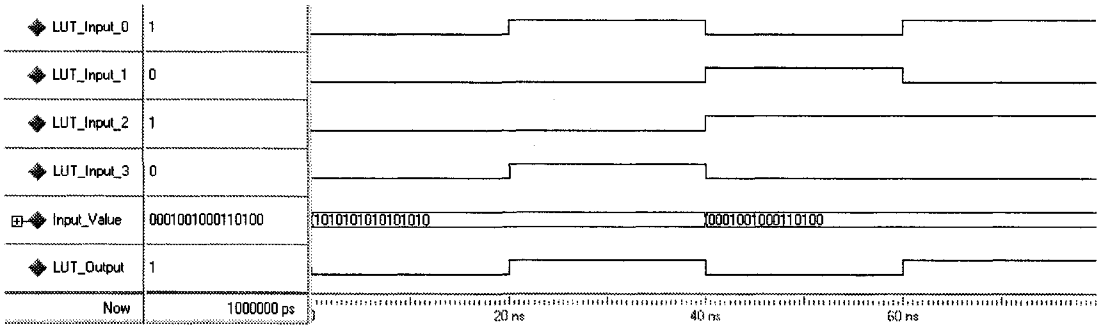


Figure 4.7: Simulated waveform view of 4-input LUT

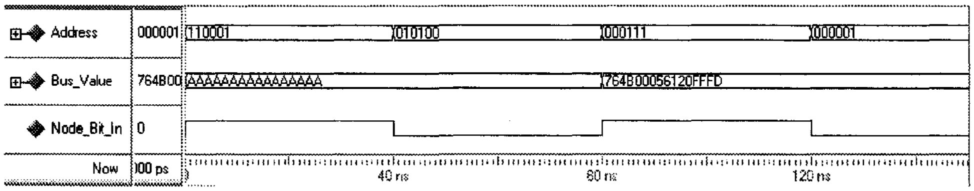


Figure 4.8: Simulated waveform view of 64-input interconnect switch

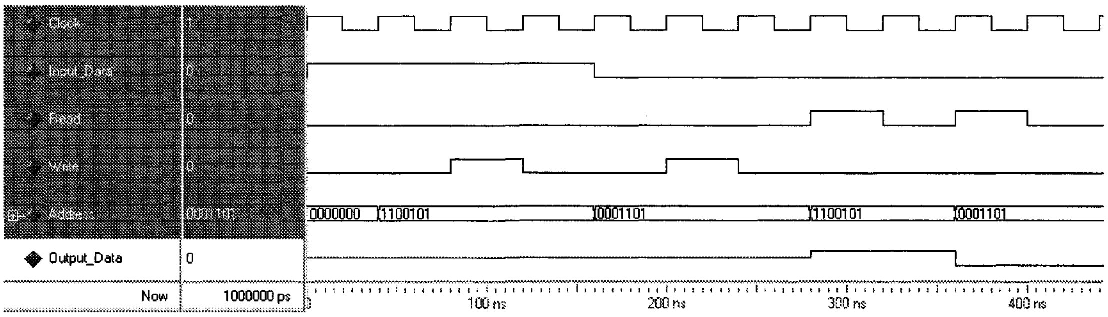


Figure 4.9: Simulated Read/Write cycles of IDR and LDR

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

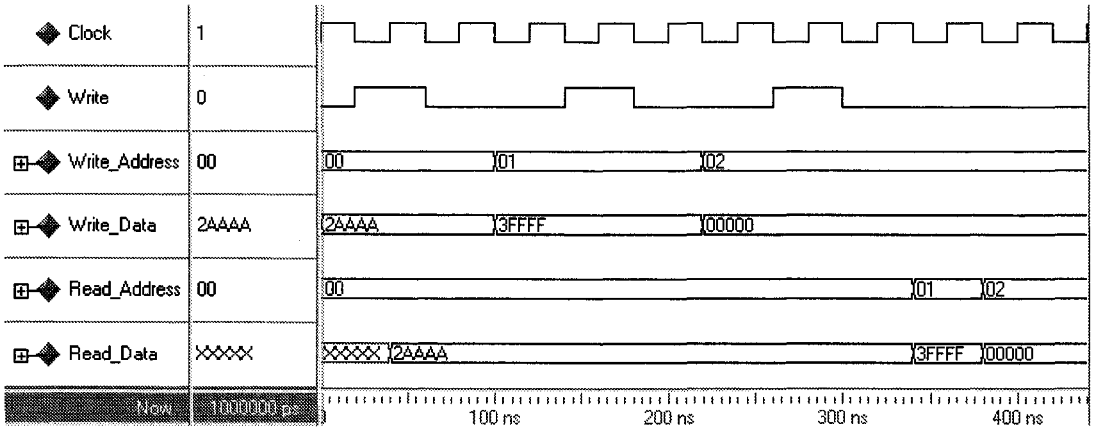


Figure 4.10: Simulated Read/Write cycles of Left Control Memory

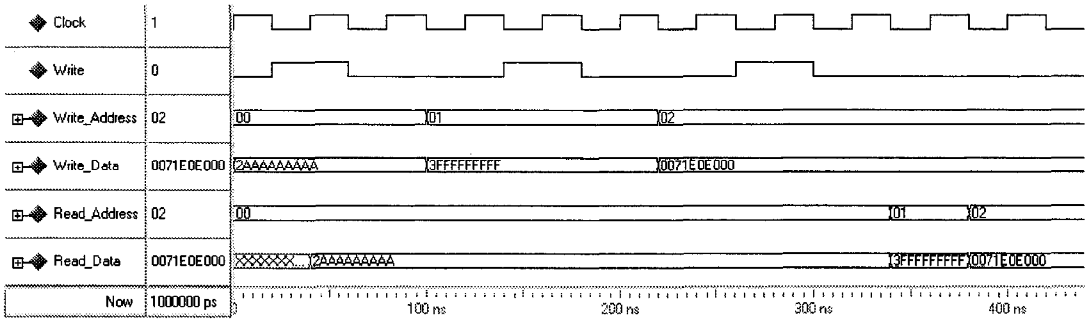


Figure 4.11: Simulated Read/Write cycles of Right Control Memory

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

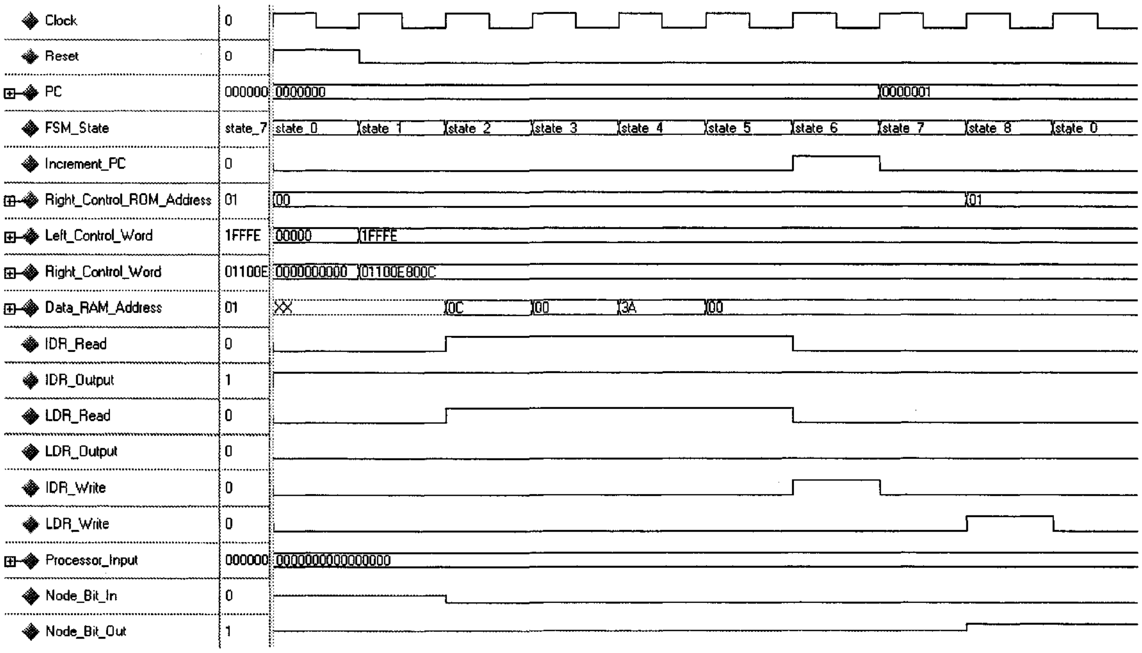

Clock 0
Reset 0
PC 000000 0000000 0000001
FSM_State state_7 state 0 state 1 state 2 state 3 state 4 state 5 state 6 state 7 state 8 state 0
Increment_PC 0
Right_Control_ROM_Address 01 00 01
Left_Control_Word 1FFFE 00000 1FFFE
Right_Control_Word 01100E 0000000000 01100E800C
Data_RAM_Address 01 XX 0C 00 3A 00
IDR_Read 0
IDR_Output 1
LDR_Read 0
LDR_Output 0
IDR_Write 0
LDR_Write 0
Processor_Input 000000 0000000000000000
Node_Bit_In 0
Node_Bit_Out 1

Figure 4.12: Simulated functionality of Central Control Unit while executing a LUTOP instruction.

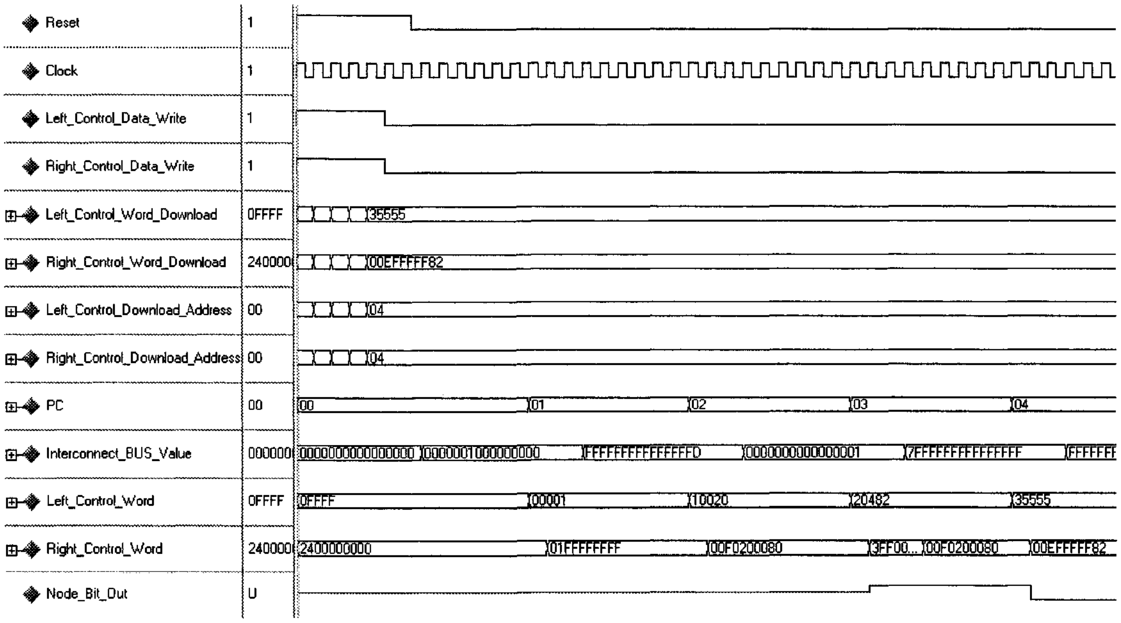

Reset 1
Clock 1
Left_Control_Data_Write 1
Right_Control_Data_Write 1
Left_Control_Word_Download 0FFFF 35555
Right_Control_Word_Download 240000 00EFFFFF82
Left_Control_Download_Address 00 04
Right_Control_Download_Address 00 04
PC 00 00 01 02 03 04
Interconnect_BUS_Value 000000 0000000000000000 0000001000000000 FFFFFFFFFFFFFFFD 0000000000000001 7FFFFFFFFFFFFFFF FFFFFFF
Left_Control_Word 0FFFF 0FFFF 00001 10020 20482 35555
Right_Control_Word 240000 2400000000 01FFFFFFFF 00F0200080 3FF00... 00F0200080 00EFFFFF82
Node_Bit_Out U

Figure 4.13: Simulation of emulation program being Downloaded/Executed on a processor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.1: Synthesis results of HEP processor and submodules.

	Virtex-2				Virtex-4			
Module	**Size (#Slice)**	**CLE Usage(%)**	**Delay (nS)**	F_{max} **(MHz)**	**Size (#Slice)**	**CLE Usage(%)**	**Delay (nS)**	F_{max} **(MHz)**
Prg_Cntr	4	0	3.6	277	4	0	1.88	531
Recon_4LUT	4	0	4.58	218	4	0	3.03	330
Inpt_Swtch	17	0	9.19	108	17	0	6.33	157
Data_RAM	60	0	4.93	202	60	0	3.15	317
Lft_ROM	228	1	5.12	195	400	0	3.19	313
Rght_ROM	479	1	5.18	193	840	0	3.3	303
Cntr_Unit	139	1	4.68	213	143	0	2.71	368
HEP	957	2	5.17	193	1529	1	3.31	301

(XC4VLX200) families of FPGA devices.

Table4.1 summarizes synthesis results for an HEP processor as well as its submodules in terms of speed, combinational path delay and FPGA resource usage while targeting both FPGA families of devices (Virtex-2 and 4). Although there are different speed packagings are available in both families of FPGAs, we are only presenting the results for the most common speed packages. As the results in the tables show, the maximum combinational path delay in the processor determines the maximum system clock frequency of the processor as well.

It is worth emphasizing that to make the VHDL design files transportable to other FPGA synthesis tools, no Xilinx proprietary library modules were used. Such assumption will force the Xilinx ISE tool to avoid using FPGA-specific resources such as embedded memory blocks.

The proposed HEP-based emulation engine, consisting of 64 HEP processors and their interconnect network was implemented while targeting Virtex-2 and Virtex-4 FPGAs from Xilinx. Table 4.2 summarizes the synthesis results obtained by Xilinx ISE. The results are summarized with respect to number of modules, FPGA resource utilization, emulation engine speed, maximum emulation time and maximum logic capacity of the HEP-based emulation system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.2: Synthesis results of HEP-based emulation system.

Feature	**Virtex-2**	**Virtex-4**
#of Modules	2	1
#of HEP/module	32	64
#Slice (%)	31150 (67%)	85525 (96%)
#I/O (%)	264 (32%)	264 (27%)
System Clock F_{max}(MHz)	193	301
Instruction Cycle (ns)	46.6	29.9
max. Emulation time (μs)	5.95	3.81
Emulation program upload time (μs)	127	77
min. Emulation frequency(KHz)	168	262
Logic Capacity	8K-160K	8K-160K

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.6 Comparison and Conclusion

Before comparing the proposed architecture, we briefly review some existing logic emulation devices that are being used mainly in academia. The survey presented here is partially derived from technical documents which are available to public. However, due to confidentiality of detailed technical information related to these system, some information are results of personal speculation.

Previous work generally fall into two main categories. The first, use time-multiplexed FPGAs in order to build denser FBE devices. Examples of such systems would be Dharma[7] and DPGA[25]. The second approach use ASIC processors developed solely for logic emulation such as YSE[26] and VEGA[40].

1. Dharma[7]: is a general-general purpose time-multiplexed FPGA designed at the University of California[3]. DUT mapped into Dharma are levelized and entire level is evaluated per clock cycle (as opposed to YSE in which circuits are serialized and only one logic block is evaluated per clock cycle). For a circuit to fit into Dharma, the number of logic blocks per level must not exceed the number of physically available logic blocks on the chip, which is a very huge disadvantage.

2. DPGA[25]: stands for Dynamically Programmable Gate Array and was developed at the MIT Artificial Intelligence Laboratory. DPGA is an FPGA with four configuration contexts and each context is stored in its configuration memory. The contexts are switched under external control. The basic logic element is in fact a 4-input LUT combined with a single flipflop that is shared among all contexts. DPGA is a general purpose hardware development platform that was not necessarily optimized for logic emulation purposes. For logic emulation purposes, a netlist must be partitioned into sub-circuits that each will fit into single context. The DPGA must contain sufficient memory capacity to store the results of each context (combinational logic blocks+flipflops) as well as configuration bitstream. Current embodiment of DPGA fails to provide such provisions, therefore, roughly speaking, it is not suitable for logic emulation. On the other hand if the time delay caused by context switching is significantly higher than emulation time of one logic slice, then emulation speed will be drastically reduced to unacceptable levels. However, DPGA demonstrate how time-multiplexing technique could result in better logic capacity utilization in FPGAs.

3. YSE[26]: Yorktown Simulation Engine was developed at IBM. Based on our classification presented before, YSE is an example of hardware-accelerated simulator that uses 256 simulation-

---

[3] It is the first time-multiplexed FPGA that has been reported in literature.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specific parallel processors to simulate (and not emulate) a logic design[4]. Unlike HEP-based emulation system, YSE does not provide in-circuit connection to target platform. Each processor in YSE is capable of simulating 4096 logic blocks. The processors are constructed from LSI TTL-based integrated circuits. The fundamental logic element used in the processors is a 4-input LUT. Signal values are represented as four-valued logic[5]. Hence, the signal state-memory has the capacity of $16K \times 2$. To allow multiple accesses to memory per clock cycle, the state-memory has five read ports and two write ports to. A 256 full-crossbar interconnect to route data among processors. Although YSE achieved low logic density due to its construction from LSI modules, it vividly proved that hardware-accelerated simulation could be 600 times faster than software simulation.

4. VEGA[40]: is an ASIC-based PBE system that was developed at the University of Toronto. Similar to HEP, VEGA also uses 4-input LUT as the basic element for emulating combinatorial logic. An additional memory associated with each processor dynamically routes the inputs/outputs to/from each processor. Although a VEGA has been implemented using ASIC technology, the emulation clock frequency is within few hundred kilo hertz.

Table 4.3 summarizes the features explained above. The last column expresses the features of HEP-based emulation engine. Comparing the results illustrated in table 4.2, the entire HEP-based emulation system, consisting of 64 processors, would require only two Vritex-2 FPGAs (XC2V8000) or just one Virtex-4 FPGA (XC4VLX200) for implementation. This means that an HEP-based emulation system is an order of magnitude smaller in size than other emulation systems. It is worth mentioning that, such reduction in size will significantly reduce the cost of HEP-based emulation system so that it is easily affordable by members of academia[6]. Also, HEP-based emulation system uses off-the-shelf FPGA modules where as most PBEs are implemented using ASIC technology. Hence, the implementation of HEP-based emulation systems takes significantly less time.

In terms of speed, an HEP-based emulation system have a clock frequency of 193-301MHz or emulation speed of 168-262KHz. Comparing with other PBEs that are using ASIC technology for implementation (e. g. VEGA), HEP-based emulation system proves to have 3-4 times faster emulation speed. Such emulation speed is quite resonable for most applications.

---

[4] However, due to architectural similarities, we can still present the results obtained by YSE

[5] In "four-valued logic" each signal can assume any of four values: "0", "1", "U" (Undefined) and "Z" (high-impedance), as opposed to Binary-valued logic in which only "0" and "1" are acceptable values.

[6] Commercially available emulation systems are at least 3 orders of magnitude more expensive than an HEP-based emulation system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.3: Comparison of HEP with other Emulation systems

Feature	**YSE**	**Dharma**	**DPGA**	**VEGA**	**HEP**
#of Elements emulated per Clock	1	1 logic level	Entire Array	1	64
#instructions per processor	4096	N/A	N/A	256-2048	128
Processing Block	4-LUT	variable K-LUT	$2 \times 4 - Lut$	4-LUT	4-LUT
Memory Architecture	5-port RAM	Latches	Flip-flop	6-port Reg. File single port RAM	Single port RAM/ROM
Max.# of Processors	256	N/A	4000	2048	64
Implementation Technology	TTL/LSI	ASIC	FPGA	ASIC	off-the-shelf FPGA

In spite the fact that most logic capacity of FPGAs will remain underutilized (due to Rent's rule), a HEP-based emulation system increases the FPGA logic utilization between 67-96% while the I/O pin utilization is only between 27-32%. Moreover, due to intrinsic flexibility in HDL, the HEP-based emulation system can be easily customized into other FPGA family of devices, such as those from Altera. Such characteristic is unique to HEP-based emulation system and is not found in other emulation systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

A CAD Tool Suite for HEP-based Emulation System

As it was mentioned in 2.3 all logic emulation systems are accompanied with associated set of CAD tools that automatically perform design compilation on DUT netlists. The ultimate goal of such tools is to perform the design compilation so that DUTs could be emulated on the emulation platform more efficiently and in less time. On the other hand, as logic designs are becoming bigger and more sophisticated, design compilation process is also becoming more time consuming. For example, logic designs as big as hundred thousand logic gates could take several hours (even days) to compile. Hence, CAD tools that prove to be efficient and fast at the same time are highly desirable.

In the previous chapters the hardware architecture of the proposed HEP-based emulation engine was described. In the following sections we are going to introduce the steps required for design compilation for HEP-based emulation engine as well as new scheduling algorithms that decrease total emulation time. At the end the results obtained by the proposed tool will be compared with others.

5.1 Basic requirements for HEP-based CAD tool

Before introducing the CAD tool flow of HEP-based emulation system, we need to understand what is the purpose of such tool and why we need it?

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

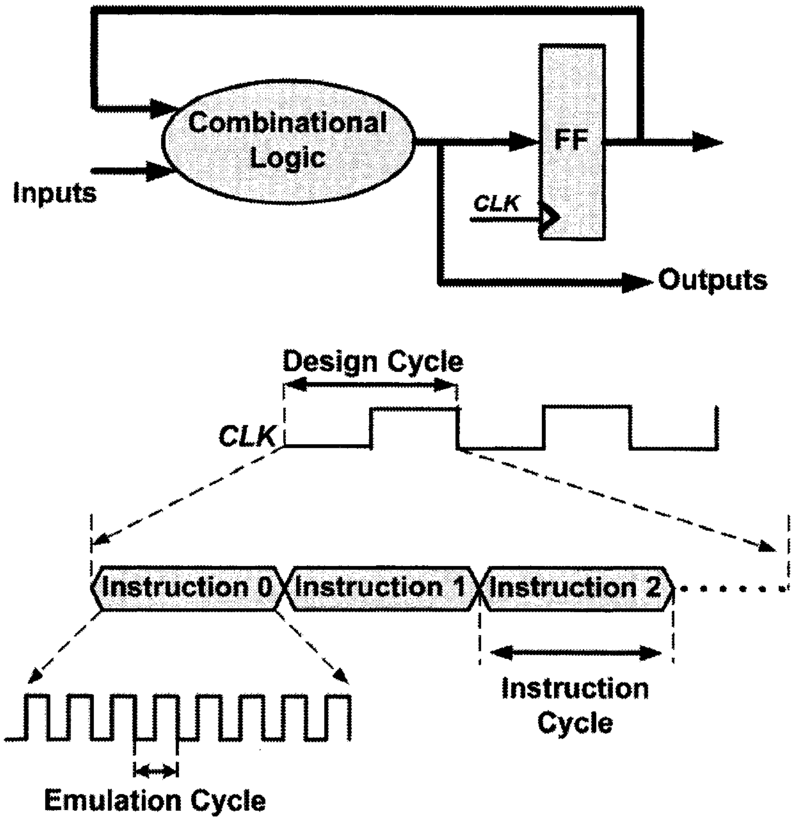


Figure 5.1: Design cycle versus Emulation Cycle in a generic DUT.

An HEP-based CAD tool should be able to automatically map any combinatorial or sequential circuit to HEP-based emulation system's hardware. A generic view of a sequential circuit is shown in Fig. 5.1. In such circuits changes in signal values is controlled (or synchronized) by "clock" signal. In this context we will refer to such signal as *design Clock*. Flip-flops are responsible for "storing" binary values and will change their values in synchronism to design clock. The combinatorial logic determines the "present-state-next-state" relationship among the signal values.

A HEP-based emulation system should be able to evaluate all signal values within time intervals marked by the design clock. During each design clock, all HEP processors will run an emulation program, by sequentially executing a series of instructions. Each instruction will take one *instruction cycle* to execute. However, for a HEP processor it takes 9 *system clock* to execute single instruction. The relation between system clock, instruction cycle and design clock is also illustrated in Fig. 5.1.

As we will see in future, an efficient CAD tool is the one that can emulate a design cycle in less number of instruction cycles.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

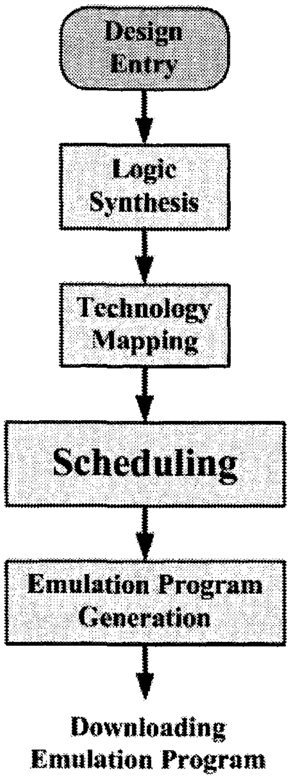


Figure 5.2: CAD Flow for HEP-based emulation system.

5.2 Overall CAD Flow

Figure 5.2 illustrates the conceptual view of proposed CAD framework for HEP-based emulation system. To map a DUT into and HEP-based emulation system, the DUT has to pass through the steps shown below.

The proposed CAD flow in most parts resembles the flow of CAD tools for PBEs, except for the fact that, now the task scheduling replaces partitioning and assignment step in PBEs. The details of each step is described below. To help the readers to have a better understanding of design compilation process, we have created a 4×4 sequential binary multiplier as a design example and taken it through the compilation steps. A block view of a 4×4 binary multiplier is shown in Fig. 4.5.

5.2.1 Design Entry

The first step of emulation CAD tool is design entry, where the user(s) (i. e. circuit designers) formally describes the functionality of the DUT. They can specify their designs through hardware description languages (e. g. VHDL/Verilog) or schematics capture tools using any industry standard tool such

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

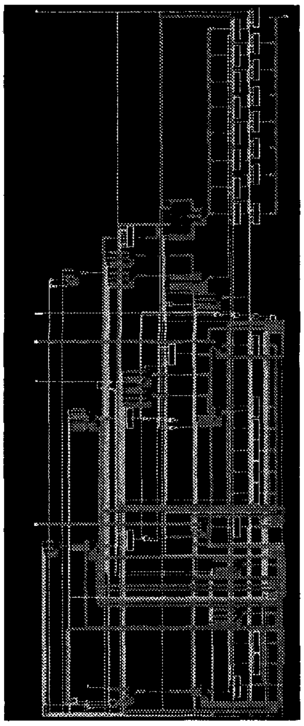


Figure 5.3: RTL view of binary multiplier produced by Synopsys Design Compiler

as Cadence Concept®HDL. In the case of the design example, the multiplier has been designed using VHDL language. The program listing of multiplier is presented in the CD accompanying this thesis.

At the end of this step, design entry tools usually produce the register-transfer level (RTL) representation of the DUT. Figure 5.3 illustrates the RTL view of the multiplier generated by Synopsys®Design Compiler.

5.2.2 Synthesis

Once the design is specified, the DUT's gate-level netlist can be obtained using any synthesis tools that support library components utilized in DUT. The synthesis tool takes the RTL netlist and automatically generates the *gate-level* netlist. An example of such synthesis tool is Synopsys Design Compiler. The synthesized gate-level netlist of the binary multiplier is shown in Fig. 5.4. It is worth emphasizing that no practical limitation on the type of the tool used for either design entry or synthesis has been set. Hence, users may use any tool available.

In order to present the results obtained by the proposed CAD tool, we have used MCNC

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

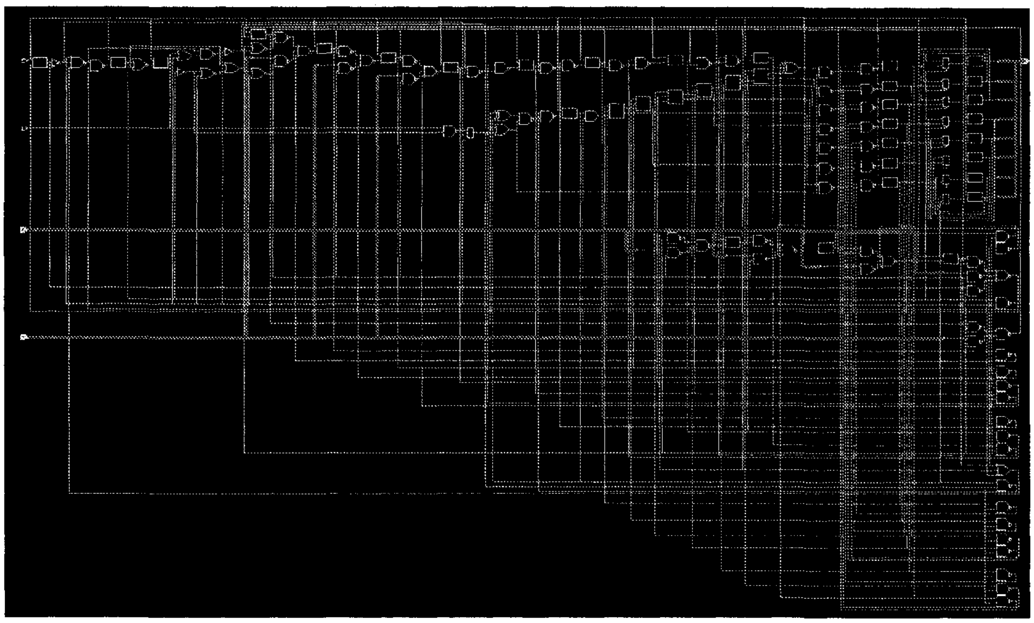


Figure 5.4: Gate-level view of binary multiplier generated by Synopsys Design Compiler.

LGSynth93 benchmark circuit suite which contains more than 100 gate-level netlists[69] presented in BLIF format. The suite contains both combinatorial and sequential circuits in various sizes ranging from a few to tens of thousands gates. However, the results of experiments performed are illustrated only for the ten biggest circuits in the suite. Table 5.1 describes the sample circuits quantitatively, in terms of number of elements (size), number of input/output and number of logic gates with fan-in[1] degrees less/greater than 4 and also the length of the critical path in the gate-level netlist "before" technology mapping. The last row of the table contains the information of the binary multiplier.

5.2.3 Technology Mapping

As the name specifies, a typical gate-level netlist contains library dependent logic primitives such as complex combinatorial logic with high fan-in degree and flip-flops. However, to emulate such design on HEP-based emulation engine, the gate-level netlist has to be transformed, so that the circuit could be mapped in to emulation system. Such transformation is called *technology mapping.* The technology mapping tool coalesces the gates/flip-flops into the basic building block of an HEP processor, i. e. a four-input LUT and flip-flops.

At this step we have used the SIS package developed at the UC Berkeley [57] to transform gate-level netlists. The "Flowmap" tool[17] was used to perform the the technology mapping. Flowmap

---

[1] Fan-in degree of a logic gate is the number of inputs to the logic gate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.1: Ten biggest MCNC circuits.

DUT	**#Logic Elements**	**#Input-Output**	**#Gates**	**#Flip-Flops**	**#Gates** (fanin≤4)	**#Gates** (fanin>4)	**Critical Path**
s38417	24011	31-109	22548	1463	22548	0	65
s38584	19699	41-307	18275	1424	18275	0	70
s35932	17793	35-320	16065	1728	16065	0	29
frisc	4425	20-117	3539	886	3539	0	23
elliptic	4724	131-115	3602	1122	3602	0	18
pdc	4775	16-40	4775	0	4775	0	9
des	2263	256-245	2263	0	1464	799	10
i10	2452	257-224	2452	0	2291	161	55
C7552	3466	207-108	3466	0	3410	56	43
C5315	3088	178-123	3088	0	3067	21	79
Multiplier	136	10-8	106	30	106	0	10

is an LUT-based technology mapping tool which produces depth-optimal mapping solution for K-bounded Boolean networks. The algorithm calculates min- cost K-feasible cuts for all the logic gates in the circuit. Flowmap can be run to minimize either total area or total delay. "Delay" minimization, in this case, is the minimization of the number of LUTs on the circuit's *critical path.* However, since maximizing the emulation speed is the main objective, circuits should be mapped to so that the area is minimized. Smaller area results in fewer LUTs, which, generally, reduces the number of emulation cycles. In case of HEP-based emulation system, since each processor contains a 4-input LUT (4-LUT), Flowmap has to convert the gate level netlists into a collection of LUTs and flip-flops. An example of technology mapping process is illustrated in Fig. 5.5. In the example shown, the technology mapping tool has not only reduced the area but also the "depth" of the circuit, resulting in a circuit with minimum delay.

However, the experiments show that if the DUT netlist is "decomposed" before it is technology mapped by Flowmap, the final circuit contains less logic elements (i. e. less area). The decomposition process is performed using SIS DMIG tool [14] that converts all the logic gates in an unbounded gate-level netlist into a collection of two-input (i. e. 2-bounded) logic gates. The DMIG tool uses *tree-balancing* technique to obtain a depth-optimal solution to break a netlist into logic gates with "fan-in" degree less than or equal to 2. Figure 5.6 illustrates technology decomposition of a logic gate with fan-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

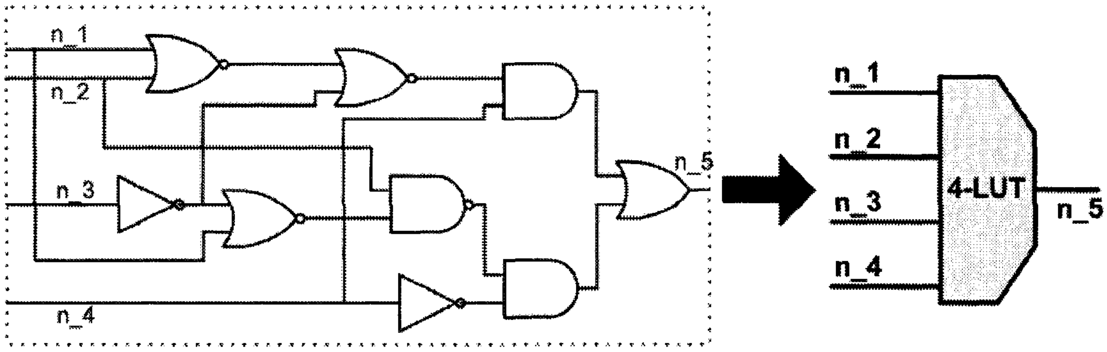


Figure 5.5: Example of technology mapping for reducing area and delay.

in degree of four. As it is shown in the figure, balanced-tree technology decomposition usually results in a circuit with shorter critical path. Although, technically speaking, logic decomposition could be performed independently from mapping, we refer to combination of both steps as technology mapping. The scripts used for logic decomposition and technology mapping is provided in the complementary CD along with this Thesis.

Table 5.2 summarizes the results obtained for the 10 biggest MCNC circuits (as well as binary multiplier example) after logic decomposition and technology mapping. The results are shown for having the circuits decomposed and not decomposed prior to mapping. Interestingly, having the circuits logically decomposed prior to mapping has reduced the critical path length in the final circuit. Such reduction results in reduction of number of emulation cycles and increases the emulation speed.

Although technology mapping helps to reduce the critical path length (almost) in all cases, but it does not necessarily reduce the size of the circuit. In some circuits (e. g. DES), the technology mapped circuit will contain even more logic elements (i. e. bigger in size) compared to its size before technology mapping. Such observation could be attributed to high fan-in degree (> 4) of substantial number of logic gates in the circuit.

5.2.4 Scheduling

According to computer science literature, an HEP-based emulation system is an example of a special purpose platform that can be classified as a synchronous *Multiple Instruction Multiple Data* (MIMD) multi-processor system. An MIMD system contains a number of *processing elements* (PE), or simply, processors, that run in parallel while each PE contains a unique area for program and data. A program is a collection of "tasks" that must be executed by processors in a specific sequence. However, the greatest challenge ahead of researchers is partitioning applications into tasks, coordinating

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

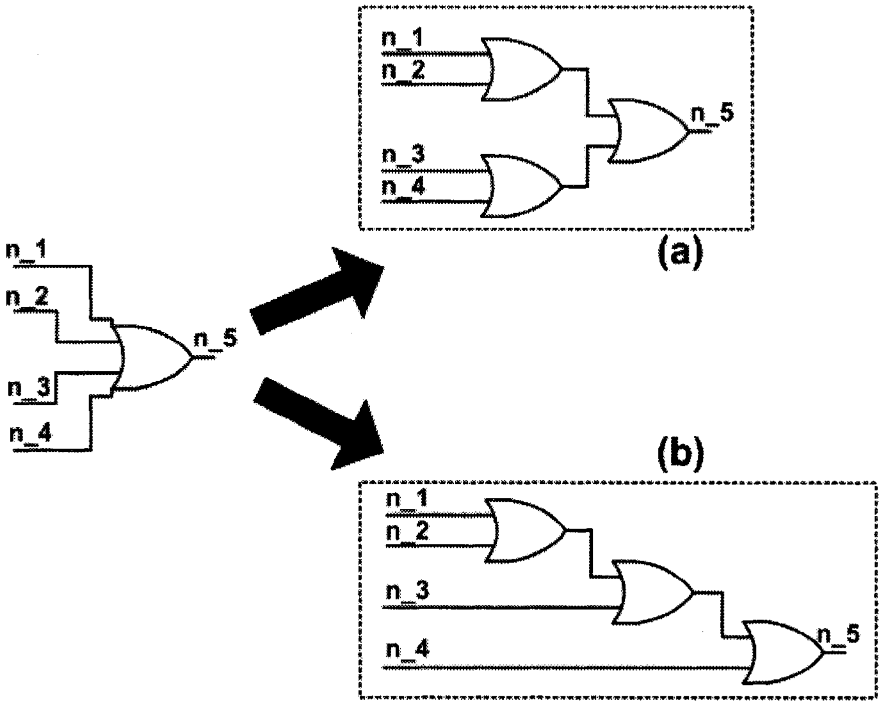


Figure 5.6: Technology decomposition. (a) Balanced-tree, (b) Unbalanced-tree.

Table 5.2: Results of technology mapping.

	Original		w/o. Decomposition		Decomposition	
DUT	**Size**	**Critical path**	**Size**	**Critical path**	**Size**	**Critical path**
s38417	24011	65	5372	11	5411	10
s38584	19699	70	6704	13	6630	9
s35932	17793	29	5152	4	5152	4
frisc	4425	23	6529	23	7362	23
elliptic	4724	18	5563	18	6190	18
pdc	4775	9	6314	9	6796	9
des	2263	10	3369	6	3957	6
i10	2452	55	1373	16	1401	13
C7552	3466	43	933	8	907	8
C5315	3088	79	837	10	802	9
Multiplier	136	10	99	8	99	8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

communication, synchronizing processors and "scheduling" tasks on the parallel platform[45].

Scheduling and allocation of tasks is extremely crucial since an inappropriate scheduling of tasks can fail to exploit true potentials of the system and can offset the gain from parallelization. *The objective of scheduling is to minimize the completion time of a parallel application by properly allocating the tasks to the processors*[45]. In a broad sense, the scheduling problem exists in two forms:

- *Static*: In static scheduling, which is usually done at compile time, the characteristics of a parallel program (such as processing times, inter-processor communication, data dependencies and synchronization requirements) are known before the program execution.
- *Dynamic*: In dynamic scheduling only a few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made "on-the-fly" (during program execution).

In this application, after technology mapping, the generated netlist consists of a collection of logic elements. Emulating the functionality of each element can be viewed as a "task" for a HEP processor in the emulation engine. Taking such analogy, the whole technology mapped netlist is considered as a parallel "program" that has to be emulated on 64 HEP processors. The most important questions here to answer are: *how should we break the program into smaller tasks? and how these task should be scheduled and assigned to processors so that the execution time is minimum?*

Obviously, due to the fact that the characteristics of the technology mapped netlist is known prior to scheduling, task scheduling can be accomplished using "static" scheduling techniques.

The scheduling problem is an NP-complete problem for most cases [45]. Hence, many heuristics with polynomial-time complexity have been suggested. However, these heuristics are highly diverse in terms of their assumptions about the structure of parallel program and the target parallel architecture.

In the following sections of this thesis, the task scheduling problem for HEP-based emulation system is addressed. In this research, new heuristic algorithms and tools that can perform the task scheduling for HEP processors that reduce the emulation time have been developed. The algorithms are extensions to the static scheduling algorithm called *list scheduling.* The algorithms described below could also be applied to any architecturally similar PBE.

5.2.4.1 Preliminaries

From the scheduling tool point of view, a DUT netlist is a parallel program that consists of hundreds to thousands of tasks that have to be executed on a number of logic processors. To schedule tasks,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

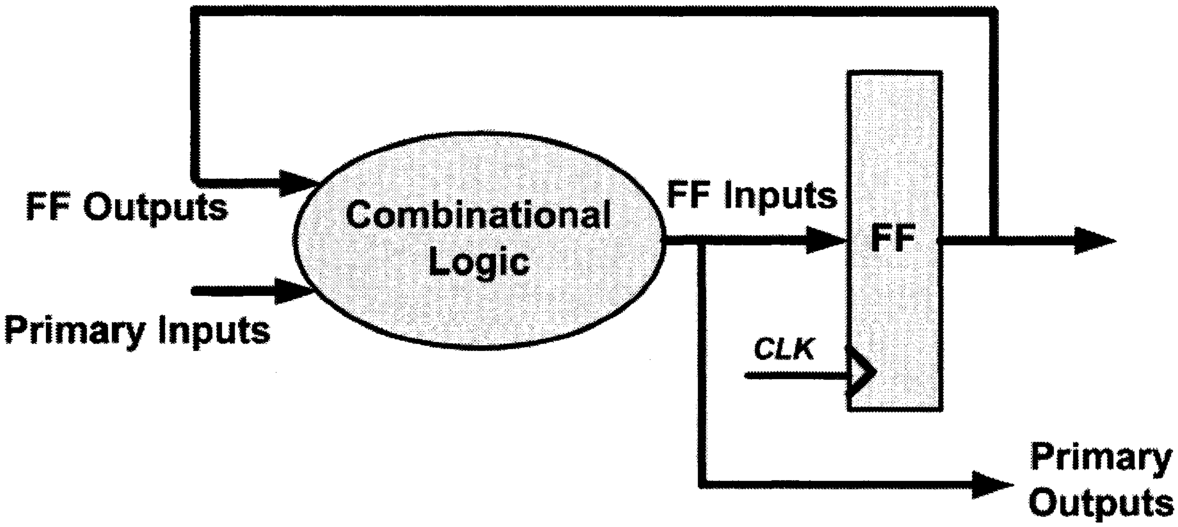


Figure 5.7: Modeling a DUT as a Mealy Machine.

first, the *task precedence graph*(TPG), in which, nodes represent the tasks and the directed edges represent the execution dependencies, as well as, the amount of communication, is built. Such modeling, is commonly used in static scheduling of a parallel programs with tightly coupled tasks on multi-processors. In circuit terminology, TPG is equivalent to *directed acyclic graph* (DAG) and therefore the two can be used in this context interchangeably.

To construct DAG representation of a netlist first the inputs and outputs of DUT must be identified. A sequential circuit could be rearranged using *Mealy machine* model illustrated in Fig. 5.7. In Mealy machine model, a DUT consists of combinatorial logic combined with flip-flops that store the "present state" of the circuit[2]. Inputs to a circuit are either the *primary inputs* (external inputs) or any fed-back *flip-flop outputs*. The combinatorial logic establishes "present-state- next-state" relationship in the circuit. The circuit outputs are either the *combinatorial outputs* or the *flip-flop inputs*[3]. In a technology mapped circuit the combinatorial logic consists of 4-input LUTs.

Figure 5.8 illustrates DAG equivalent of a DUT netlist. A node in DAG is equivalent to a logic element (4-LUT or FF) in the DUT netlist. Mathematically, a DAG is shown as $G = (V, E)$, where V is the set of all the vertices (nodes) and E is the set of all the edges. The weight $w(n_i)$ assigned to node n_i represents its computation cost. However, in an HEP processor the computation costs for

---

[2] Roughly speaking, a flip-flop (FF) is one bit of "memory" element that can store a binary value for infinite duration of time. Hence, a flip-flop can also be regarded as a logic unit that is capable of keeping a "history" of signal values

[3] The same definitions for input/outputs will also apply to merely combinatorial circuits (memory-less circuits) except that they do not include flip-flop inputs/outputs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

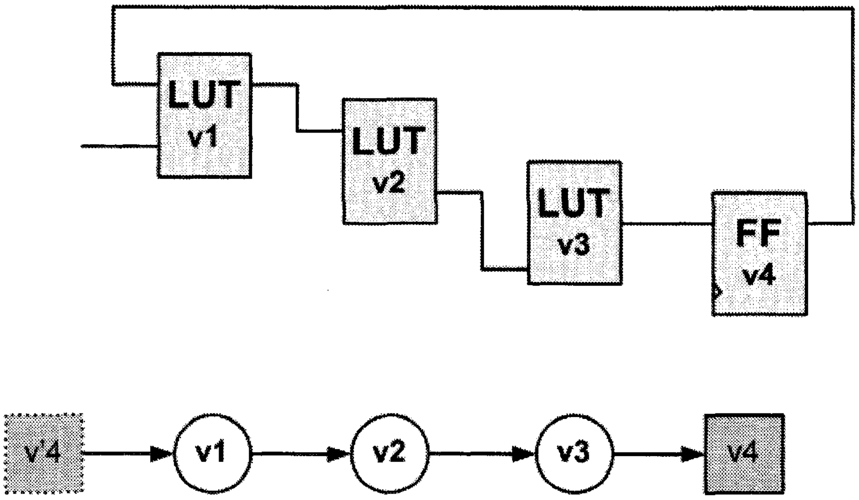


Figure 5.8: DAG representation of netlist

all logic elements are equal, because each logic element can be emulated in one HEP's instruction cycle. Thus, $w(n_i) = 1$ for all $n_i \in V$. Also, the weight $w(e_{ij})$ assigned to edge e_{ij} represents the communication cost between two nodes n_i and n_j. Recalling from previous chapters, during each instruction cycle, an HEP processor is capable of receiving/transmitting value calculated for one logic element in the graph from one processor to another. Hence $w(e_{ij}) = 1$ for all $e_{ij} \in E$. Once DUT is modeled as a DAG, the scheduling objective is to minimize the *program completion time* or maximize the *speed-up* (we will define these terms shortly).

5.2.4.2 Levelization

We are given a netlist represented in DAG in which nodes are already mapped to LUTs and FFs. The objective is to map each node into a suitable instruction word in a HEP processor. If the number of HEP processors is represented by P and the number of available instruction words in each processor is represented by W, then the total number of available instruction words is $P \times W$. In the proposed HEP-based emulation engine where $P = 64$ and $W = 128$, there are total of 8192 (8K) instruction words available. The *instruction memory map*(IMM) of HEP-based emulation engine is shown in Fig. 5.9. The process of assigning nodes to instruction words in IMM is done through subdividing the DUT netlist into slices and allocating nodes in each slice to instruction words. However to preserve functional correctness of the mapped netlist, the slicing of the DAG is subject to the following rules:

- An LUT node must be scheduled to an instruction word no earlier than all the nodes that generate it inputs (i. e. fan-in nodes).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

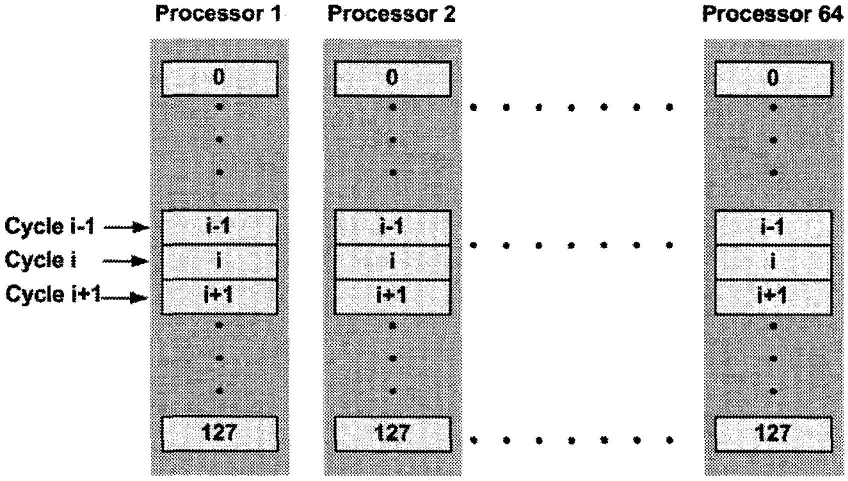


Figure 5.9: Instruction memory map (IMM) of HEP-based emulation system.

- All flip-flip outputs used as feedback inputs are considered as virtual inputs to DUT and must be scheduled prior to all nodes it is driving (i. e. fan-out nodes).
- No two nodes with a common fan-out node should be assigned to the same instruction cycle.

While the first two rules are referred to as *precedence constraints*, the last rule is referred to as *communication constraint*. The problem consists of slicing the DAG into smallest number of partitions so that none of the rules stated above is violated and nodes in each partition are assigned to instruction words in IMM so that the total execution time for all nodes in one partition is minimized. The largest number of partitions allowed is bounded by W (number of available instruction words in each HEP processor).

A straight forward solution for slicing DAG while observing the precedence constraints is obtained through *levelization*. Levelized scheduling orders the nodes with respect to the number of logic stages (i. e. distance) from the inputs. Each node in DAG is labeled with its "level". Primary inputs to the circuits and outputs of flip-flops are given level 0. All other nodes are given a level that is one greater than the maximum level of their fan-in nodes. Such labeling can be done with a simple tree traversal algorithm such as *Depth-First Traversal* (DFT). If nodes are evaluated in level order (all level 1 nodes before all level 2 nodes and so on), then the generated outputs after the last level (corresponding to the primary outputs and flip-flop inputs) will have their correct values.

Two DAG levelization algorithms are known, ASAP and ALAP. *As-Soon-As-Possible* (ASAP) levelization, shown in Fig. 5.10, rearranges each node as soon as all fan-in nodes are levelized. *As-Late-As-Possible* (ALAP) levelization, shown in Fig. 5.12, assigns a node to one level before its

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

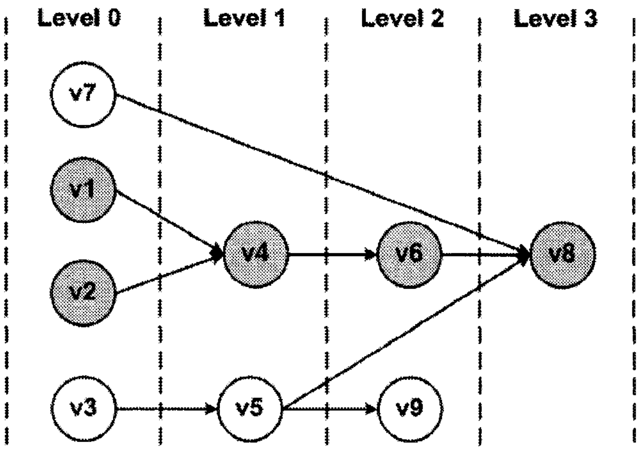


Figure 5.10: ASAP Levelization

output is required. The pseudo codes for both ASAP and ALAP algorithms are shown in 5.11 and 5.13 respectively. The ASAP algorithm starts from the input nodes and moves towards the output nodes while performing "forward" depth-first labeling. The label value assigned by ASAP algorithm to node v_i is represented as $ASAP(v_i)$. Similarly, the ALAP algorithm starts from the output nodes and moves towards the input nodes while performing "backward" depth-first labeling. The label value assigned by ALAP algorithm to node v_i is represented as $ALAP(v_i)$.Using the "parallel programming" analogy on a multi-processor platform where each node (vertex) v_i in TPG represents a single "task", $ASAP(v_i)$ and $ALAP(v_i)$ correspond to the earliest time and latest time that task v_i can start running respectively.

Although ASAP and ALAP levelizations produce correct emulation results that satisfy precedence constraints, they do not create a *balanced processor workload.* Figure 5.14 shows a histogram of processor workload through time (i. e. cycles) while an average-sized netlist, for example "elliptic. blif" ($\leq$ 6200 logic elements), is being emulated. The blue and red lines show the processors activity when the netlist is levelized using ASAP and ALAP algorithm respectively. The ASAP levelization tends to shift most of the processors' workload closer to early cycles while ALAP levelization shifts the workload closer to later cycles. In either case, most HEP processors remain "idle" during intermediate cycles. The peaks on the left and right indicate that many nodes could be scheduled in any instruction cycles. The shapes of these curves are typical of majority of designs especially large ones.

Circuits containing more than 6300 logic elements fail to be scheduled in to the HEP-based emulation engine's IMM if the designs were to be scheduled using either ALAP or ASAP levelization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
$ASAP(G = (V, E))$
  {
    FOR each $v_i \in G$ DO
      IF fanin($v_i$) = $\phi$ THEN
        $v_i \cdot ASAP = 1$;
        $G = G - \{v_i\}$;
      ELSE
        $v_i \cdot ASAP = 0$;
      ENDIF
    ENDFOR
    WHILE $G \neq \phi$ DO
      FOR each $v_i \in G$ DO
        IF all fanin($v_i$) are levelized THEN
          $v_i \cdot ASAP$ = MAX(fanin($v_i$) $\cdot ASAP$) + 1;
          $G = G - \{v_i\}$;
        ENDIF
      ENDFOR
    ENDWHILE
    RETURN;
  }
```

Figure 5.11: ASAP algorithm in pseudo code.

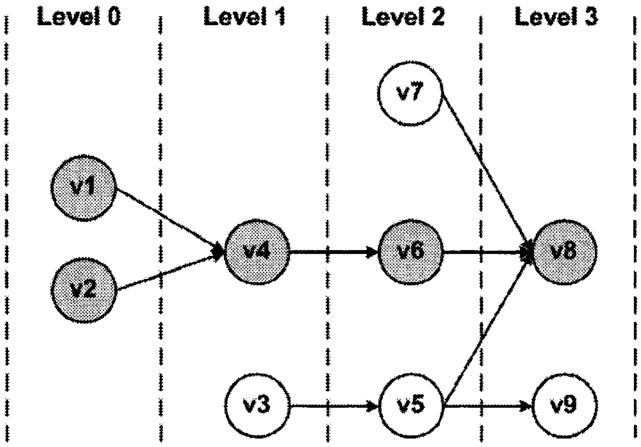


Figure 5.12: ALAP Levelization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
$ALAP(G = (V, E))$
  {
    FOR each $v_i \in G$ DO
      IF fanout($v_i$) = $\phi$ THEN
        $v_i \cdot ALAP = \text{CPL}; \backslash * CPL = CriticalPathLength * \backslash$
        $G = G - \{v_i\}$;
      ELSE
        $v_i \cdot ALAP = 0$;
      ENDIF
    ENDFOR
    WHILE $G \neq \phi$ DO
      FOR each $v_i \in G$ DO
        IF all fanout($v_i$) are levelized THEN
          $v_i \cdot ALAP = \text{MIN(fanout}(v_i) \cdot ALAP) - 1$;
          $G = G - \{v_i\}$;
        ENDIF
      ENDFOR
    ENDWHILE
    RETURN;
  }
```

Figure 5.13: ALAP algorithm in pseudo code.

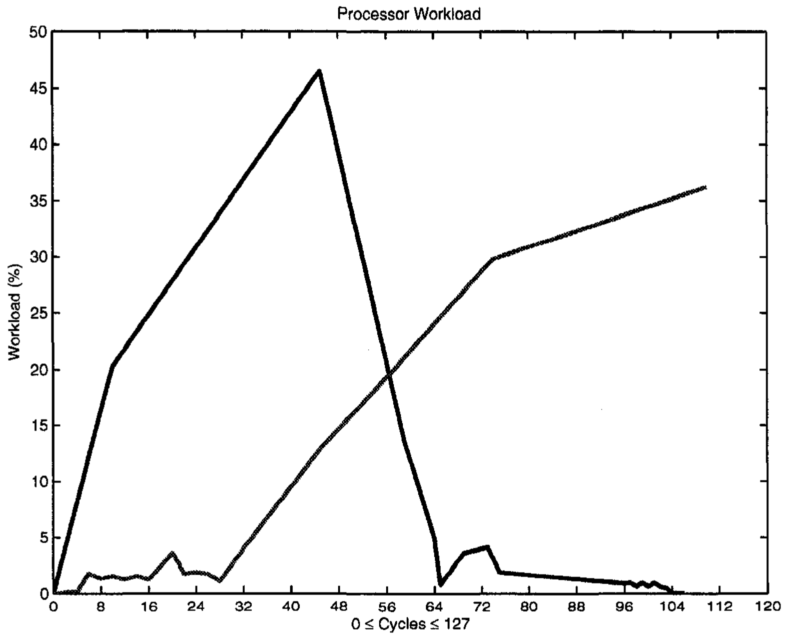


Figure 5.14: Processor workload after levelizing "elliptic". Blue and red lines represents ASAP and ALAP levelization respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

techniques. Hence, a scheduler heuristic should be capable of not only mapping all the circuits into the emulation system but also minimize emulation time by maximizing the *average processor workload* for all 64 processors in the emulation engine.

5.2.4.3 Modified List Scheduling (MLS)

Although ASAP and ALAP levelization algorithms produce correct results there are significant leeway in the partial order for nodes that are not on the *critical path.*

Definition: *In a technology mapped netlist represented by DAG* $G = (V, E)$, *the critical path is the path with maximal length between inputs and outputs.* For example, in Fig. 5.10 the critical path consists of $v_1 \rightarrow v_4 \rightarrow v_6 \rightarrow v_8$. Nodes on critical path are called *critical path node* (CPN), which are shaded in gray color in Fig. 5.10. It is worth mentioning that, based on the definition, it is possible for a circuit to have multiple critical paths. For example in Fig. 5.10, $v_2 \rightarrow v_4 \rightarrow v_6 \rightarrow v_8$ is also a critical path.

To balance processor workload and improve emulation speed, the scheduling tool should be able to identify non-critical path nodes within the DAG and reschedule them effectively into other instruction cycles in order to minimize "the maximum number of instructions". For example, comparing figures 5.10 and 5.12, node v_7 can be moved from level 0 into level 2, while not violating the precedence constraints, to decrease processor's workload in level 0 and increase the processor's workload in level 2, thus balancing workload in both levels.

The scheduling tool introduced in this section uses a variation of *list scheduling*[32] algorithm, originally developed for high-level synthesis. The proposed scheduling algorithm is referred to as *modified list scheduling* or MLS. The pseudo code for MLS is shown in 5.15.

- The first step is to generate ASAP and ALAP levelization of DAG (lines 3-4). As a result the range of levels into which each node can be assigned is determined.

  Lemma: For node $v_i \in V$ if $ASAP(v_i) = ALAP(v_i)$ then v_i is on critical path (i. e. v_i is a CPN). Similarly, v_i is non-CPN if and only if $ALAP(v_i) - ASAP(v_i) \neq 0$ (line 6-12). The length of critical path is denoted as C_L and $C_L = Max(ALAP(v_i))$ for all $v_i \in V$ (line 5).

  Observation 1: Any circuit C, represented by graph $G = (V, E)$, will require at least C_L cycles to be emulated on any parallel processing platform. The ultimate goal for any scheduling heuristics is to reduce the number of emulation cycles (=emulation time) closer to C_L.

- The MLS iterates (line 13-41) through levels, starting from level 0 to maximum of C_L ($0 \leq L_j \leq C_L$). At each level (L_j), all "ready-to-schedule" nodes are sorted in ascending order

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
MLS($G = (V, E)$)
  {
    ASAP($G = (V, E)$);
    ALAP($G = (V, E)$);
    $C_L = MAX(v_i \cdot ALAP)$;
    FOR each $v_i \in G$ DO
      IF $v_i \cdot ALAP - v_i \cdot ASAP = 0$ THEN
        $v_i$ is CPN;
      ELSE
        $v_i$ is non-CPN;
      ENDIF
    ENDFOR
    FOR $L_j = 0$ TO $C_L$ DO
      $V' = \phi$;
      FOR all $v_i \cdot ASAP \geq L_j$ DO
        $v_i \cdot MOB = v_i \cdot ALAP - L_j$;
        $V' = V' + \{v_i\}$;
      ENDFOR
      $V'$ = SORT($V'$,"ascending mobility");
      Max_Cycle=Min_Cycle=0;
      WHILE $V' \neq \phi$ DO
        IF $v_i \in CPN$ THEN
          allocate_and_collapse_IMM($v_i$, Max_Cycle, Min_Cycle);
          $V' = V' - \{v_i\}$;
        ENDIF
      ENDWHILE
      WHILE $V' \neq \phi$ DO
        IF $v_i \cdot MOB = 0$ THEN
          allocate_and_collapse_IMM($v_i$, Max_Cycle, Min_Cycle);
          $V' = V' - \{v_i\}$;
        ENDIF
      ENDWHILE
      WHILE $V' \neq \phi$ DO
        $v_i = HEAD(V', random)$; \*randomly select $v_i$*\
        IF allocate_and_collapse_IMM($v_i$, Max_Cycle, Min_Cycle) successful THEN
          $V' = V' - \{v_i\}$;
        ELSE
          leave $v_i$ for next iteration and do nothing;
        ENDIF
      ENDWHILE
    ENDFOR
    RETURN;
  }
```

Figure 5.15: MLS algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with respect to their "mobility". In other words, nodes are prioritized with respect to their mobility, so that a node with the lowest mobility has the highest priority.

Definition: Node v_i is "ready-to-schedule" if $ASAP(v_i) \leq L_j$ and v_i has not yet been allocated into a word inside IMM.

Definition: For node v_i, "Mobility" is calculated as $MOB(v_i) = ALAP(v_i) - L_j$. In other words, the mobility of node v_i determines how many levels the node can be "postponed" for scheduling.

Sorting the nodes in ascending order with respect to their mobility, virtually, categorizes all "ready-to-schedule" nodes into three subclasses:

- critical path nodes: At level L_j, any ready-to-schedule node (v_i) that belongs to critical path will have a mobility of 0 ($MOB(v_i) = ALAP(v_i) - L_j = ALAP(v_i) - ASAP(v_i) = 0$).
- semi-critical nodes: A ready-to-schedule node (v_i) is a semi-critical node if it is neither on critical path nor can be "postponed" (i. e. moved) to later levels ($L_{j+1}, \cdots$) either, because $L_j = ALAP(v_i)$. For such nodes $MOB(v_i) = 0$ as well.
- postponable node: Node v_i is postponable if $MOB(v_i) \neq 0$.

- At each level (L_j) once all ready-to-schedule nodes are identified they are sorted and prioritized with respect to their mobility (line 15-19). First "all" the critical path nodes (in level L_j) are allocated into IMM (line 21-26). Next, "all" the semi-critical nodes will also be allocated into the IMM (line 27-32). And, finally, the algorithm tries to allocate postponable nodes into IMM, by selecting a node from a list with least mobility. If two postponable nodes have same mobility the algorithm will select one node ***randomly*** (line 33-40). Note that all nodes are allocated to IMM while observing the communication constraint.

- At each iteration, if "allocate_and_collapse_IMM()" function fails to allocate a postponable node to IMM, the node will be moved to next level (L_{j+1}).

The pseudo code illustrated in Fig. 5.15 explains the main steps involved in MLS algorithm. However, to avoid confusion in the code we excluded the details of steps during "allocate_and_collapse_IMM()" function calls which we will describe below.

- The main objective of "allocate_and_collapse_IMM()" is to *collapse* those nodes that satisfy the communication constraint. Collapsible nodes can be allocated into the same instruction cycle (but on separate HEP processors). Figure 5.16 illustrates how collapsing two nodes could reduce length of emulation program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

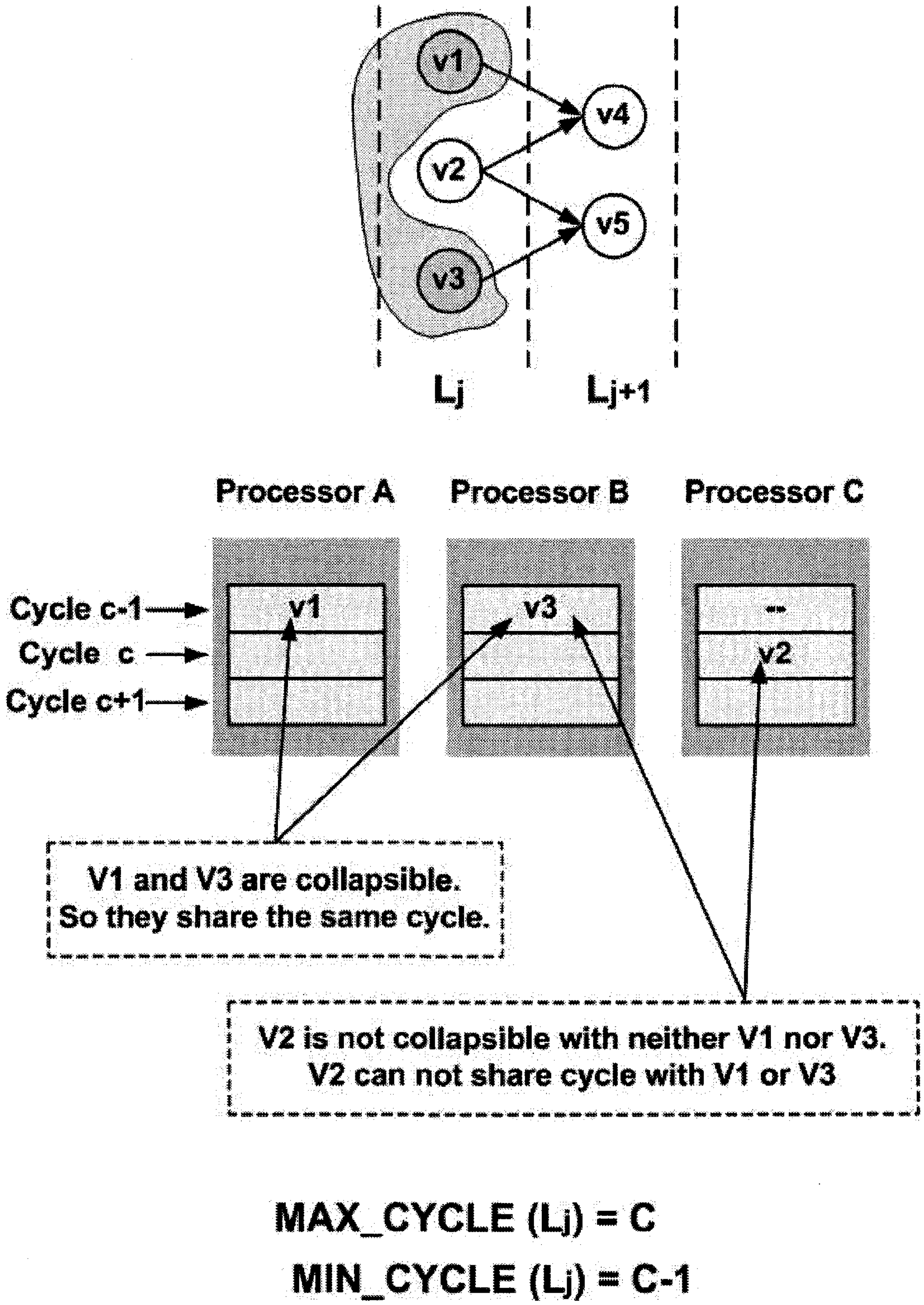

v1
v2
v3
v4
v5
Lj
Lj+1
Processor A
Processor B
Processor C
Cycle c-1
Cycle c
Cycle c+1
v1
v3
--
v2
V1 and V3 are collapsible.
So they share the same cycle.
V2 is not collapsible with neither V1 nor V3.
V2 can not share cycle with V1 or V3
MAX_CYCLE (Lj) = C
MIN_CYCLE (Lj) = C-1

Figure 5.16: Examples of collapsing nodes during IMM allocation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- At each level L_j, the algorithm tries to collapse critical path nodes into the same instruction cycles. If two critical path node are not collapsible the algorithm will allocate the nodes into two different instruction cycles. The algorithm keeps track of minimum and maximum instruction cycles occupied by all mutually non-collapsible CPNs in level L_j. The cycle numbers are referred to Min_Cycle(L_j) and Max_Cycle(L_j). In the example shown in Fig. 5.16, $Max\_Cycle(L_j) = C$ and $Min\_Cycle(L_j) = C - 1$. If a non-collapsible CPN is to be added to IMM, that node is allocated to level Max_Cycle(L_j)+1 and Max_Level(L_j) will be updated automatically. To initiate collapsing and allocating nodes, the MLS algorithm sets both Min_Cycle and Max_Cycle to 0 (Line 20).

- MLS allocates and collapses semi-critical nodes the same way it treats CPNs. The only difference is that now the $Max\_Cycle(L_j) \neq Min\_Cycle(L_j)$. In such case, the algorithms tries to fit the nodes in between cycles Max_Cycle(L_j) and Min_Cycle(L_j). If no suitable cycles were found then Max_Cycle(L_j) is incremented by 1.

- At the final step, MLS starts allocating postponable nodes. However this time MLS will start searching to find free instruction word in IMM "only" within the range between Max_Cycle(L_j) and Min_Cycle(L_j). If the node could not fit within that range then the node is moved to next level (L_{j+1}).

It is worth indicating that before MLS starts the scheduling process it initializes all the instruction words in IMM by filling them all with "NOP" instruction. At the end of scheduling, those instruction words in IMM to which no node has been assigned are left intact (="NOP" instructions).

As we will discuss later, the ratio of used instruction words with respect to number of "NOP" instructions (i. e. processor idle time) in one HEP processor is the most important evaluation metrics for comparing scheduling algorithms. Any optimization technique that could improve such ratio is highly desirable.

5.2.4.4 MLS+BFF Scheduling

Task scheduling for a multi-processor platform is an NP-complete problem, for which no optimal solution exists. Although MLS scheduling produces close to optimal solution in a reasonable amount of time we could still apply some optimization techniques that might further improve the the scheduling result. The improvement over MLS algorithm that is explained below results in an increase in *average processor workload* or reduction of processor idle time which, in turn, reduces emulation time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As mentioned earlier, at each level, the MLS algorithm prioritizes the circuit nodes according to their mobility and assigns higher priority to CPNs or semi-critical nodes over postponable nodes. However, it does not distinguish postponable nodes with "equal" mobility. In such cases, the MLS algorithm will randomly selects a node for collapsing and allocation into IMM.

The problem with such scheme is that the algorithm does NOT make any "prediction" about the signal flow within DAG. Lack of such prediction capability results in more frequent failures in collapsing and allocating postponable nodes, as these nodes are accumulated into later cycles.

An intuitive improvement to MLS is explained through the following example. As illustrated in Fig. 5.17, node v_1 driving the inputs to two other nodes v_2 and v_3. In other words, v_1 has the "fan-out" degree of 2. Obviously, if $ASAP(v_1) = L$ then $ASAP(v_2) = ASAP(v_3) = L + 1$. Similarly node v_4, also with $ASAP(v_4) = L$ has a fan-out degree of 3 (driving nodes v_5,v_6,v_7). If during MLS scheduling both nodes v_1 and v_4 were identified as postponable nodes, the algorithm will choose either nodes randomly as the next candidate for scheduling. However, if v_4 was selected first over v_1, then input values to three nodes (i. e. v_5,v_6,v_7) will be calculated earlier without being postponed to later iterations. This means that three HEP processors that emulate v_5, v_6, and v_7 would have less "waiting" time to have their inputs ready. In this sense, v_4, with fan-out degree of 3 would be preferred over v_2 (with fan-out of 2) simply because v_4 *keeps less number of HEP processors waiting.* Based on the above example, an improved scheduling algorithm introduced here is referred to as "modified list scheduling with biggest fan-out first" or shortly MLS+BFF[4]. Figure 5.18 explains the MLS+BFF algorithm in pseudo code. MLS+BFF algorithm performs identically to MLS algorithm except when it tries to schedule postponable nodes. For such nodes, MLS+BFF will further sort (i. e. prioritize) all the postponable nodes with equal mobility with respect to their "fan-out degrees", so that nodes with greater fan-out will have higher priority over nodes with same mobility and less fan-out (line 35-36).

The results obtained by MLS+BFF scheduling algorithm shows improvements in average processor workload, as we will see shortly. Such improvement is solely obtained due to the fact that, at each iteration, MLS+BFF is capable of "predicting" the processors workload in next iteration by profiling signal flow of the circuit.

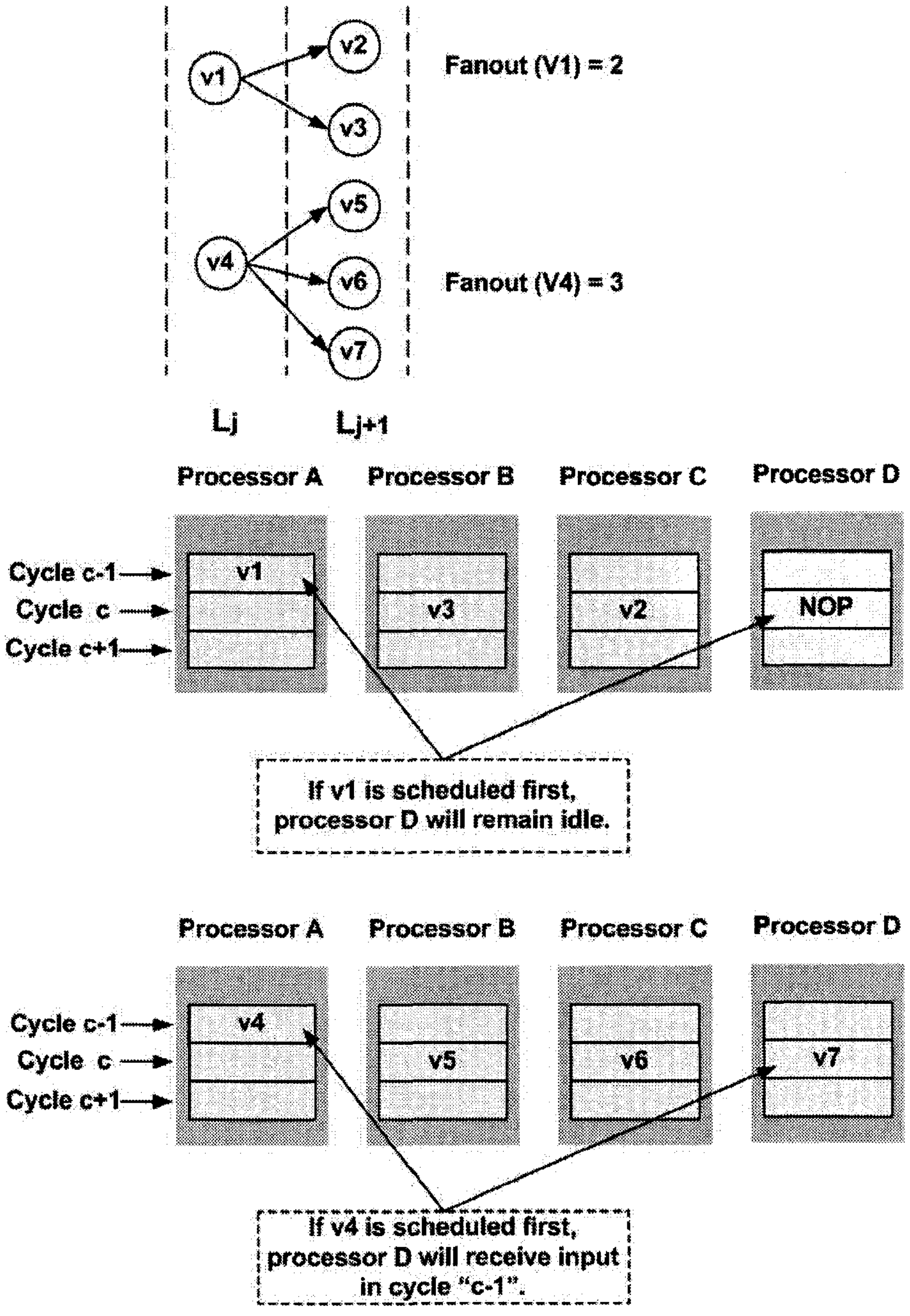
5.2.4.5 Mathematical Formulation

To be able to compare the results with previous work, first we should establish the mathematical foundations. The formulation of the scheduling problem along with the evaluation metrics are

---

[4] I could not find a shorter descriptive name.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



If v1 is scheduled first, processor D will remain idle.

If v4 is scheduled first, processor D will receive input in cycle "c-1".

Figure 5.17: Prioritizing nodes with equal mobility with respect to their fan-out degree.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
$MLS + BFF(G = (V, E))$
  {
    $ASAP(G = (V, E))$;
    $ALAP(G = (V, E))$;
    $C_L = MAX(v_i \cdot ALAP)$;
    FOR each $v_i \in G$ DO
      IF $v_i \cdot ALAP - v_i \cdot ASAP = 0$ THEN
        $v_i$ is CPN;
      ELSE
        $v_i$ is non-CPN;
      ENDIF
    ENDFOR
    FOR $L_j = 0$ TO $C_L$ DO
      $V' = \phi$;
      FOR all $v_i \cdot ASAP \geq L_j$ DO
        $v_i \cdot MOB = v_i \cdot ALAP - L_j$;
        $V' = V' + \{v_i\}$;
      ENDFOR
      $V'$ = SORT($V'$,"ascending mobility");
      Max_Cycle=Min_Cycle=0;
      WHILE $V' \neq \phi$ DO
        IF $v_i \in CPN$ THEN
          $allocate\_and\_collapse\_IMM(v_i, Max\_Cycle, Min\_Cycle)$;
          $V' = V' - \{v_i\}$;
        ENDIF
      ENDWHILE
      WHILE $V' \neq \phi$ DO
        IF $v_i \cdot MOB = 0$ THEN
          $allocate\_and\_collapse\_IMM(v_i, Max\_Cycle, Min\_Cycle)$;
          $V' = V' - \{v_i\}$;
        ENDIF
      ENDWHILE
      WHILE $V' \neq \phi$ DO
        $V'$ = SORT($V'$,"descending fanout");
        $v_i = HEAD(V')$;
        IF $allocate\_and\_collapse\_IMM(v_i, Max\_Cycle, Min\_Cycle)$ successful THEN
          $V' = V' - \{v_i\}$;
        ELSE
          leave $v_i$ for next iteration and do nothing;
        ENDIF
      ENDWHILE
    ENDFOR
    RETURN;
  }
```

Figure 5.18: MLS+BFF algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

presented below [28]. Let C be the technology-mapped design to be scheduled. We will represent C by a directed graph, G, in which each logic element (LUT/FF) is represented by a vertex (node) in the graph. The directed graph G is shown as $G = (V, E)$, where V is the set of all vertices and E is the set of uni-directional edges hence:

- each $v_i \in V$ represent a logic element in C for $1 \leq i \leq |V|$;
- each $(v_i, v_j) \in E$ represents a directed wire from logic element i to logic element j in C. In this case v_i is "fan-in" node of v_j. And, v_j is "fan-out" node of v_i;
- The graph $G' = (V, E')$ is the *acyclic flow graph* of $G = (V, E)$ where $E' \subseteq E$ obtained by depth first search starting from both LUT vertices with zero fan-in or fed-back Flip-flop outputs.

Static task scheduling is a NP-complete problem for which heuristic solutions is required. One method for obtaining acceptable solutions is to formulate the scheduling problem using *Integer Programming* (IP).

Definition: A binary variable $x_{i,j}$ is associated with each $v_i \in V$ in G' where:

- $x_{i,j} = 1$ iff the logic element i, represented by v_i, is scheduled in cycle j;
- $x_{i,j} = 0$ otherwise.

Let the earliest and latest cycles in which a vertex v_i can be scheduled be $E(i)$ and $L(i)$, respectively[5].

Definition: The *scheduling interval* of vertex v_i is defined as the set of integers $S(i) = \{E(i), E(i) + 1, \cdots, L(i)\}$. The longest path in DAG is called *critical path* and is denoted by CP. The length of the critical path (i. e. number of nodes on critical path) is shown as $C_L = |CP|$. Obviously, the overall scheduling interval for every v_i will be $S(i) = \{1, \cdots, C_L\}$.

Assignment Constraint: In order to have a correct scheduling solution, it is imperative that each vertex in DAG be scheduled for only one cycle in its scheduling interval. In other words:

$$\sum_{j \in CP} x_{i,j} = 1, \forall v_i \in V \tag{5.1}$$

Precedence Constraint: It is also imperative to observe the two precedence constraints mentioned before to guarantee the correct scheduling. Mathematically speaking:

$$\sum_{j_2 \leq j} x_{i_2,j_2} + \sum_{j_1 > j} x_{i_1,j_1} \leq 1, \forall (v_{i_1}, v_{i_2}) \in E', v_{i_1}, v_{i_2} \in V, \forall j \in \{S(i)\}. \tag{5.2}$$

---

[5] Obviously, $ASAP(v_i) = E(i)$ and $ALAP(v_i) = L(i)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Resource Constraint: At every level, we must ensure that there are enough computing resources, in the form of instruction words in IMM, to map the prioritized vertices into IMM.

$$\sum_{\forall v_i \in levelL} x_{i,j} \leq 64 \times (Max\_Cycle(L) - Min\_Cycle(L)), \forall j \in LevelsL+1, L+2, \cdots, C_L \tag{5.3}$$

It is obvious that by scheduling the closest element of the DAG to outputs as early as possible, a minimum number of instruction cycles needed to emulate the entire design can be achieved. The logic elements of the design immediately connected to primary outputs are represented by vertices without successors in G'. We will ignore all the vertices of G' that have one or more successors and consider only the vertices without successors for cycle minimization in the following manner:

$$min \sum_{j \in S(i) j \cdot x_{i,j}}, \forall v_i \in Vwithoutsuccessors. \tag{5.4}$$

5.2.4.6 Evaluation Metrics

The efficiency of an algorithm that targets the problem of task scheduling for parallel processing platform can be measured in various ways. We will explain the definition and mathematical formulation for each evaluation metrics in this subsection. The results obtained by the scheduling algorithms are explained later in this chapter.

Minimum emulation time: An HEP-based consists of $P \times W$ processing elements (= total number of words in IMM), where P is the number of emulation processors and W is depth (size) of HEP's control memory. Hence, if circuit C represented by $G = (V, E)$ was to be emulated on HEP-based emulation system, the theoretical lower bound for emulation time (delay) D_{min} is calculated as:

$$C_L \leq D_{min} = \left\lceil \frac{|V|}{P} \right\rceil. \tag{5.5}$$

Processor Workload and Idle Time: Let's assume that program T consists of total of M tasks that are to be executed using single processor (e. g. P_1) is represented by $T = \{T_{P_1,1}, T_{P_1,2}, \cdots, T_{P_1,M}\}$. The execution time of task T_i on one processor is shown as E_{T_i}. Thus the execution time of program T is:

$$E_{total,P_1} = E_{T_1,P_1} + E_{T_2,P_1} + \cdots + E_{T_M,P_1} = \sum_{T_1 \leq T_i \leq T_M} E_{T_i,P_1} \tag{5.6}$$

However, if program T is to be executed on a parallel-processor platform, execution of tasks will be delayed due to communication overhead and inter-task dependencies. The execution graph for program T is illustrated in Fig. 5.19. In such case, the total execution time of program T will be prolonged by the total delay time:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

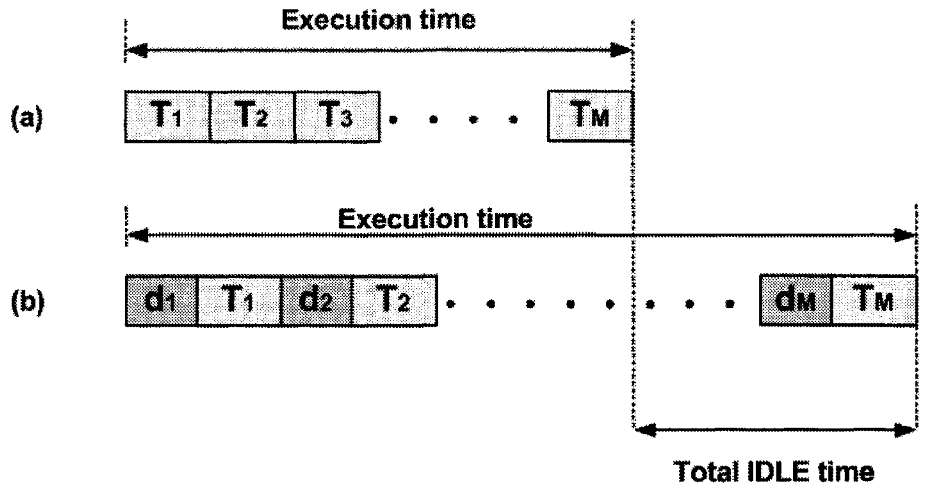


Figure 5.19: Executing program on a parallel platform. (a) executing program on single processor (no delay between tasks). (b) executing program on a parallel processor.

$$\sum_{T_1 \le T_i \le T_M} E_{T_i,P_1} + \sum_{i \le M} \delta_i \tag{5.7}$$

The second term in the above equation, $(\sum_{i \le M} \delta_i)$, is usually referred to as *processor idle time* (i. e. time during which processor is not executing anything). "Processor workload",ϕ, is the ratio of time during which a processor is "busy" executing tasks with respect to the total execution time:

$$\phi_{p_1} = \frac{\sum\limits_{T_1 \le T_i \le T_M} E_{T_i,P_1}}{\sum\limits_{T_1 \le T_i \le T_M} E_{T_i,P_1} + \sum\limits_{i \le M} \delta_i} \tag{5.8}$$

A good scheduling tool for a parallel processing platform thrives on maximizing workload for each and every processor in the system, as well as, balancing the workload among all processors. Also, the scheduler should minimize the total processor idle time. Based on the above formulation, the *average processor workload* $(\overline{\phi})$ is defined as:

$$\overline{\phi} = \frac{\sum\limits_{1 \le i \le 64} \phi_{P_i}}{64} \tag{5.9}$$

To achieve acceptable balance of workload among processors the following relation should hold:

$$\overline{\phi} \approx \phi_{P_i} \tag{5.10}$$

Speed-up: The speed-up is defined as the time required for sequential execution of a program divided by the time required for parallel execution. The amount of speed-up is measured according to the number of cycles (rather than time). The speed-up is denoted by λ.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Execution Delay: Execution delay is the defined as the amount of time that execution of a task is delayed (postponed). In this application, the earliest time that task v_i can be executed is determined by $ASAP(v_i)$. If task v_i is executed at level L, then the delay for task v_i is $\Delta = L - ASAP(v_i)$.

5.2.4.7 Implementation of MLS/MLS+BFF Scheduling Tools

For the purpose of this research a software tool in "C" language called "GSchedule" on Unix/Linux platform has been developed[6]. Source listings for "GSchedule" is provided in a CD-ROM accompanying this thesis. The following command line illustrates how the tool is run against MCNC benchmark circuits:

```
$ GSchedule [-BFF] netlist_name.blif
```

The GSchedule schedules a technology mapped netlist (in BLIF format) using MLS algorithm and presents the results on standard output. The [-BFF] option makes the tool to use MLS+BFF algorithm.

We have used dynamic memory allocation and linked-lists to implement the data structure used in GSchedule to minimize memory usage by the tool. Each node in DAG, is a "C" structure consists of several fields such as name, fan-in list, fan-outs degree, and ASAP/ALAP level numbers. The GSchedule builds a netlist of such node structure by parsing the input BLIF netlist.

Once the scheduling is finished, the GSchedule will generate the emulation program for each and every 64 HEP processors in the emulation engine. A sample snapshot of the output generated by GSchedule is shown in Fig. 5.20. Notice that node names in each column represent the instruction words will be downloaded into each HEP's control memory.

5.2.5 Experimental Results

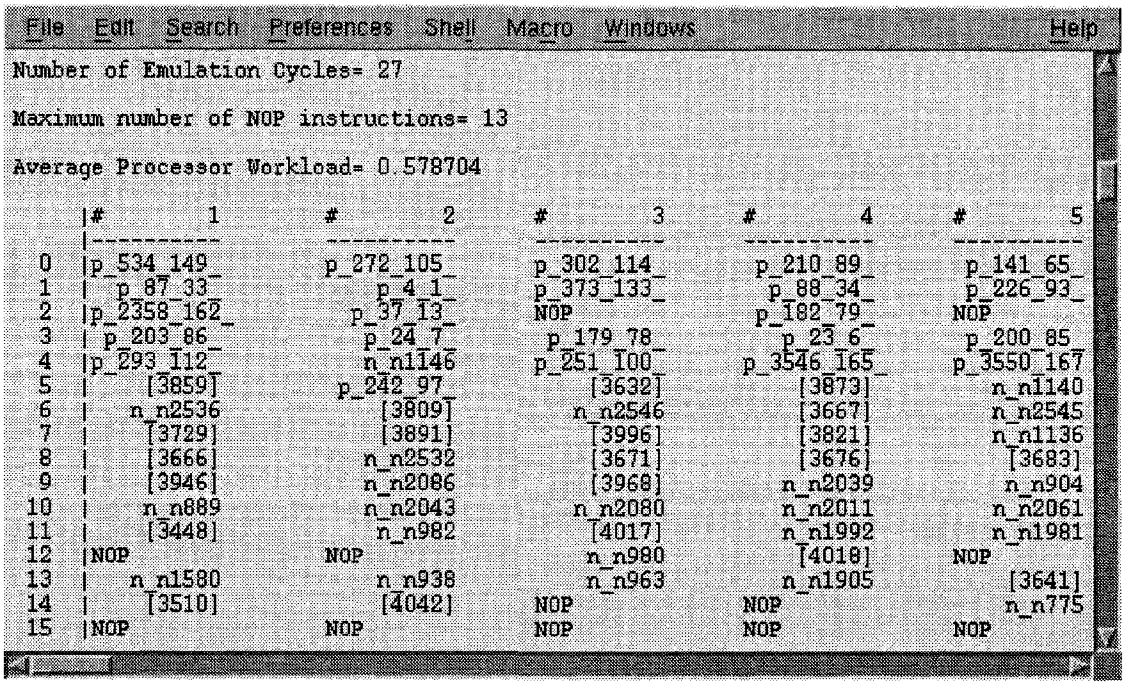
In this section the results obtained by the scheduling tools, MLS and MLS+BFF, are presented. The tools were tried on almost all circuits in MCNC benchmark suite. However, we will only present the results for the 10 biggest circuits.

- Table 5.3 illustrates both the average ($\overline{\phi}$) as well as maximum HEP processor workload. As the results show the MLS scheduling has managed to achieve total average processor workload of 83.9% while the deviation of workload among processors is less than 3%. That means, during the emulation process, the workload is evenly distributed among all 64 HEP processors in the

---

[6] The source listings consists of approximately 4000 lines of codes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



```
Number of Emulation Cycles= 27

Maximum number of NOP instructions= 13

Average Processor Workload= 0.578704

    |#        1    #        2    #        3    #        4    #        5
    |----------    ----------    ----------    ----------    ----------
  0 |p_534_149_    p_272_105_    p_302_114_     p_210_89_     p_141_65_
  1 |  p_87_33_        p_4_1_    p_373_133_      p_88_34_     p_226_93_
  2 |p_2358_162_     p_37_13_    NOP            p_182_79_    NOP
  3 | p_203_86_       p_24_7_     p_179_78_       p_23_6_     p_200_85_
  4 |p_293_112_       n_n1146    p_251_100_   p_3546_165_    p_3550_167
  5 |    [3859]     p_242_97_        [3632]        [3873]       n_n1140
  6 |   n_n2536        [3809]       n_n2546        [3667]       n_n2545
  7 |    [3729]        [3891]        [3996]        [3821]       n_n1136
  8 |    [3666]       n_n2532        [3671]        [3676]        [3683]
  9 |    [3946]       n_n2086        [3968]       n_n2039        n_n904
 10 |    n_n889       n_n2043       n_n2080       n_n2011       n_n2061
 11 |    [3448]        n_n982        [4017]       n_n1992       n_n1981
 12 |NOP           NOP                n_n980        [4018]    NOP
 13 |   n_n1580        n_n938        n_n963       n_n1905        [3641]
 14 |    [3510]        [4042]    NOP           NOP                n_n775
 15 |NOP           NOP           NOP           NOP           NOP
```

Figure 5.20: Example of output generated by GSchedule tool. Each column represents the emulation instructions executed by one processor.

emulation system. In some cases the MLS scheduling has achieved almost optimal scheduling solution (99.4%). Also, as shown in the table, the total processor idle time is less than 9 cycles in average.

Table 5.4 represents same statistics about the sequential binary multiplier circuit[7] example. In case of very small circuits (such as binary multiplier) the statistics show that most processing resources in the HEP-based emulation system remains under utilized. Hence the average processor workload for such sparse circuits is considerably lower.

- Table 5.5 illustrates how the MLS+BFF optimization algorithm has not only increased the average processor workload but also has reduced the average processor idle time in at least half of the test cases. Such increase in the average processor workload is reported to be between 0.7 – 6.2%, with an average value of +1.5%. Also the reduction in processor idle time is between 1-3 cycles, with an average value of 1.2 cycles. It is worth emphasizing that MLS+BFF scheduling tool does not create a significant improvement in small circuits such as the binary multiplier example.

---

[7] Binary multiplier does not belong to MCNC benchmark suite. So we decided to present the results for that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.3: Processor workload calculated after MLS scheduling.

DUT	ϕ_{min} (%)	$\overline{\phi}$ (%)	**Deviation** (%)	**Avg. Idle Time (cyc.)**
s38417	93.3	95.3	2	5
s38584	97.2	97.9	0.7	3
s35932	98	99.4	1.4	1
frisc	91.8	93.7	1.9	8
elliptic	96	96.8	0.8	4
pdc	85.2	87.2	2	16
des	92.7	95.4	2.7	4
i10	53.3	57.7	4.4	20
C7552	51.6	58.5	6.9	13
C5315	51.8	57.8	6	12
TOTAL	81.1	83.9	2.8	8.6

Table 5.4: Processor workload after MLS scheduling on multiplier.

DUT	ϕ_{min} (%)	$\overline{\phi}$ (%)	**Deviation** (%)	**Avg. Idle Time (cyc.)**
Multiplier	7.1	12	4.9	12

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.5: Processor workload after MLS+BFF scheduling.

DUT	**MLS** $\bar{\phi}(\%)$	**MLS+BFF** $\bar{\phi}(\%)$	**Improvement** (%)	**MLS Avg. Idle Time**	**MLS+BFF Avg. Idle Time**	**Reduction** (%)
s38417	95.3	95.3	0	5	5	0
s38584	97.9	97.9	0	3	3	0
s35932	99.4	99.4	0	1	1	0
frisc	93.7	96.1	2.4	8	5	3
elliptic	96.8	99.7	2.9	4	3	1
pdc	87.2	87.9	0.7	16	14	2
des	95.4	98.2	2.8	4	1	3
i10	57.5	57.5	0	20	20	0
c7552	58.5	64.7	6.2	13	10	3
c5315	57.8	57.8	0	12	12	0
TOTAL	83.9	85.4	1.5	8.6	7.4	1.2

- The emulation time for ten biggest circuits when the designs are scheduled by both MLS and MLS+BFF are shown in Tables 5.6 and 5.7 respectively. Last two columns of each algorithm show the total emulation time when the HEP-based emulation engine is implemented on Virtex-II and Virtex-4 family of FPGAs. As it is shown in tables, an HEP-based emulation system is capable of emulating the largest circuit (i. e. "frisc. blif") in 3.58 – 5.59μS if the circuit is scheduled by MLS+BFF algorithm. Also the amount of speed-up obtained by each algorithm is reported for each circuit. As the results show the average speed-up gained by MLS algorithm is $\overline{\lambda} = 50.4$,while the average speed-up gained by MLS+BFF is $\overline{\lambda} = 51.3$.

- The time complexity of MLS and MLS+BFF algorithms to perform ASAP and ALAP levelization on circuit C, denoted by $G = (V, E)$, is $O(2|V| + 2|E|)$. Assuming that there are total average of $|\overline{V}|$ nodes at each level, prioritizing and allocating nodes to 64 processors will have the time complexity of $O(64 \cdot |\overline{V}| \log |\overline{V}|$. Hence the total time complexity of MLS (and MLS+BFF) algorithm is $O(2|V| + 2|E|) + O(64 \cdot |\overline{V}| \log |\overline{V}|$. Both scheduling tools were run in Linux environment on a personal computer with an Intel Pentium 2.8GHz processor. The scheduling tools managed to schedule most test circuits in less than 1 hour. Average execution time for purely combinatorial circuits such as "C7552" is less than 3 minutes. Also, the ex-

---

separately.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.6: Emulation time and speed-up obtained by MLS scheduling.

DUT	MLS				
	D_{min}	**Emulation Inst. Cycles**	**Speed-up** λ	**Virtex II** (μS)	**Virtex 4** (μS)
s38417	85	90	60.1	4.19	2.69
s38584	104	108	61.3	5.03	3.22
s35932	81	86	59.8	4.01	2.57
frisc	116	123	59.8	5.73	3.67
elliptic	97	102	60.6	4.75	3.04
pdc	107	122	55.7	5.68	3.64
des	62	69	57.3	3.21	2.06
i10	22	45	31.1	2.09	1.34
c7552	15	31	29.2	1.44	0.92
c5315	13	27	29.7	1.2	0.8
Multiplier	2	14	7.1	0.65	0.41

Table 5.7: Emulation time and speed-up obtained by MLS+BFF scheduling.

DUT	MLS+BFF				
	D_{min}	**Emulation Inst. Cycles**	**Speed-up** λ	**Virtex II**	**Virtex 4**
s38417	85	90	60.1	4.19	2.69
s38584	104	108	61.3	5.03	3.22
s35932	81	86	59.8	4.01	2.57
frisc	116	120	61.3	5.59	3.58
elliptic	97	99	62.5	4.61	2.96
pdc	107	121	56.1	5.64	3.61
des	62	67	59.1	3.12	2.0
i10	22	45	31.1	2.09	1.3
c7552	15	28	32.3	1.30	0.83
c5315	13	27	29.7	1.25	0.80
Multiplier	2	14	7.1	0.65	0.41

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

periments show that the optimization technique introduced by MLS+BFF causes no overhead on design compilation time due to the fact that the improvement is made by adding local conditions to MLS algorithm. Hence the execution time of MLS+BFF algorithm is identical to execution of MLS algorithm.

5.2.6 Code Generation and Download

The last steps in the proposed CAD flow (Fig. 5.2) are code generation and downloading. Once the scheduling tool generated the memory map for each HEP processor, the instruction words will be filled with mnemonic names of nodes in the netlist. The task of code generation consists of replacing the mnemonic names with actual executable binary op-codes for HEP processors. The code generator will replace the unused instruction words in IMM with binary code for "NOP" instruction. Similarly, if the mnemonic represents an LUT or flip-flop output, it will be replaced by "LUTOP" and "RAMREF" instructions respectively. The "ROMREF" instructions are used when corresponding flip-flop contains an initial value of non-zero.

Once the whole IMM is parsed and binary code representing each instruction word is generated the generated bit-stream can be downloaded into the HEP processors' control memories through "download manager" module on the emulation system. As it is shown in Fig. 5.2 once the binary codes are downloaded into HEP-based emulation system the design is ready to be emulated.

5.3 Comparison and Conclusion

In this chapter a CAD framework for design compilation targeting HEP-based emulation systems has been proposed. As a part of this proposal, two scheduling algorithms called MLS and MLS+BFF were introduced and developed. The tools were run on 10 biggest circuits from MCNC benchmark suite. As a result of scheduling algorithms, the HEP-based emulation system can emulate the biggest test circuit in less than $6\mu S$.

Table 5.8 compares the emulation time of ten circuits on HEP-based emulation system with those reported by VEGA architecture[40]. The author of [40] has reported the results for four of sample circuits that have been used in this study. The results show, the HEP-based emulation system has 4-5 times faster emulation speed. However, it should be emphasized that the ASIC-based emulation processors used in VEGA architecture were fabricated using CMOS $1.2\mu m$ fabrication technology where as Virtex-2 and virtex-4 are fabricated using $0.15\mu m$ and $0.09\mu m$ technologies respectively.

The MLS and MLS+BFF algorithms create close to optimum scheduling solutions especially for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 5.8: Comparing emulation time of HEP and VEGA

DUT	**size**	**MLS/MLS+BFF** μS	**VEGA** μS
s38417	5411	4.19-4.19	21.7
s38584	6630	5.03-5.03	23
pdc	6796	5.68-5.64	25.6
i10	1401	2.09-2.09	23.5

large circuits. In fact, the empirical results show that, as circuits become denser the utilization of processing elements increases which is on the contrary to the results obtained by similar FBE systems. In FBE systems, as the DUT size increases as long as there are enough logic elements and I/O pins available in the target FPGA chips. However, due to Rent's rule, significant FPGA's logic capacity remains under-utilized. If the size of the circuit increases beyond effective logic capacity of FPGAs then multiple FPGA devices will be required. In that case the log utilization in FPGA modules will drop as it is shown, conceptually, by the dotted red curve in Fig. 5.21. Also, The FPGA logic utilization hardly reaches above 80%. In Fig. 5.21 the blue curve represents the percentage of processing resources used with respect to the design size in a HEP-based logic emulation system which illustrates better resource utilization with respect to FBEs. Obviously, robustness of MLS/MLS+BFF scheduling algorithms against bigger size circuits is a great advantage over similar tools. However it should be emphasized that the curve shown for FBE systems is conceptually correct but values are not accurate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

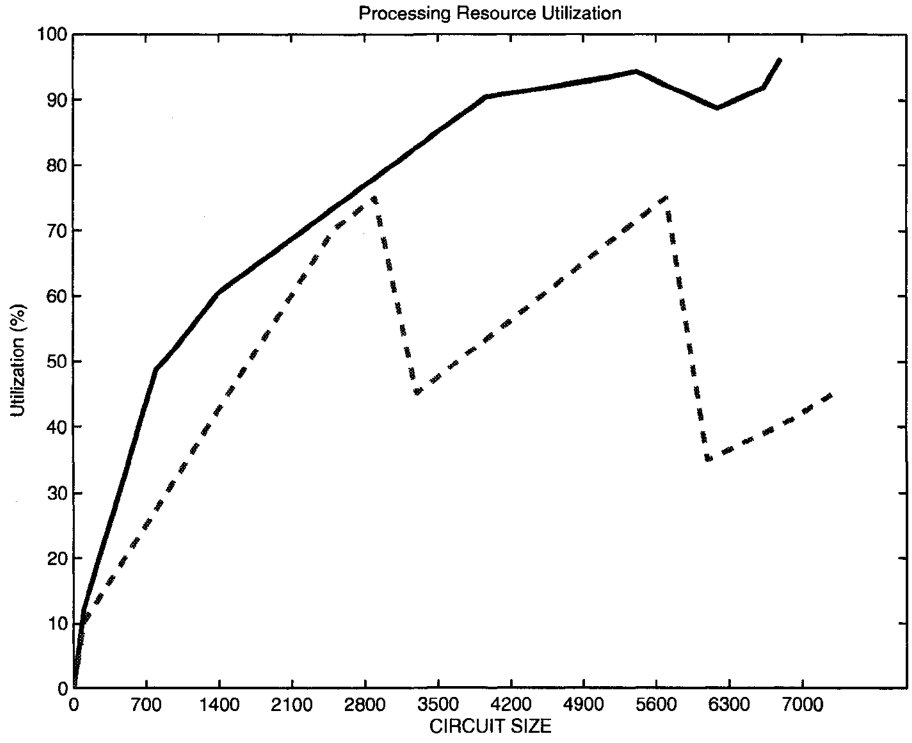


Figure 5.21: Resource utilization in HEP-based emulation system and FBEs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Conclusions and Future Work

The contributions made by this research can be classified in two sections. First, this work has presented the design of a specialized processor called hybrid-emulation processor (HEP) that can be easily implemented on any FPGA platform. A collection of 64 HEP processors were embedded into Xilinx FPGA devices to build a logic emulation engine. The emulation engine is capable of emulating the functionality of digital circuits as large as 160000 logic gates and flip-flops. While relatively simple in architecture, it can emulate a design at speeds of up to $262KHz$. The embodiment of 64 HEP processors requires only one or two of-the-shelf FPGA modules. Such small hardware reduces the cost of HEP-based emulation system by orders of magnitude with respect to its commercial counterparts. The HEP architecture can be easily expanded to higher capacities while eliminating the need for redesigning the hardware platform.

More importantly, two task scheduling algorithms, MLS and MLS+BFF, have been introduced and developed as a part of a CAD framework that automatically map DUT's netlists into HEP-based emulation system. It has been shown that the proposed scheduling heuristics can maximize processors workload and reduce total emulation time while keeping the scheduling time within reasonable range. The ten largest circuits from MCNC benchmark suite were used to evaluate the performance of the scheduling tools. Based on this evaluation, the scheduling algorithms, substantially increase in the average workload in emulation processors. As a result, a large circuit, as big as 22000 gates, can be emulated in $6\mu s$. An optimization technique, introduced in MLS+BFF algorithm has further improved the average workload by 1-6% while causing no overhead on design compilation time. More

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

interestingly, unlike FBE CAD tools, the scheduling tool favors denser circuits over small circuits and produces better resource utilization for bigger circuits.

Finally, a complete CAD framework that can be used for design compilation of DUTs into HEP-based emulation systems, has been explained in details that has eliminated the need for partitioning, placement and routing tools. Hence, the design compilation time is significantly shorter and more predictable.

6.1 Future Work

The followings are a number of possible suggestions, concerning hardware and software of HEP-based emulation system, that we would like to share with readers for possible future researches.

6.1.1 Improvements in Hardware Architecture

Due to the fact that size of digital circuits is constantly increasing (Moore's law) HEP-based emulation systems with larger logic capacity will soon be needed. Fortunately, flexibility of programmable logic devices (e. g. FPGAs) allows us to not only design HEPs with higher logic capacity but also to integrate more number of them into FPGAs. Hence, providing easily scalable soft IP (Intellectual Property) core for HEP-based emulation systems will assist verification engineers to easily develop fast and cheap logic emulation systems with variable size and logic capacity. HEP-based multi-FPGA systems for emulating very large designs is also an interesting topic for future research.

The HEP based emulation engine introduced in this thesis is only capable of emulating combinatorial and fully synchronous sequential logic circuits. Although, such circuits constitute majority of all logic designs, having an HEP processor that can also emulate logic circuits with multiple asynchronous clocks may be very useful.

Lastly, integrating HEP-based emulation engine with complementary peripheral modules such as download manager, monitoring and supervisory modules will make the HEP-based emulation system a desirable verification tool for all small and medium size IC manufacturing companies.

6.1.2 Improvement in Design Compiler Tool

The improvement made by MLS+BFF algorithm is mainly due to the fact that the algorithm is capable of "predicting" the flow of signals in netlist from one level to the next immediate level. However, if the algorithm was *somehow* capable of profiling the flow of all signals in to further

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

depths within the circuits, scheduler might create even better solutions. Task scheduling for parallel processing platforms is widely open to researchers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

[1] M. Abramovici, Y. H. Levendel, and P. R. Menon. A logic simulation machine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(2):82–94, Apr. 1983.

[2] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):609–626, Jun. 1997.

[3] M. L. Bailey, J. V. Briner, and R. D. Chamberlain. Parallel logic simulation of vlsi systems. *ACM Computing Surveys (CSUR)*, 26(3):255–293, Sept. 1994.

[4] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge. HSS–a high-speed simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4):601–617, Jul. 1987.

[5] G. Beraudo and J. Lillis. Timing optimization of fpga placements by logic replication. In *Proceedings of IEEE Design Automation Conference*, pages 196–201, Jun. 2003.

[6] 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*, pages 213–222, 1997.

[7] N. B. Bhat, K. Chaudhary, and E. S. Kuh. Performance-Oriented Fully Routable Dynamic Architecture for a Field Programmable Logic Device, 1993. Memorandum No. UCB/ERL M93/42, Electronics Research Labratory, University of California, Berkeley.

[8] S. Brown, J. Rose, and Z. G. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(5):620–628, May 1992.

[9] S. Brown, J. Rose, and Z. G. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(5):620–628, May 1992.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. ACM/IEEE Design Automation Conference*, pages 46–51, Jun. 1990.

[11] M. Butts and J. Batcheller. Method of using electronically reconfigurable logic circuits, 1991. U. S. Patent 5036473.

[12] M. R. Butts. Logic multiprocessor for FPGA implementation, Jun. 2004. U. S. Patent Application 2004/0123258 A1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[13] Y. W. Chang, S. Thakur, K. Zhu, and D. F. Wong. A new global routing algorithm for fpgas. In *Proceedings of the IEEE/ACM international conference on computer-aided design*, pages 356–361, Nov. 1994.

[14] K. C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar. Dag-map: graph-based fpga technology mapping for delay optimization. *IEEE Design and Test of Computers*, 9(3):7–20, Sept. 1992.

[15] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, Jun. 1996.

[16] D. Cock and A. Carpenter. A proposed hardware fault simulation engine. In *IEEE Proc. of the European Conference on Design Automation(EDAC)*, pages 570–574, Feb. 1991.

[17] J. Cong and Y. Ding. Flowmap: an optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, Jan. 1994.

[18] J. Cong, J. Peck, and Y. Ding. Rasp: A general logic synthesis system for sram-based fpgas. In *Proceedings of ACM Fourth International Symposium on Field-Programmable Gate Arrays*, pages 137–143, Feb. 1996.

[19] J. Cong and M. Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in vlsi design. In *IEEE 30th Conference on Design Automation*, pages 755–760, Jun. 1993.

[20] Altera Corp. Available at: www.altera.com, 2006.

[21] Aptix Corp. Product brief: The System Explorer MP4, Available at: www.aptix.com, 1998.

[22] Mentor Graphics Corp. Availabele at: www.mentor.com, 2006.

[23] P. Curzon and S. Tahar. Automating the verification of parameterized hardware using a hybrid tool. In *IEEE Proc. international conference on Microelectronics(ICM)*, pages 257–260, Oct. 2001.

[24] Cadence Incisive Palladium Datasheet, 2006. Available at: www.cadence.com/datasheets/incisivepalladiumII_ds.pdf.

[25] A. DeHon. A First Generation DPGA Implementation, 1995. MIT Transit Note 114. Available at:¡http://www.ai.mit.edu/projects/transit/tn114/tn114.html¿.

[26] M. M. Denneau. The yorktown simulation engine. In *ACM Proceedings of the 19th Conference on Design Automation*, pages 431–435, Jan. 1982.

[27] R. Eastham and K. Thirunarayan. Proof strategies for hardware verification. In *IEEE Proc. of National Aerospace and Electronics Conference*, pages 451–458, May 1996.

[28] A. Ejnioui and N. Ranganathan. Design partitioning on single-chip emulation systems. In *IEEE Thirteenth International Conference on VLSI Design*, pages 234–239, Jan. 2000.

[29] W. F. Beausoliel et al. Multiprocessor for hardware emulation, 1996. U. S. Patent 5551013.

[30] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of 19th Design Automation Conference*, pages 175–181, Jun. 1982.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[31] R. J. Francis, J. Rose, and K. Chung. Chortle: a technology mapping program for lookup table-based field programmable gate arrays. In *Proceedings of 27th ACM/IEEE Design Automation Conference*, pages 613–619, Jun. 1990.

[32] D. Gajski, N. Dutt, A. Wu, and S. lin. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1994.

[33] R. Hartley, K. Welles, M. Hartman, A. Chatterjee, P. Delano, B. Molnar, and C. Rafferty. A rapid-prototyping environment for digital-signal processors. *IEEE Design and Test of Computers*, 8(2):11–25, Jun. 1991.

[34] S. Hauck, G. Borriello, and C. Ebeling. Mesh routing topologies for multi-fpga systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(3):400–408, Sept. 1998.

[35] A. Hemani. Charting the EDA roadmap. *IEEE J. Circuits and Devices Magazine*, 20(6):5–10, Nov. 2004.

[36] J. Hwang and A. El-Gamal. Optimal replication for min-cut partitioning. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 432–435, Nov. 1992.

[37] T. T. Hwang, R. M. Owens, M. J. Irwin, and K. H. Wang. Logic synthesis for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(10):1280–1287, Oct. 1994.

[38] Cadence Design System Inc. Available at: www.cadence.com, 2006.

[39] Xilinx Inc. Available at: www.xilinx.com, 2006.

[40] D. Jones. A Time-Multiplexed FPGA Architecture for Logic Emulation, 1995. M. A. Sc. Thesis, University of Toronto.

[41] K. Keutzer. The need for formal verification in hardware design and what formal verification has not done for me lately. In *IEEE International Workshop on the HOL Theorem Proving System and Its Applications*, pages 77–86, Aug. 1991.

[42] M. .A. S. Khalid and J. Rose. A novel and efficient routing architecture for multi-fpga systems. *IEEE Transactions on VLSI Systems*, 8(1):30–39, Feb. 2000.

[43] S. Kirkpatrick, C. Gelatt, and M. Vecchi. In Science, 1983. Vol. 220, No. 4598,671.

[44] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits (Accepted for future publication)*, PP(99):1–13.

[45] Y. K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, Dec. 1999.

[46] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C-20(12):1469–1479, Dec. 1971.

[47] H. Li, W. K. Mak, and S. Katkoori. Force-directed performance-driven placement algorithm for fpgas. In *Proceedings of IEEE Computer society Annual Symposium on VLSI*, pages 193–198, Feb. 2004.

[48] D. MacMillen, R. Camposano, M. Butts, D. Hill, and T. W. Williams. An industrial view of electronic design automation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1428–1448, Dec. 2000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[49] P. Maidee, C. Ababei, and K. Bazargan. Fast timing-driven partitioning-based placement for island style fpgas. In *Proceedings of IEEE Design Automation Conference*, pages 598–603, Jun. 2003.

[50] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for fpgas. In *Proceedings of ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 203–213, Feb. 2000.

[51] G. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8), 1956. Available: ftp://download.intel.com/research/silicon/moorespaper.pdf.

[52] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. On clustering for minimum delay/area. In *IEEE International Conference on Computer-Aided Design (Digest of Technical Papers)*, pages 6–9, Nov. 1991.

[53] C. Pixley, A. Chittor, F. Meyer, S. McMaster, and D. Benua. Functional verification 2003: technology, tools and methodology. In *IEEE Proc. International Conference on ASIC*, pages 1–5, Oct. 2003.

[54] V. R. Pratt, P. D. Mosses, M. Nielsen, and M. I. Schwartzbach. Anatomy of the pentium bug. *Theory and Practice of Software Development (TAPSOFT), Vol. 915 of Lecture Notes in Computer Science, Springer-Verlag*, pages 97–107, 1995.

[55] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, Jul. 1993.

[56] M. Schutz. How to efficiently build vhdl testbenches. In *IEEE Proc. EURO Design Automation Conference (EURO-DAC) with EURO-VHDL*, pages 554–559, Sept. 1995.

[57] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis, 1992. EECS Department, University of California, Berkeley, URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html, No. UCB/ERL M92/41.

[58] K. Shahookar and P. Mazumder. Vlsi cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, Jun. 1991.

[59] N. A. Sherwani. *Algorithms for VLSI Physical Design Automation (Second Printing).* Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, 1994.

[60] S. Singh, J. Rose, and P. Chow D. Lewis. The effect of logic block architecture on fpga performance. *IEEE Journal of Solid-State Circuits*, 27:281–287, Mar. 1992.

[61] L. Soule and T. Blank. Parallel logic simulation on general purpose machines. In *Proc. of 25th ACM/IEEE Design Automation Conference*, pages 166–171, Jun. 1988.

[62] H. P. Su and Y. L. Lin. A phase assignment method for virtual-wire-based hardware emulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(7):776–783, Jul. 1997.

[63] S. Trimberger. Scheduling designs into a time-multiplexed fpga. In *Proceedings of ACM/SIGDA sixth international symposium on Field Programmable Gate Arrays*, pages 153–160, 1998.

[64] S. Trimberger, D. Carberry, and A. Johnson J. Wong. A time-multiplexed fpga. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, Apr. 1997.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[65] J. Varghese, M. Butts, and J. Batcheller. An efficient logic emulation system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):171–174, Jun. 1993.

[66] K. Wakabayashi and T. Okamoto. C-based SoC design flow and EDA tools: an ASIC and system vendor perspective. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), Dec. 2000.

[67] S. Walters. Computer-aided prototyping for asic-based systems. *IEEE Design and Test of Computers*, 8(2):4–10, Jun. 1991.

[68] Y. C. Wei and C. K. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):911–921, Jul. 1991.

[69] S. Yang. Logic Sythesis and Optimization Benchmarks, version 3.0, 1991. Micro-electronics Centre of North Carolina, P. O. Box 12889, Research Triangle Park, NC. , 27709, USA.

[70] A. A. Yazdanshenas and M. A. S. Khalid. Logic emulation systems: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 2006. Paper currently under review.

[71] Y. Zhu and T. Marshall. Design verification using formal techniques. In *IEEE Proc. International Conference on ASIC*, pages 21–28, Oct. 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA AUCTORIS

Amir Ali Yazdanshenas was born in , Tehran, Iran, on June 10, 1975. He received his B.A.Sc. degree in Computer Hardware Engineering in 1999 from the Iran University of Science and Technology (IUST). He is currently a candidate in the electrical and computer engineering M.A.Sc. program at the University of Windsor. His research interests include Logic Emulation Systems, field-programmable logic devices, embedded system design, computer architecture, and high performance VLSI circuit design.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.