THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Reconfigurable Architectures for the Next Generation of Mobile Device Telecommunications Systems

Ahmed Osman El-Rayis

Thesis submitted for the degree of Doctor of Philosophy.

The University of Edinburgh

September 2014

Declaration of originality

I hereby declare that the research recorded in this thesis and the thesis itself was composed and originated entirely by myself in the School of Engineering at the University of Edinburgh.

Ahmed Osman El-Rayis

ACKNOWLEDGEMENT

Praise be to GOD the Almighty, who made me capable enough to complete this thesis.

I am also grateful to my supervisor, Prof. Tughrul Arslan, for his knowledge, expert guidance, trust, support, motivation and advice. His technical acumen, suggestions and timely discussions are heartily appreciated.
I wish to express my gratitude to Dr. Khaled Benkrid for his helpful suggestions and lively discussions during the writing of this work. I am grateful to Dr. Ahmet T. Erdogan for his technical guidance, and also to the RICA team for their support and assistance in using the tools and keeping upgrades to the latest available version.

Very special thanks go to my parents, Osman and Zeinab, for their prayers, continuous support and encouragement throughout my life. This thesis is dedicated to them for their unlimited love, patience and unwavering belief in my capabilities.

Special thanks to my wife Olga and my beautiful daughters Laila and Sara in whose presence I always find that life is at its brightest.

Last but not least, my sisters are always on my side supporting me and following my success eagerly. Aya and Amel - thank you, my dears.

PUBLICATIONS

Publications directly related to this thesis:

- **A.O. El-Rayis**, T. Arslan, K. Benkrid, Reconfigurable architectures for next generation mobile devices, *Consumer Electronics Times (CET) Journal*, Accepted Sept. 2014. [Chapter 2]

- **A.O. El-Rayis,** T. Arslan, A.T. Erdogan, A processing engine for GPS correlation, *IEEE 8th Symposium on Application Specific Processors (SASP) 2010*, pp. 44-49, 13-14 June 2010. [Chapter 6]

- **A.O. El-Rayis,** X. Zhao, T. Arslan, A.T. Erdogan, Low power RS codec using cell-based reconfigurable processor, *22nd IEEE International SOC Conference (SOCC) 2009*, pp. 279-282, 9-11 Sept. 2009. [Chapter 5]

- **A.O. El-Rayis,** X. Zhao, T. Arslan, A.T. Erdogan, Dynamically programmable Reed Solomon processor with embedded Galois Field multiplier, *International Conference on Field-Programmable Technology (FPT) 2008*, pp. 269-272, 8-10 Dec. 2008. [Chapter 5]

- **A.O. El-Rayis**, T. Arslan, A. Erdogan, High performance embedded reconfigurable concatenated convolution-puncturing fabric for 802.16, *NASA/ESA Conference on Adaptive Hardware and Systems 2007*, pp. 190-194, August 5-8, 2007. [Chapter 3]

Publications inspired by or indirectly related to the work in this thesis:

- **A.O. El-Rayis,** T. Arslan, A.T. Erdogan, Addressing future space challenges using reconfigurable instruction cell-based architectures, *NASA/ESA Conference on Adaptive Hardware and Systems, (AHS) 2008*, pp. 199-203, 22-25 June 2008.

- W. Li, T. Arslan, J. Han, A.T. Erdogan, **A.O. El-Rayis,** N. Haridas, E. Yang, Energy efficiency enhancement in satellite based WSN through collaboration and self-organized mobility, *IEEE Aerospace conference 2009*, pp. 1-8, 7-14 March 2009.

- W. Li, T. Arslan, **A.O. El-Rayis,** N. Haridas, A.T. Erdogan, E. Yang, Distributed adaptability and mobility in space-based wireless pico-satellite sensor networks, *NASA/ESA Conference on Adaptive Hardware and Systems, (AHS) 2008*, pp. 277-282, 22-25 June 2008.

ABSTRACT

Mobile devices have become a dominant tool in our daily lives. Business and personal usage has escalated tremendously since the emergence of smartphoness and tablets. The combination of powerful processing in mobile devices, such as smartphoness and the Internet, have established a new era for communications systems. This has put further pressure on the performance and efficiency of telecommunications systems in delivering the aspirations of users. Mobile device users no longer want devices that merely perform phone calls and messaging. Rather, they look for further interactive applications such as video streaming, navigation and real time social interaction. Such applications require a new set of hardware and standards. The WiFi (IEEE 802.11) standard has been at the forefront of reliable and high-speed internet access telecommunications. This is due to its high signal quality (quality of service) and speed (throughput). However, its limited availability and short range highlights the need for further protocols, in particular when far away from access points or base stations. This led to the emergence of 3G followed by 4G and the upcoming 5G standard that, if fully realised, will provide another dimension in “anywhere, anytime internet connectivity.” On the other hand, the WiMAX (IEEE 802.16) standard promises to exceed the WiFi signal coverage range. The coverage range could be extended to kilometres at least with a better or similar WiFi signal level.

This thesis considers a dynamically reconfigurable architecture that is capable of processing various modules within telecommunications systems. Forward error correction, coder and navigation modules are deployed in a unified low power communication platform. These modules have been selected since they are among those with the highest demand in terms of processing power, strict processing time or throughput. The modules are mainly realised within WiFi and WiMAX systems in addition to global positioning systems (GPS). The idea behind the selection of these modules is to investigate the possibility of designing an architecture capable

of processing various systems and dynamically reconfiguring between them. The GPS system is a power-hungry application and, at the same time, it is not needed all of the time. Hence, one key idea presented in this thesis is to effectively exploit the dynamic reconfiguration capability so as to reconfigure the architecture (GPS) when it is not needed in order to process another needed application or function such as WiFi or WiMAX. This will allow lower energy consumption and the optimum usage of the hardware available on the device.

This work investigates the major current coarse-grain reconfigurable architectures. A novel multi-rate convolution encoder is then designed and realised as a reconfigurable fabric. This demonstrates the ability to adapt the algorithms involved to meet various requirements. A throughput of between 200 and 800 Mbps has been achieved for the rates 1/2 to 7/8, which is a great achievement for the proposed novel architecture. A reconfigurable interleaver is designed as a standalone fabric and on a dynamically reconfigurable processor. High throughputs exceeding 90 Mbps are achieved for the various supported block sizes. The Reed Solomon coder is the next challenging system to be designed into a dynamically reconfigurable processor. A novel Galois Field multiplier is designed and integrated into the developed Reed Solomon reconfigurable processor. As a result of this work, throughputs of 200Mbps and 93Mbps respectively for RS encoding and decoding are achieved. A GPS correlation module is also investigated in this work. This is the main part of the GPS receiver responsible for continuously tracking GPS satellites and extracting messages from them. The challenging aspect of this part is its real-time nature and the associated critical time constraints. This work resulted in a novel dynamically reconfigurable multi-channel GPS correlator with up to 72 simultaneous channels.

This work is a contribution towards a global unified processing platform that is capable of processing communication-related operations efficiently and dynamically with minimum energy consumption.

CONTENTS

Declaration of originality ii

PUBLICATIONS iv

ABSTRACT v

CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES xv

LIST OF ACRONYMS AND ABBREVIATIONS xvi

1 INTRODUCTION 1

1.1 MOTIVATION 1
1.2 CONTRIBUTION 3
1.3 STRUCTURE 5
1.4 SUMMARY OF CONTRIBUTION 6

2 RECONFIGURABLE ARCHITECTURES 7

2.1 INTRODUCTION 7
2.2 BACKGROUND 8
2.3 GENERAL, DSP, FINE AND COARSE PROCESSORS 9
2.3.1 Reconfigurable Computing Classes *11*
2.4 COARSE GRAIN RECONFIGURABLE ARCHITECTURES 12
2.4.1 CRISP: A Coarse-Grained Reconfigurable Instruction Set Processor *12*
2.4.2 Systolic Ring Architecture *14*
2.4.3 MATRIX Architecture *15*
2.4.4 Cell Matrix and vCell Matrix Architectures *18*
2.4.5 Pleiades Architecture *21*
2.4.6 OneChip Architecture *22*
2.4.7 Chimaera Architecture *23*
2.4.8 REMARC Architecture *24*
2.4.9 RaPiD Architecture *26*
2.4.10 Garp Architecture *28*
2.4.11 SRGA Architecture *30*
2.4.12 CHESS Architecture *31*
2.4.13 DART Architecture *33*
2.4.14 DReAM Architecture *34*
2.4.15 PADDI Architecture *37*
2.4.16 MorphoSys Architecture *39*

2.4.17 *PipeRench Architecture* *40*
2.4.18 *rDPA Architecture* *41*
2.4.19 *KressArray Architecture* *43*
2.4.20 *MOVE Architecture* *45*
2.4.21 *RICA Architecture* *48*
2.4.22 *CDDS Variable Datapath Architecture* *51*
2.4.23 *BilRC Architecture* *52*
2.5 COMPARISON AND DISCUSSION 53
2.6 CONCLUSION 56

3 MULTIRATE CONVOLUTION ENCODER 58

3.1 INTRODUCTION 58
3.2 CONVOLUTION ENCODER AND PUNCTURING CONFIGURATION 60
3.2.1 *Convolution Encoder* *60*
3.2.2 *Puncturing Configuration* *62*
3.3 TECHNIQUE FOR THE PARALLELIZATION PUNCTURED CONVOLUTION ENCODER 62
3.4 RECONFIGURABLE CONCATENATED CONVOLUTION-PUNCTURING ARCHITECTURE 66
3.5 RESULTS 70
3.6 CONCLUSION 71

4 RECONFIGURABLE INTERLEAVER ON DYNAMICALLY CELL-BASED ARCHITECTURE AND AS A FABRIC 73

4.1 INTRODUCTION 73
4.2 INTERLEAVER/DE-INTERLEAVER 74
4.3 RECONFIGURABLE INTERLEAVER 77
4.4 RECONFIGURABLE INTERLEAVER FABRIC 78
4.5 INTERLEAVER ON A DYNAMICALLY RECONFIGURABLE PROCESSOR 82
4.6 RESULTS AND ANALYSIS 84
4.7 CONCLUSION 87

5 DYNAMICALLY PROGRAMMABLE REED SOLOMON PROCESSOR 88

5.1 INTRODUCTION 88
5.2 RECONFIGURABLE RS-CODEC ALGORITHMS 90
5.3 RECONFIGURABLE RS PROCESSOR 91
5.4 RS ENCODER AND DECODER IMPLEMENTATION ON NOVEL RS PROCESSOR 93
5.4.1 *RS Encoding and Novel Design* *93*
5.4.2 *RS Decoder on RS Processor* *95*
5.5 GALOIS FIELD MULTIPLIER CELL FOR RS PROCESSOR 98
5.6 RS PROCESSOR IMPLEMENTATIONS AND OPTIMISATIONS 100
5.6.1 *Architecture Specific Optimisations* *100*
5.6.2 *RS Encoder Implementation and Optimisation* *107*
5.6.3 *RS Decoder Implementation and Optimisation* *109*
5.7 PERFORMANCE, COMPARISON ANALYSIS AND RESULTS 111
5.7.1 *GF Multiplier Cell* *111*
5.7.2 *RS Codec Processor* *112*

5.8 CONCLUSION ... 115

6 GPS DIGITAL MATCHED FILTERS USING DYNAMICALLY RECONFIGURABLE ARCHITECTURE ... 116

6.1 INTRODUCTION ... 116
6.2 CORRELATION ARCHITECTURES ... 117
6.2.1 Serial Search Correlation ... 119
6.2.2 Conventional Digital Matched Filter ... 120
6.2.3 Differential Digital Matched Filter ... 121
6.2.4 Segment Processing Digital Matched Filter ... 122
6.2.5 Algorithmic Comparison ... 123
6.3 ENGINE ARCHITECTURE ... 123
6.4 ANALYSIS AND OPTIMISATIONS ... 125
6.4.1 Tier 1: Correlation Implementation ... 125
6.4.2 Tier 2: Architecture Optimisations ... 127
6.4.3 Tier 3: Engine Optimisation ... 129
6.5 PERFORMANCE ANALYSIS ... 131
6.6 CONCLUSION ... 138

7 GPS MULTI-CHANNEL CORRELATION USING THE DYNAMICALLY RECONFIGURABLE PLATFORM ... 139

7.1 INTRODUCTION ... 139
7.2 GPS ENGINE CORRELATOR CAPABILITY REVIEW ... 139
7.3 GPS CORRELATION ‘ENGINE 2’ ... 140
7.4 MULTI-CHANNEL CORRELATION SOLUTION ... 142
7.5 ANALYSISOF RESULTS AND DISCUSSION ... 144
7.6 CONCLUSIONS ... 151

8 SUMMARY AND FUTURE WORK ... 152

8.1 INTODUCTION ... 152
8.2 SUMMARY OF THESIS AND CONTRIBUTION ... 152
8.3 FUTURE WORK ... 156

REFERENCES ... 158

APPENDIX A: MATLAB MODELS ... 169

A.1 INTERLEAVER MODEL: ... 169
A.2 16-QAM INTERLEAVER MODEL: ... 170
A.3 QPSK INTERLEAVER MODEL: ... 172

APPENDIX B: VERILOG DESIGNS ... 174

B.1 INTERLEAVER DESIGN: ... 174
B.2 GF RECONFIGURABLE MULTIPLIER DESIGN ... 187
B.3 RS ENCONDER DESIGN ... 189

LIST OF FIGURES

Figure 1-1 General block diagrams of WiMAX/WiFi transmitter and receiver in addition to GPS receiver. The hashed blocks are the subject of this thesis 2
Figure 2-1 FPGA building block - Xilinx..8
Figure 2-2 Computation architecture characteristics 10
Figure 2-3 CRISP architecture construction of processor and a reconfigurable functional unit [13] 12
Figure 2-4 Reconfigurable slice internal structure [13]...13
Figure 2-5 The Ring architecture layout [14] 15
Figure 2-6 Dnode architecture [14] 16
Figure 2-7 Ring style for Dnode interconnections [14]...16
Figure 2-8 MATRIX Processing element (functional unit) [15] 17
Figure 2-9 MATRIX nearest neighbour interconnect [15] 18
Figure 2-10 PE structure for the Cell matrix architecture [16].......................................19
Figure 2-11 PE structure for the vCell matrix architecture [19].....................................19
Figure 2-12 Two-dimensional array structure for the vCell matrix architecture [19] 19
Figure 2-13 Two-dimensional array structure for the Cell Matrix architecture [19]...........20
Figure 2-14 Pleiades processor reconfigurable architecture layout [17] 22
Figure 2-15 OneChip architecture layout 23
Figure 2-16 Chimaera architecture layout [7]..23
Figure 2-17 REMARC architecture layout [8] 25
Figure 2-18 REMARC architecture internal structure [8] 25
Figure 2-19 RaPiD-I basic cell structure [9]..27
Figure 2-20 Garp architecture block diagram [18] 27
Figure 2-21 Garp array organisation [18] 29
Figure 2-22 Garp program flowchart [18] 30
Figure 2-23 SRGA architecture interconnect mesh [11] 31
Figure 2-24 CHESS architecture layout and neighbouring interconnections [10] 32
Figure 2-25 DART system level architecture 33
Figure 2-26 DART cluster construction [12]..34
Figure 2-27 DReAM architecture structure 35
Figure 2-28 DReAM hierarchy control for dynamic reconfiguration [19].........................36

Figure 2-29 PADDI architecture structure [20] ... 38
Figure 2-30 PADDI EXU architecture [20] ... 38
Figure 2-31 MorphoSys architecture layout [25] ... 39
Figure 2-32 MorphoSys 8x8 reconfigurable cell array and row-column connectivity between each reconfigurable cell (RC) [25] ... 40
Figure 2-33 PipeRench architecture layout [26] ... 41
Figure 2-34 rDPA architecture with ALU controller [27] ... 42
Figure 2-35 KressArray architecture [27] ... 43
Figure 2-36 rDPU four configuration layers ... 44
Figure 2-37 Move architecture structure [30] ... 46
Figure 2-38 MOVE32INT block diagram [31] ... 47
Figure 2-39 Dynamic allocation of instruction cells into processing steps, scheduled within the GCC tool chain ... 50
Figure 2-40 CDDS architecture's reconfigurable datapath separated into static and dynamic parts [40] ... 51
Figure 2-41 PE column based structure in BilRC [41] ... 52
Figure 2-42 Classification based on control/arithmetic ratio [42] ... 54
Figure 3-1 Convolution encoder and puncturing configuration as separate units in the transmitter of a wireless communication system ... 59
Figure 3-2. Convolution encoder (rate=1/2, k=7) ... 61
Figure 3-3. Parallel punctured convolution encoder [49] ... 62
Figure 3-4. Punctured convolution encoder for the rate 2/3 ... 64
Figure 3-5. Punctured convolution encoder for the rate 5/6 ... 64
Figure 3-6. Punctured convolution encoder for rate 7/8 ... 65
Figure 3-7. First conventional approach for implementing punctured convolutional encoder for different rates ... 66
Figure 3-8. Second conventional approach for implementing punctured convolution encoder for different rates ... 67
Figure 3-9 Proposed top-level architecture for low power reconfigurable concatenated convolution-puncturing module for 802.16 ... 68
Figure 3-10 Reconfigurable interconnections for the convolution-puncturing core (delay units are implemented as registers) ... 68
Figure 3-11: Reconfigurable serial to parallel ... 69
Figure 4-1. The interleaver and de-interleaver in OFDM WiMAX baseband ... 74

Figure 4-2: Proposed reconfigurable interleaver 80
Figure 4-3. WIMAX interleaver for QPSK modulation 81
Figure 4-4 Architecture Cells dynamic power in µW for interleaver 576 64-QAM with two design methods and their optimisations 82
Figure 4-5. Architecture's execution time in µs for interleaver 576 64-QAM with two design methods and their optimisations 82
Figure 4-6. Architecture cells dynamic power in µW for interleaver 768 16-QAM with two design methods and their optimisations 83
Figure 4-7. Architecture's execution time in µs for interleaver 768 16-QAM with two design methods and their optimisations 84
Figure 4-8. Reconfigurable interleaver execution time in µs for the various modes on the dynamically reconfigurable architecture RICA 85
Figure 4-9. Reconfigurable interleaver dynamic power consumption in µW for the various modes on the dynamically reconfigurable architecture RICA 85
Figure 4-10. Reconfigurable interleaver "steps" count for the various modes on the dynamically reconfigurable architecture RICA 86
Figure 5-1 Reed-Solomon processor based on dynamically reconfigurable heterogeneous cell array. 92
Figure 5-2. Reed-Solomon encoder using linear feedback shift register with n-k stages: (a) classical RS encoder architecture; (b) novel design of parallel parity output 94
Figure 5-3. Reed Solomon decoder main algorithms 95
Figure 5-4. Syndrome computation architecture 95
Figure 5-5. The internal architecture of a single GF multiplier for 8-bit data width 98
Figure 5-6. A novel 8 bit GFMUL cell with four embedded GF multipliers maximising the throughput by applying the SIMD technique (a) GF multiplier cell layout, (b) GF multiplier internal structure 99
Figure 5-7. SIMD architecture for syndrome computation 102
Figure 5-8 Reed Solomon decoder (a) classical software approach of RS decoder; (b) new approach with GF multiplier cell 103
Figure 5-9 RICA architecture tool flow 104
Figure 5-10 Programmable Reed Solomon processor architecture based on heterogeneous cell array (dark coloured cells represent active cells in a certain configuration). The three cases are: (a) initial configuration; (b) intermediate configuration, in which certain cells are

configured to code/decode data from/to input/output ports; and (c) final configuration, flushing remaining data out, and preparing for the subsequent configuration..................106
Figure 5-11 Reconfigurable RS encoder data-flow graph 107
Figure 5-12 Reconfigurable RS encoder modified data-flow graph using GFMUL cell ..108
Figure 5-13. Reed-Solomon decoder algorithm design 109
Figure 5-14 Memory usage for RS decoding on RS processor 112
Figure 5-15 Throughput of RS decoding on RS processor..113
Figure 5-16 Energy consumption for the RS decoder on RICA......................................114
Figure 6-1 Serial search correlation...119
Figure 6-2 Conventional digital matched filter (CDMF)...120
Figure 6-3 Differential digital matched filter (DDMF) 121
Figure 6-4 Segment processing digital matched filter (SPDMF) with K=2 122
Figure 6-5 Modified conventional digital matched filter (MCDMF) architecture 128
Figure 6-6 Modified differential digital matched filter (MDDMF) architecture..............129
Figure 6-7. Modified segment processing digital matched filter (MSPDMF) architecture for K=2 130
Figure 6-8 Comparison of resulting Correlation times (ms) for the matching filter architectures..132
Figure 6-9 Comparison of cell dynamic energy (μJ) for various matched filter architectures 133
Figure 6-10 Comparison of data memory access energy (μJ) for various matched filter architectures..133
Figure 6-11 Comparison of total energy* (μJ) for various matched filter architectures....133
Figure 6-12 Comparison of total step count for various matched filter architectures 134
Figure 6-13 Comparison of memory usage (bytes) for various matched filter architectures 134
Figure 6-14 Comparison of program memory access energy (μJ) for various matched filter architectures..134
Figure 6-15 Comparison of correlation time (μs) with packed data results for matching filter architectures 135
Figure 6-16 Comparison of data and Program memory access energy (μJ) for various matched filter architectures with packed data optimisation...136
Figure 6-17 Comparison of cells and total dynamic energy (μJ) for various matched filter architectures with packed data optimisation..136

Figure 6-18 Comparison of total step count for various matched filter architectures with packed data optimisation 137
Figure 6-19 Comparison of memory usage (bytes) for various matched filter architectures with packed data optimisation 137
Figure 7-1 'Engine 2' is capable of providing 12 GPS correlation channels 141
Figure 7-2 'Engine 3' is capable of providing 72 GPS correlation channels 143
Figure 7-3 'Engine 4' is capable of providing 144 GPS correlation channels 144
Figure 7-4 'Engine 5' core based on two 'Engine 4' cores or twenty four 'Engine 2' cores for providing 288 GPS correlations..145
Figure 7-5 Variation in the single correlation results between the different multi-correlation engines..146
Figure 7-6 Memory usage for the different multi-correlation engines 146
Figure 7-7 Total dynamic energy for the different multi-correlation engines...................146
Figure 7-8 Data memory access energy for the different multi-correlation engines 147
Figure 7-9 Number of core cells used in each engine...147
Figure 7-10 Normalised cell numbers per correlation engine 149
Figure 7-11 Normalised number of cells used per engine and the associated normalised number of correlations..149
Figure 7-12 Cost of adding "Engine 2" to the various engines 150
Figure 8-1 Fully reconfigurable baseband architecture 156

LIST OF TABLES

Table 2-1 Main types of instruction cells types in the RICA architecture..........................48

Table 2-2 Comparison of studies of reconfigurable architectures......................................55

Table 3-1 WiMAX puncture pattern configuration and resulting convolution code serialization...61

Table 3-2: External and internal configuration words...70

Table 3-3. Area and estimated power for all supported rates...71

Table 3-4. Throughput for all supported rates for the core unit and whole module............71

Table 4-1 Block sizes of bit interleaver for WiMAX [44]...75

Table 4-2 Value of the *s* parameter in the interleaver/de-interleaver equations..................76

Table 4-3 Block sizes of bit interleaver for Wimax. Circle marks show similar block sizes with different modulations and N_{cbps} [1]...78

Table 4-4 External configuration word for interleaver/de-interleaver................................79

Table 4-5 Internal configuration word: 4 bit 80

Table 5-1 Number of additions and multiplications in the Reed-Solomon decoder............98

Table 5-2 Implementation results for the GF multiplier cell 111

Table 5-3 Performance comparison of the novel RS processor and StarCore 140 for RS(255,239)...113

Table 6-1 Comparison of theoretical computational complexity for various correlation algorithms 123

Table 6-2 Comparison of results for digital matched filter correlator architectures (without packed data)...132

Table 6-3 Comparison of results for digital matched filter correlator architectures (with packed data)...135

Table 7-1 Industrial correlator processors and associated correlation channels...............140

Table 7-2 Summary of key cell numbers used in each engine 148

Table 7-3 Summary of the number of correlations and embedded ‘Engine 2’s’ in each engine...150

LIST OF ACRONYMS AND ABBREVIATIONS

ACS	Add compare select
ADC	Analogue to digital converter
ADSL	Asymmetric digital subscriber line
ALM	Adaptive logic module
ALU	Arithmetic logic unit
ASIC	Application specific integrated circuit
ASRF	Application specific reconfigurable fabric
BPSK	Binary phase shift keying
CBox	Connection box
CCM	Custom compute machine
CDMA	Code division multiple access
CLB	Configurable logic blocks
CMOS	Complementary metal oxide semiconductor technology
DCT	Discrete cosine transform
DP	Data ports
DSP	Digital signal processing
FEC	Forward error correction
FFT	Fast Fourier transform
FIFO	First in, first out
FIR	Finite impulse response filter
FLUF	Full loop unfolding
FP	Forward processor
FPGA	Field programmable gate array
FSM	Finite state machine
FSM RAM	Forward state metrics random access memory
FU	Functional unit
GPS	Global positioning system
GSM	Global system for mobile communications
I/O	Input-output
IFU	Interconnected functional units

LAB	Logic array block
LB	Logic block
LC	Logic cell
LE	Logic element
LFSR	Linear feedback shift register
LLR	Log likelihood ratio
LSB	Least significant bit
LUF	Loop unfolding
LUT	Look-up-table
MAC	Media access control
MAC	Multiply accumulate
Mbps	Mega bit per second
MDF	Machine description file
MSB	Most significant bit
OFDM	Orthogonal frequency division multiplexing
PE	Processing element
PGAs	Programmable gate arrays
PHY	Physical layer
QoS	Quality of service
QPSK	Quadrature phase shift keying
RAM	Random access memory
RD	Reconfigurable design
RF	Reconfiguration fabric
RISC	Reduced instruction set computing
ROM	Read only memory
RP	Reverse processor
RS	Reed solomon
RSC	Recursive systematic convolutional
RTL	Register transfer language
SAIF	Switching activity interchange format
SBOX	Switch box
SDF	Standard delay format
SIMD	Single instruction multiple data
SIMT	Single instruction multiple transports

SISO	Soft input soft output
SNR	Signal to noise ratio
SOC	System on chip
SRAM	Static random access memory
TTA	Transport triggered architecture
VLIW	Very large instruction word
VLSI	Very large scale integration
WCDMA	Wide band carrier division multiple access
WiMAX	Worldwide interoperability for microwave access
WL	Window length
WLAN	Wireless local area network
WRT	With respect to

Chapter 1

INTRODUCTION

1.1 Motivation

Telecommunications systems are integrated into almost every consumer device. They can be found in cars, wristwatches and, lately, in spectacles. The most obvious example is mobile devices and, in particular, Smartphones. Consumers expect their smartphone to be compact and condensed with functions. Its most important features are the telecommunications functions. Communication protocols range widely from WiMAX (IEEE 802.16) and WiFi (IEEE 802.11) to 3G, 4G and the upcoming 5G. In addition, there are essential communication systems such as GPS (Global Positioning System) for location services. All these functions require dedicated processors with high performance.

Power consumption is a key challenge in mobile devices. It has been widely observed that the smartphones suffer from shorter battery life. This is mainly due to the all-time connectivity to the internet and all active communication services with classic processors. Therefore, the main motivation of this work is to try to reduce the energy consumption on such devices. A two-fold approach has been chosen: firstly, using a reconfigurable architecture for the realisation of a communication system; and secondly, making this architecture dynamically reconfigurable. The dynamism will not only allow functions to be reprogrammable but the architecture will also be dynamically reconfigurable in order to process another function. This work includes the design of key functions of communication systems into dynamically reconfigurable processor in order to reduce power consumption. Furthermore, the processor could switch or reconfigure from one communication system to another. This will create savings in area and power consumption and

enable the use of dynamically reconfigurable or programmable baseband processors.

Key functions within the communication systems of WiFi, WiMAX and GPS have been identified as challenging in terms of processing time, power or resources. Furthermore, they are certainly challenging in terms of the various modes, options or configurations that such functions or modules are required to support. Hence, the following modules or functions have been considered in this work. Firstly, convolution, puncturing and de-puncturing are co-located modules that have to support multiple rates in WiFi as well as WiMAX systems. Secondly, interleaver and de-interleaver have multiple block sizes; and finally, Reed Solomon encoder and decoder have different block sizes and various primitive polynomials as well as correction capabilities and the associated challenges. All the above-mentioned functions are constrained in throughput for the transmitter part to be up to at least 70Mbps. In addition, the GPS correlation function is challenging in terms of both processing and time constraints. Figure 1-1 presents general block diagram of the

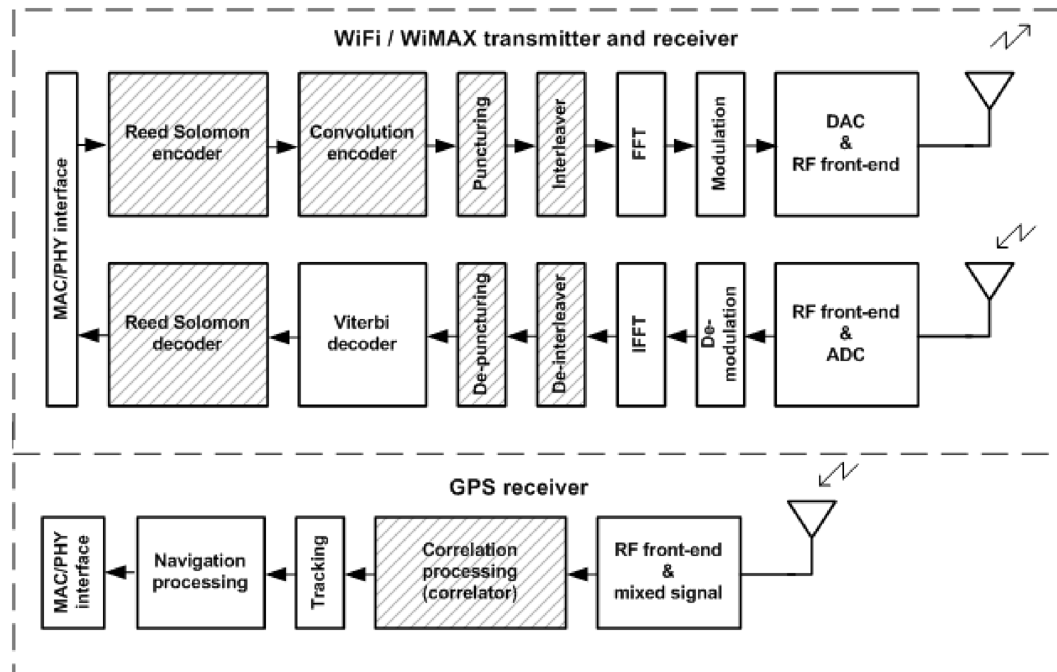

Figure 1-1 General block diagrams of WiMAX/WiFi transmitter and receiver in addition to GPS receiver. The hashed blocks are the subject of this thesis

WiFi and WiMAX transmitter and receiver along with the GPS receiver. The hashed blocks are the ones being considered and studied in this work.

1.2 Contribution

The main contribution of this work is to design reconfigurable processors that could be exploited to implement various telecommunications functions using dynamically reconfigurable architectures.

A novel reconfigurable architecture that provides a multi-rate punctured convolution coder is introduced (Chapter 3). This architecture incorporates both convolution and puncturing and can be used in wire/wireless communication systems. The convolution-punctured multi-rate architecture has achieved a superior throughput of 100 Mbps for all the required rates. Although the main architecture is the core, which provides the concatenated convolution-punctured code, reconfigurable input and output interfaces were designed and added to broaden the usability of this reconfigurable fabric. The main advantage of this architecture is that a single clock cycle is sufficient to provide the parallel convolution punctured code for its parallel inputs, which can be used to maximise the throughput of the whole transmitter system.

A novel reconfigurable interleaver is also presented (Chapter 4). The target application was the WiMAX standard with its sophisticated block size system. The interleaver has been researched and designed with a reconfigurable fabric architecture and dynamically reconfigurable instruction cell-based architecture (RICA). The interleaver throughput as a reconfigurable fabric satisfies the standard requirements, while on RICA the throughput as well as the dynamic power consumption are superior to the fabric realisation and other ASIC realisations. These results are a step forward toward a fully reconfigurable baseband telecommunications system. Moreover, the results represent a promising step toward integrating the whole WIMAX system on a dynamically reconfigurable architecture.

A novel Reed Solomon (RS) encoder architecture with parallel parity output is also introduced (Chapter 5). A novel high-speed and low-power 4x8-bit Galois Field (GF) multiplier cell is embedded within the novel low-power processor. A programmable Reed Solomon coding processor is introduced along with its design, optimisation and implementation. The real-time programmable RS encoder and decoder processor supports several communication standards, such as WiMAX and DVB-H. A number of approaches and optimisation techniques have been implemented in order to enhance the processor performance. The processor achieves high throughput and provides significant improvements in performance and energy consumption. The novel dedicated GF multiplier cell leads to a reduction in memory access energy of 72.4%, which improves the overall performance of the processor. Different design approaches and optimisation techniques have been applied in order to improve the processor throughput and reduce its energy consumption. The throughputs achieved are up to 200 Mbps and 92 Mbps for the encoder and decoder respectively. The associated dynamic energy consumption is in the range of 0.34 to 0.6μJ, demonstrating a design suitable for present and future telecommunications systems.

Furthermore, a novel engine is presented based on the dynamically programmable platform (RICA) targeting the computationally intensive correlation function used in GPS-based positioning (Chapter 6). Various optimisation techniques have been exploited in order to achieve the best performance on the platform. In addition, modified correlation architectures are introduced, which demonstrate superiority in terms of correlation time and energy consumption. Furthermore, the bitwise optimisation technique has been applied for digital matched filters, which demonstrate the maximum utilisation of the architecture leading to a higher correlation speed of 62 μs for 1023 phase search correlations. Comparisons of the achieved results and other relevant architectural configurations are presented, showing that this work is a promising step towards high-speed, ultra-low-energy GPS receivers.

A novel optimised multi-correlation processor is then introduced (Chapter 7). It is concluded that, for the practical realisation of the multi-correlation engine, “Engine 2” and “Engine 3” provide the optimum solutions where 12 and 72 parallel correlation channels respectively are being calculated. Moreover, a compromise could be achieved through having an engine of three instead of six embedded “Engine 2”, which is the case in “Engine 3A”. The new “Engine 3A” would provide 36 parallel correlations. This work represents a step forward in the area of dynamically reconfigurable architectures and correlation systems.

The contributions above taken together show how a dynamically reconfigurable architecture can be used to dynamically reconfigure a device to perform the desired function at any given time. For example, if the WiFi system is on and then the mobile device moves out of the WiFi signal range, the processor will dynamically reconfigure the device to work as a WiMAX system. Then if the present location needs to be known, dynamic reconfiguration can get GPS positioning information before switching back to WiMAX or another function. These possibilities clearly show the potential power of dynamic reconfiguration for the next generation of mobile devices. In particular, this can be very effective for miniature systems such as wearable devices.

1.3 Structure

The structure of this thesis is as follows:

- Chapter 2 presents a review of research work in the area of reconfigurable architectures with a focus on their suitability for telecommunications systems
- Chapter 3 describes the reconfigurable architecture introduced so far that provide multi-rate punctured convolution coders.
- Chapter 4 presents a novel reconfigurable interleaver and its design as a reconfigurable fabric and on a dynamically reconfigurable architecture.

- Chapter 5 discuses the novel high-speed and low-power 4x8-bit Galois Field (GF) multiplier cell embedded within the novel low-power processor for programmable Reed Solomon coding, along with its design, optimisation and implementation.
- Chapter 6 presents a novel engine based on the dynamically programmable platform targeting the computationally intensive correlation functions used in GPS-based positioning systems.
- Chapter 7 discusses the multi-engine correlation and introduces the novel optimised multi-correlation processor.
- Chapter 8 gives a summary of and the conclusions drawn from the work.
- Appendix A contains the Matlab codes focusing mainly on the interleaver.
- Appendix B contains the Verilog code for the interleaver, GF cell and RS coder.

1.4 Summary of Contribution

- Proposed multi-rate punctured convolution coder with 100 Mbps throughput (Chapter 3).
- WiMAX interleaver reconfigurable fabric and WiMAX interleaver on dynamically reconfigurable architecture (Chapter 4).
- Proposed RS encoder architecture with parallel parity output (Chapter 5).
- Novel high-speed and low-power 32 bit GF multiplier cell (Chapter 5).
- Novel real-time low power processor for programmable Reed Solomon codec (Chapter 5).
- Modified matched filter designs (Chapter 6).
- Novel GPS correlation engine (Chapter 6).
- Multi-correlation GPS engine for 12 and up to 72 parallel correlations (Chapter 7)

Chapter 2

RECONFIGURABLE ARCHITECTURES

2.1 Introduction

The development of mobile devices has challenged hardware designers to come up with suitable architectures. Challenges such as power consumption, flexibility, processing power and area are likely to lead to the need for a reconfigurable architecture to cater for the growing demands made of mobile devices and to suit the needs of the next generation of devices. Parallelism and multifunction in real-time will be the minimum required characteristics of the architectures of such devices. This chapter reviews the currently available reconfigurable architectures. The focus here is on coarse-grain reconfigurable architectures, with particular attention to those which support dynamic reconfiguration with low-power consumption. The capacity for dynamic reconfiguration will be a key factor in defining the most suitable architecture for future generations of mobile devices.

This chapter describes existing reconfigurable platforms. Their principles of operation, architectures and structures are discussed highlighting their advantages and disadvantages. Various coarse-grain reconfigurable architectures are discussed along with their improvement with time. Finally the key characteristics which are required for a reconfigurable architecture to be suitable for telecommunication systems are identified and these are then the subjects of the following chapters in this work. A comparison is given of for the various architectures discussed in terms of suitability for telecommunications applications. The selected architecture will be the subject pursued in this work.

2.2 Background

The origins of reconfigurable computing date back to the 1960s, the concepts proposed by Gerald Estrin [1]. The first FPGA (field programmable gate array) was introduced by Carter et al [2]. Before the 1980s, software programmed microprocessors were the only available resource for providing flexibility. The emergence of the FPGA changed this situation. Configuration bits changes the hardware realisation in FPGA as software instructions is programming the processor. This led to another definition of reconfigurable computing, as a system incorporating programmable logic to customise existing hardware. Programmable logic is connected by flexible interconnects which can be changed periodically to execute different implementations on the same hardware, thus providing an ASIC (application-specific integrated circuit) solution with post-fabrication programmability.

The FPGA consists of two main components, which are logic blocks and interconnections or switches as illustrated in Figure 2-1. The programmable logic blocks can be programmed along with the interconnections or switched array to perform a certain logic function. The programmable logic block of one of the leading FPGA manufacturers, Xilinx consists of a 3-input look-up table (LUT), a multiplexer and a flip-flop in its basic building block [3]. Nowadays, Xilinx's series seven includes more complicated FPGAs with embedded processors, large

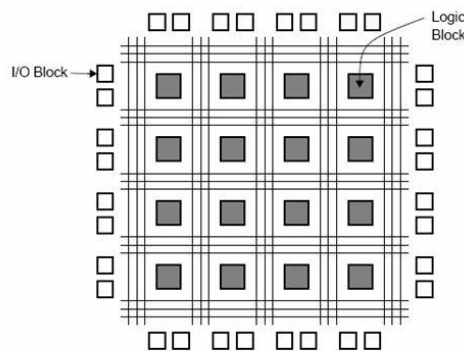

Figure 2-1 FPGA building block - Xilinx

memory blocks and even transceivers along with various interface protocols [4]. The FPGA is a fine-grain reconfigurable architecture where it is based on a single bit operation. FPGAs require a high volume of configuration data, and the mapping functions are difficult and require specific skills.

The FPGA has various advantages, such as quick prototyping, speedy development and clear basic blocks. However, it is obvious that it has various drawbacks, such as being fine-granular. This property means that configuration data are complicated, which as a consequence leads to the need for a large configuration memory. This in turn increases the power consumption, area and design complexity.

Architectures that use FPGA-based coprocessor along with a general purpose processor are capable in processing complex algorithms [5]-[7]. However, this type of configuration has two main drawbacks. Firstly, a wider datapath requires a large area and longer delays occur due to the small width of the programmable logic block and secondly FPGAs have lower logic density and are slower than a custom ASIC [8].

In coarse-grain architectures, the datapath ranges from 2 to 32 bits or more. The selection of the datapath width is a trade-off between flexibility, efficiency and programmability. The main advantage of a reconfigurable processor or functional unit is the ability to customise hardware for the requirements a specific algorithm or function.

2.3 General, DSP, Fine and Coarse Processors

Multiprocessors or multi-core systems are increasingly introduced in personal computing and smart mobile devices. There have been several approaches to the enhancement of general-purpose architectures in order to increase performance, such as the SIMD (single instruction, multiple data), MIMD (multiple instructions, multiple data) and VLIW (very long instruction word) methods.

Reconfigurable computation can enable increased computational performance and lower energy consumption. Configurable computing combines the performance of application-specific hardware with the reprogrammability of general-purpose computers [9]. The FPGA may have advantages in reconfigurable computing; however, it is based on a single bit which limits its capabilities compared to ASIC. Such limitations can include low arithmetic density, reduced clock speeds, and low internal RAM density and bandwidth, as well as the cost of higher reconfiguration times. This is in addition to the large power consumption and large area, which are crucial parameters in today's and tomorrow's compact designs [10].

Figure 2-2 presents a distribution of the computation architectures in relation to flexibility, area, power consumption and performance. It is clear that ASIC is best in terms of performance and having the lowest power consumption and area, while on the other hand the general purpose processor has the highest flexibility but also suffers having being the highest power consumption, lowest performance and largest area. The focus in this chart is the highlighted dotted rectangle at the bottom right-hand side, where low power consumption and area are the main characteristics. From the point of view of flexibility within this zone, then the coarse-grain reconfigurable architecture gives the best of both worlds in terms of power consumption, flexibility, area and function diversity. This would appear to be the privileged space for architectures that could satisfy the requirements of

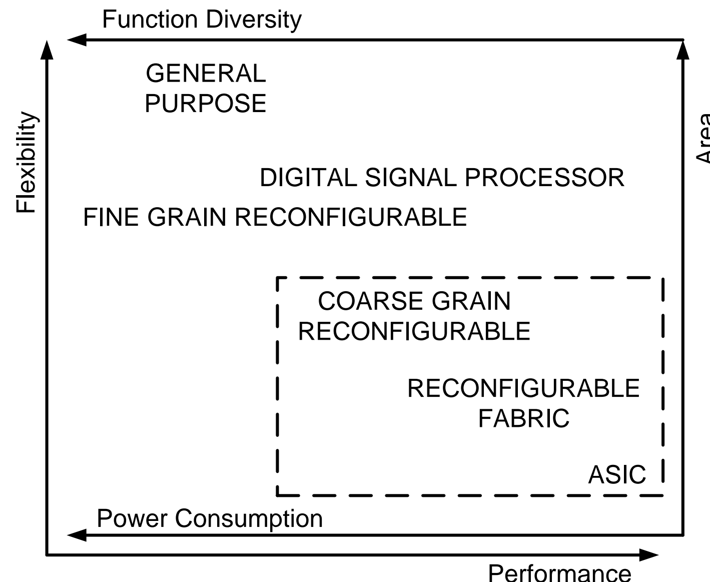

Figure 2-2 Computation architecture characteristics

mobile communication systems.

From the point of view of functional diversity microprocessors or general purpose processors can achieve more functions than FPGA and other reconfigurable architectures. On the other hand, a reconfigurable architecture can achieve higher performance than processors on highly repetitive computing tasks with limited functional diversity [5].

The rapid increase in demand for computation load has resulted in a number of accelerator styles. These can take the form of specialised extended instruction-specific processors, custom hardware, intense kernel codes, and reconfigurable computing. Such accelerators can be implemented as independent processors, co-processors or customised IP [10].

Wireless applications need processing modules that simultaneously demonstrate high computational performance, ultra-low-power consumption and a high degree of flexibility and adaptability. Reconfigurability is a necessity in the presence of multiple and evolving standards in dynamic conditions. The computing challenges for mobile devices are area, power and computing power efficiency.

To increase computing power, approaches such as larger processors, dedicated fabrics with application-specific cores, and reconfigurable computing have been considered. Most computationally complex applications spend 90% of their execution time on only 10% of their code [19].

2.3.1 Reconfigurable Computing Classes

As mentioned earlier, a key interest in discussing reconfigurable architectures is their capacity for reconfiguration and their flexibility. From the literature, it is clear that there are different types of reconfiguration. In partial reconfiguration (PR) only a part of the reconfigurable fabric is reconfigured and there are two types, static and dynamic. The meaning of static partial reconfiguration (SPR) is clear from its name; however, dynamic partial reconfiguration (DPR) usually requires an external configuration control module. An improvement upon DPR is dynamic partial self-reconfiguration (DPSR), which does not require an external configuration control

module. Various reconfigurable architectures are discussed in the following sections.

2.4 Coarse Grain Reconfigurable Architectures

Reconfigurable architectures are discussed in this section and their suitability for communication systems is considered.

2.4.1 CRISP: A Coarse-Grained Reconfigurable Instruction Set Processor

Francisco et al. [13] presented a coarse-grain reconfigurable processor named CRISP, which consists of a processor and a reconfigurable logic. Thus it is based on the concept of a co-processor for the reconfiguration part or as an add-on functional unit. The architecture is illustrated in Figure 2-3. The reconfiguration functional unit is activated through a special reconfigurable instruction from the main processor. The architecture targets multimedia applications, with performance

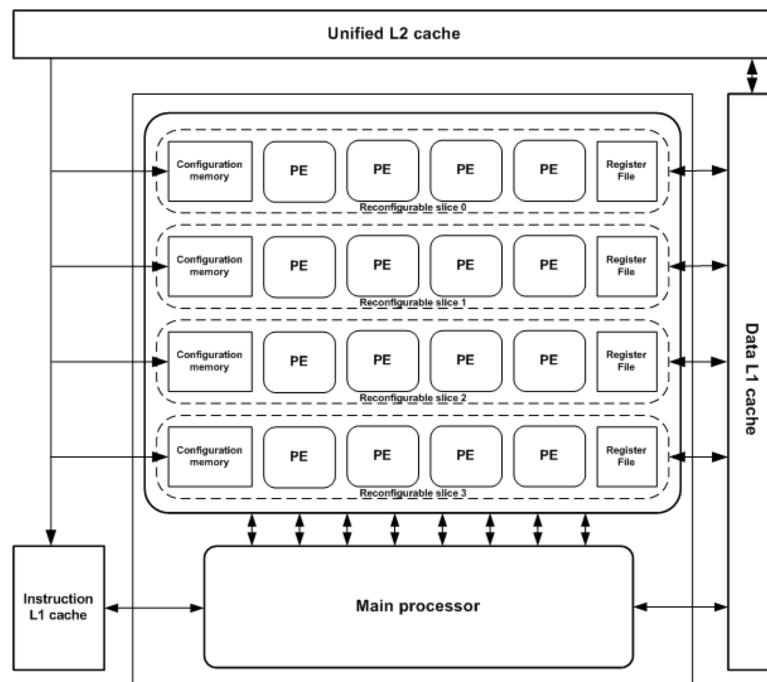

Figure 2-3 CRISP architecture construction of processor and a reconfigurable functional unit [13]

claimed by the authors to be 2.5 times that of a RISC processor with an 18% energy overhead [13].

This architecture operates on 8 to 32 bits. The reconfigurable logic in this architecture consists of reconfigurable slices. Each slice contains several processing elements (PEs), a register file, a programmable interconnect and a small configuration memory. The internal elements of the single reconfigurable slice are shown in Figure 2-3. Each PE can be an ALU (arithmetic logic unit), a shifter, multiplier or memory unit. The interconnection used in this architecture is a full crossbar.

The CRISP architecture is novel in that its reconfigurable part is based on two levels or layers, the slice and PE layers. From one point of view, this appears to be incompatible to other mainstream reconfigurable architectures where usually the first layer is the PE and underneath it can be another layer. However, from a power savings point of view it is an interesting concept, since the inactive slice will be turned off completely along with all embedded PEs. From the programmability, mapping and computation distribution points of view, it appears that it is quite complex to realise functions on this architecture. The slices appear to be integrated reconfigurable processors where, by default, the results of all PEs have to be written into the register file of the slice prior to interaction with the outer world such as other slices, memory or the main processor. The authors claim that the interconnections could be configured in such a way as to allow direct interactions between PEs within different slices [5] and [13]. The realisation of this will be complex from a mapping point of view and costly in terms of configuration time. In conclusion, the architecture is appealing for its power saving ability when unused

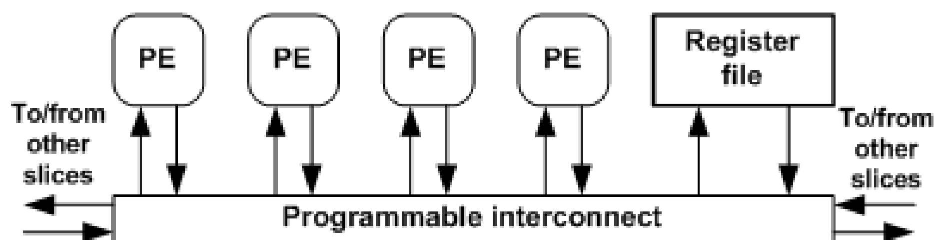

Figure 2-4 Reconfigurable slice internal structure [13]

slices are disabled and its ability to execute loops with multi-integrated configurations. The cost arising from the latter is degraded performance if the number of configurations would surpasses configuration cache limitations.

2.4.2 Systolic Ring Architecture

The systolic ring architecture is a coarse-grained arithmetic block which includes a custom RISC (reduced instruction set computing) processor [14]. The role of the processor here is to dynamically configure the architecture and control dataflow at the operative layer. The architecture is divided into operation and configuration layers, as illustrated in Figure 2-5. The operation layer is the reconfiguration part where the processing elements reside, while the configuration layer consists of RAM that holds the configuration information which resembles a FPGA. The RAM contents change every clock cycle.

The PE of this architecture is called the Dnode (data node). It consists of an ALU, datapath components and a few registers. It is configured using micro instruction code. Figure 2-6 demonstrates the PE or Dnode architecture. Each Dnode has two execution modes, normal and standalone. In normal mode, the Dnode is in operation where it follows the micro-instruction code. The stand-alone mode allows the Dnode to take up to six clock cycles to process data or instructions located internally in its own seven registers.

This architecture is called a "ring" due to the fact that the Dnodes are arranged in a ring style or pipelined systolic structure. Each two adjacent Dnodes create a layer and can interact with neighbouring layers through the switches shown in Figure 2-7. The length of the structure is the number of layers, and its width is the number of Dnodes per layer.

In this architecture the datapath (dataflow) is separated from data feedback or Dnode results. Data feedback passes through isolated dedicated pipelines through the switches as seen in Figure 2-7. Its designers claim that this technique of separation dramatically reduces routing problems and supports the architecture scalability [14]. It is clear that this architecture imitates the FPGA in having

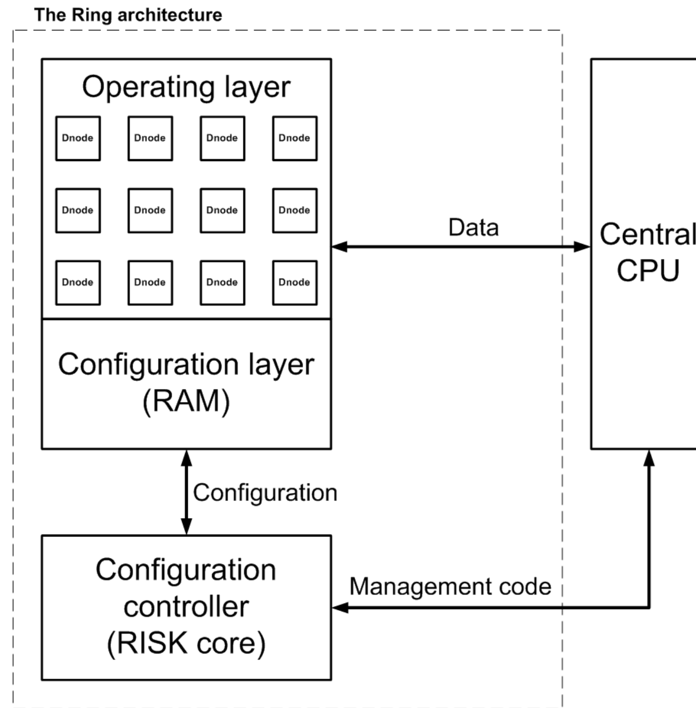

Figure 2-5 The Ring architecture layout [14]

operational CLBs and a configuration layer which is usually a large SRAM. This architecture is a clear step forward for coarse-grain reconfiguration and its interconnection principle is interesting. However, mapping would be a complicated task given the complex ring structure used for the connections. Most importantly, the usage of RAM will clearly increase area and power consumption of the architecture. Also the architecture is clearly complex from an implementation point of view, due to the need for an embedded RISC processor within the architecture just to drive the configuration, while the whole architecture has to interface with an external processor in order to act as a co-processor as illustrated in Figure 2-5.

2.4.3 MATRIX Architecture

The MATRIX (multiple ALU architecture with reconfigurable interconnect experiment) architecture is built according to an application-specific methodology, aiming to be suitable for general purpose applications [15]-[16].

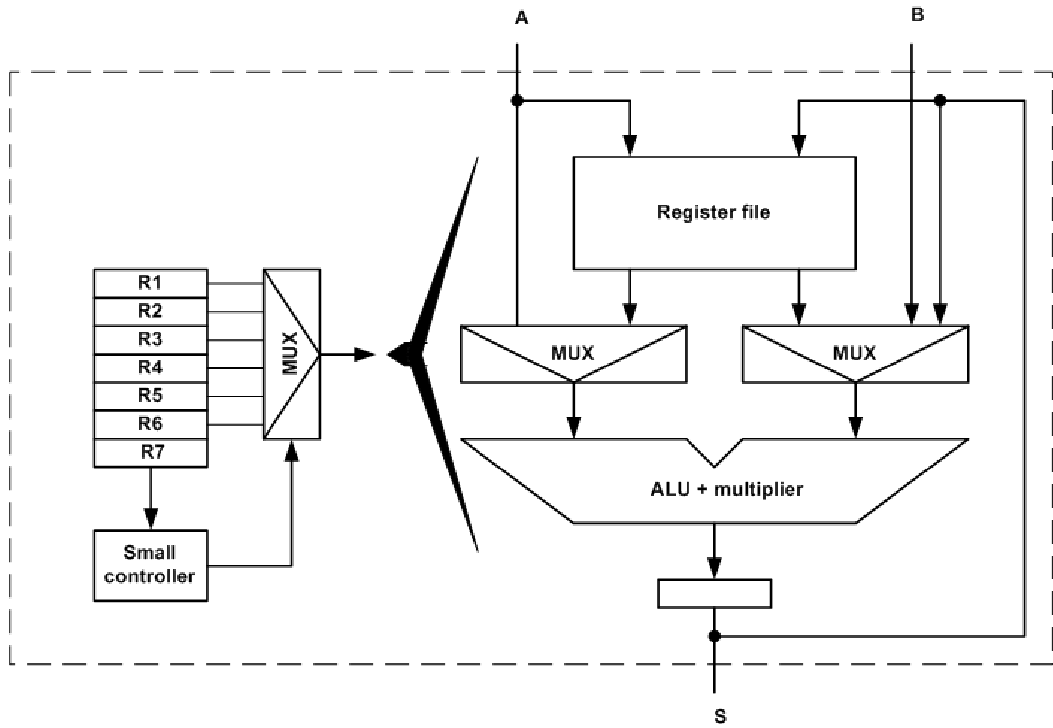

Figure 2-6 Dnode architecture [14]

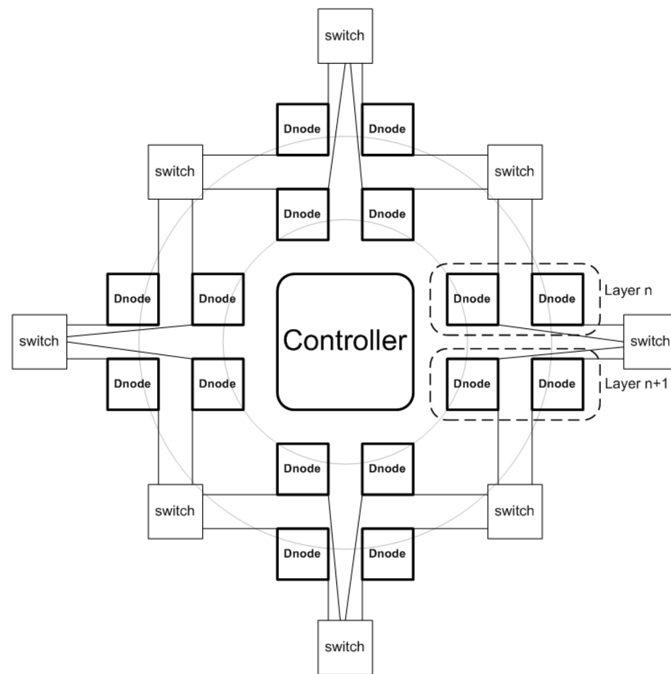

Figure 2-7 Ring style for Dnode interconnections [14]

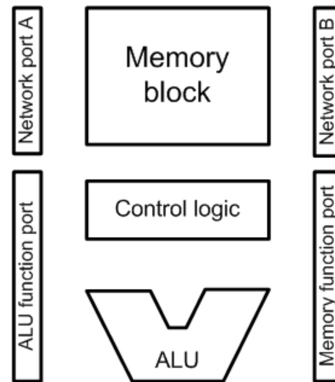

Figure 2-8 MATRIX Processing element (functional unit) [15]

The architecture is composed of an array of identical 8-bit basic functional units and a configuration network. The basic functional units (or processing elements) in this architecture include the following three main components as shown in Figure 2-8: 256 x 8-bit memory; an 8-bit ALU and multiply unit, and a control logic.

A multiply operation takes two operating cycles. The architecture is by default pipelined due to the existence of a pipeline register at the input port of each function unit.

The interconnection used in this architecture is almost a crossbar style interconnection network. It has the capability of connecting nearest neighbours. Also it has four bypass connections and global lines. Global lines imply the usage of the four interconnect lines. Nearest neighbour interconnections can allow a single processing element to have direct connections with up to 12 neighbouring PEs as shown in Figure 2-9.

One key aspect of this architecture is the port programmability of the basic function unit. The port configuration can be a holder of the input values of the ALU and this is termed static value mode. Meanwhile, in the static source mode, the word hold in the port is used to select the network bus from which data can be received. Another mode for the port configuration is the dynamic source mode where the port configuration word is ignored and the associated floating port controls the input source on a cycle-by-cycle basis.

Another attractive point in this architecture is its ability to be configured in order to operate VLIW, SIMD, MIMD, MSIMD or hybrids of these. Moreover, the architecture datapath can be wired up in an application specific manner.

The architecture's authors claimed that no specific applications are targeted and that the architecture can be a general purpose one [16].

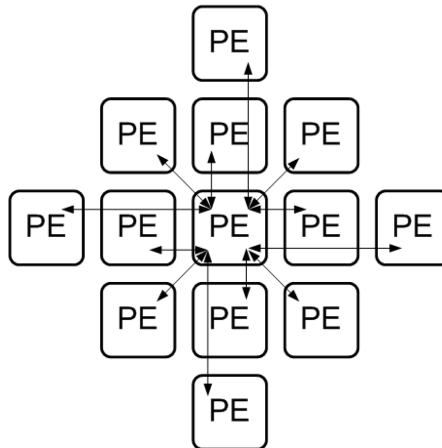

Figure 2-9 MATRIX nearest neighbour interconnect [15]

2.4.4 Cell Matrix and vCell Matrix Architectures

The vCell Matrix architecture [19] is based on a commercially available architecture named the cell matrix architecture [16]. The vCell architecture promises a simpler and faster reconfiguration mechanism compared with the cell matrix. Both architectures consist of two-dimensional homogeneous cell arrays.

The PE in the Cell matrix architecture is called a Cell and is illustrated in Figure 2-10, while the PE of the vCell matrix architecture is called the vCell and its structure is illustrated in Figure 2-11.

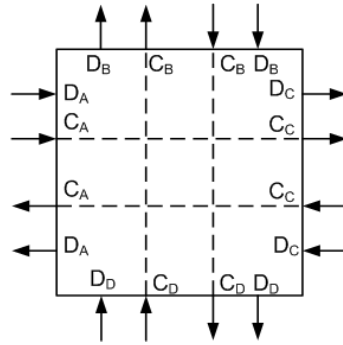

Figure 2-10 PE structure for the Cell matrix architecture [16]

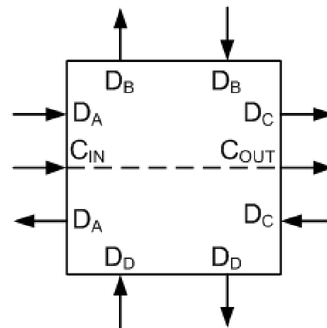

Figure 2-11 PE structure for the vCell matrix architecture [19]

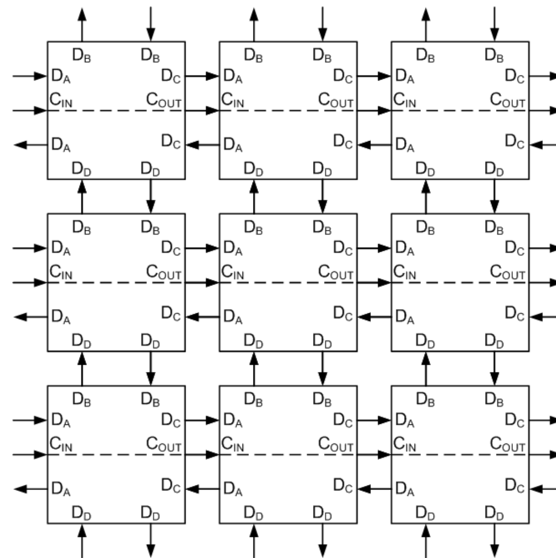

Figure 2-12 Two-dimensional array structure for the vCell matrix architecture [19]

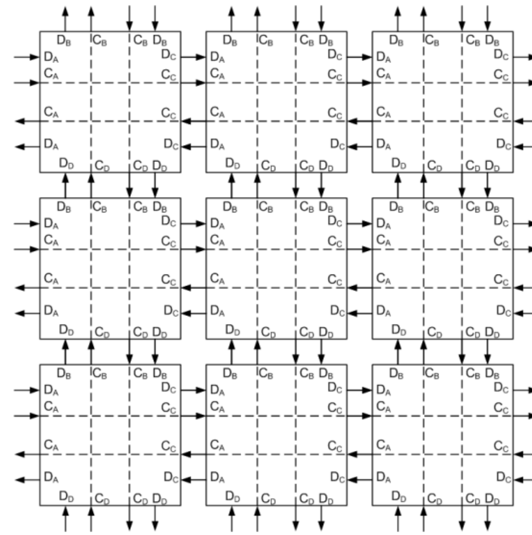

Figure 2-13 Two-dimensional array structure for the Cell Matrix architecture [19]

Each vCell has four input and four output data ports of 1-bit each distributed on its four sides ($D_{A\text{-}D}$). In addition, the vCell has two configuration control ports, one input and one output, named C_{IN} and C_{OUT}. When the input configuration control port is activated, it allows the vCell to store the configuration data through the data ports into its internal LUT. The output configuration control port allows the vCell to control the mode of operation of its neighbouring cells. Each cell is connected to its nearest neighbour to the north, south, east and west, as illustrated in Figure 2-12. In the Cell matrix architecture, the reconfiguration process is distributed, so that any cell can initiate the reconfiguration process by configuring its nearest neighbours; hence it supports dynamic partial self-reconfiguration (DPSR). The array structure of the Cell matrix architecture is illustrated in Figure 2-13, where each cell is capable of configuring its nearest neighbour. The cell needs to be configured first as a data bus in order to pass configuration data to the furthest cell. In this architecture, the configuration mechanism allows great flexibility; however, it requires a complex and sophisticated configuration algorithm.

Conversely, in the vCell matrix architecture, each vCell can configure only its eastern neighbour, as shown in Figure 2-12. In addition, the vCell cannot initiate the configuration process by itself. It has only two configuration ports, C_{IN} west and C_{OUT} east, whereas the Cell has eight configuration ports covering all the sides

of the cell. The reduction in the number of configuration ports in the vCell significantly reduces the architecture's configuration flexibility as compared with that of the Cell architecture. Despite this drawback, the reduced number of configuration ports has the advantages of a simpler configuration mechanism and a smaller LUT within each individual vCell. The Cell matrix architecture is intended for a wide range of applications, being general purpose. The vCell matrix architecture, however, is suitable for applications that require a regular datapath and a simple control path, and thus mainly DSP applications.

2.4.5 Pleiades Architecture

The Pleiades processor architecture is based on the combination of a main processor coupled with an array of heterogeneous computational units of various granulates [17]. The PEs here are heterogeneous computational units and are named satellite processors. In addition to the satellite-processors, the architecture includes a reconfigurable interconnect network. The architecture's layout is presented in Figure 2-14.

The processor runs on data intensive loops called "kernels." Synchronisation between the satellite processors achieved is by a data-driven communication protocol in relation to the kernels.

The architecture operates direct memory read/write. The mesh structure has a two-level hierarchical reconfigurable interconnect network. The architecture address generator can handle addressing issues in addition to nested loops with loop counters. It controls the dataflow threads from initiation until end.

Because the system is realised on a data-driven principle, synchronisation between the processing elements employs a two-phase self-timed handshaking protocol consisting of request and acknowledge signals. This is realised in asynchronous fashion.

The architecture data-links consist of a 16-bit fixed-width data word in addition to 2-bit control signals, while the configuration bus is 32bit.

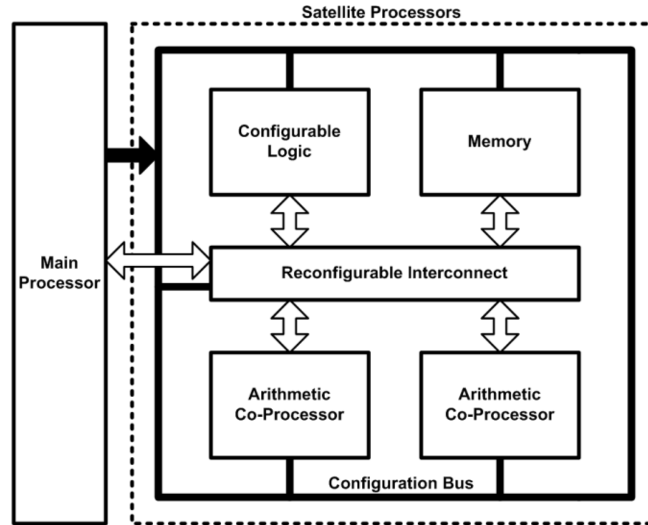

Figure 2-14 Pleiades processor reconfigurable architecture layout [17]

The hierarchical network architecture has sufficient connection flexibility for targeted applications and in addition it cuts the interconnect energy to a seventh of that of traditional crossbar network implementations [17]. This is achieved by having a universal switch box associated with each mesh level; in addition to cross-level interconnect switches so that only few buses are therefore required.

Targeted applications for this architecture are wireless devices and related baseband applications.

2.4.6 OneChip Architecture

The OneChip processor architecture is based on the combination of a fixed-logic processor core with large reconfigurable logic resources. This is illustrated in Figure 2-15, and the idea is to offer large reconfigurable resources with the core processor. This is the classic processor and co-processor co-located approach. The main drawbacks of the processor and co-processor approach are the limitations on processor-coprocessor bandwidth and the rigidity in control and interaction of the coprocessor [5].

The OneChip architecture consists of the integration of a 32-bit core RISC processor surrounded by the reconfigurable logic resources which are tightly integrated into the processor pipeline. In this architecture there are two distinctive

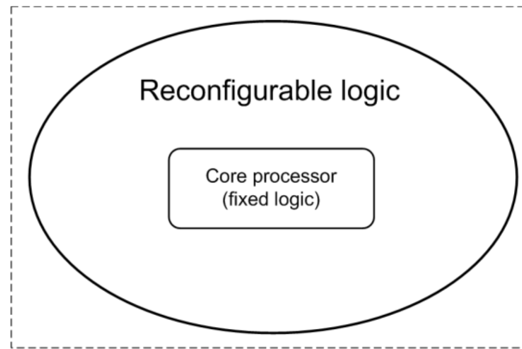

Figure 2-15 OneChip architecture layout

PEs, the basic functional unit (BFU) and the programmable functional unit (PFU). The BFU is responsible for basic functions, mainly arithmetic and logic operations; while PFU is more complex and can perform various functions in combinational or sequential form and in addition, it can work as glue logic whenever required.

It is worth mentioning that the OneChip architecture is an advanced version of the PRISC architecture [6]. The key difference is that the PRISC PFU only supports small combinational operations and is limited to one clock cycle operation. This leads applications of PRISC to be limited to bit-level applications. On the other hand, the OneChip architecture targets DSP applications.

2.4.7 Chimaera Architecture

The Chimaera architecture is based on the integration of reconfigurable logic into

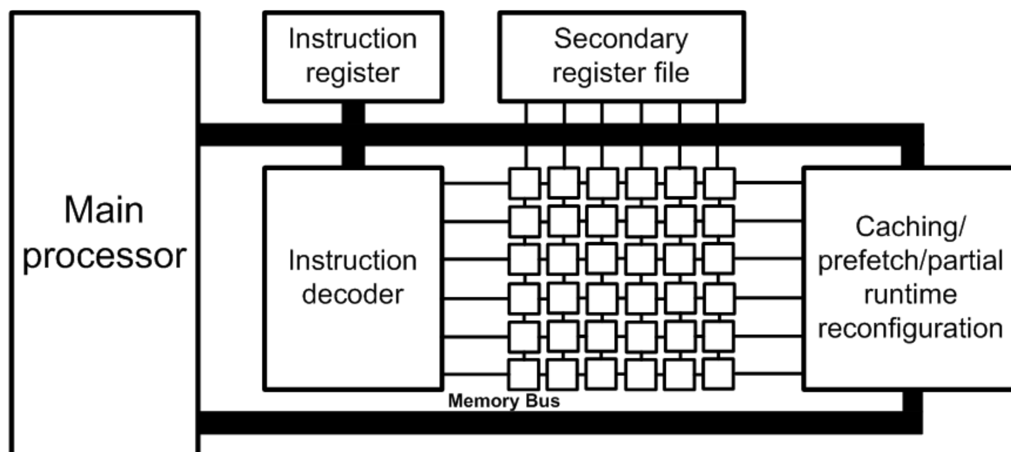

Figure 2-16 Chimaera architecture layout [7]

the host processor itself [7]. This allows direct access to the processor's register file and enables a set of new operands.

In addition, reconfigurable logic is always slower than the processor's own functional units when it comes to standard arithmetic computation. In the Chimaera architecture, the designers have integrated the advantages of the reconfigurable logic along with those of the processor. The processor is responsible for executing the bulk of the functionality while the most critical computation kernels are accelerated using the reconfigurable logic. Figure 2-16 represents the Chimaera architecture layout with the reconfigurable logic at its heart. The architecture comprises a microprocessor with an embedded reconfigurable functional unit, which is described as a miniaturised FPGA array. The reconfigurable logic in this architecture is considered to be a cache for reconfigurable functional unit instructions. This architecture can be classified as fine-grained, due to its reconfigurable fabric being 1-bit based. An interesting aspect of this architecture is its instruction decoder, which supports multi-output functions and the efficient implementation of complex operations. Another interesting aspect is the availability of partial run-time reconfiguration, where the reconfigurable functional unit functions as an operation cache holding necessary instructions for the current operations. Many applications could potentially use the Chimaera, since the aim is for it to be a general purpose.

2.4.8 REMARC Architecture

REMARC stands for reconfigurable multimedia array coprocessor, and this architecture is a reconfigurable coprocessor coupled with a main RISC processor. The coprocessor includes a global control unit along with 64 programmable logic blocks or nano-processors, which are the PEs in this architecture [8].

The main processor has three coprocessors memory management and exception handling; a floating point processor; and the REMARC co-processor.

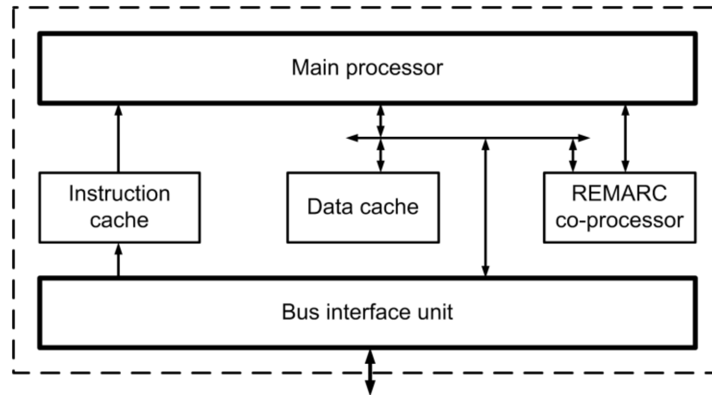

Figure 2-17 REMARC architecture layout [8]

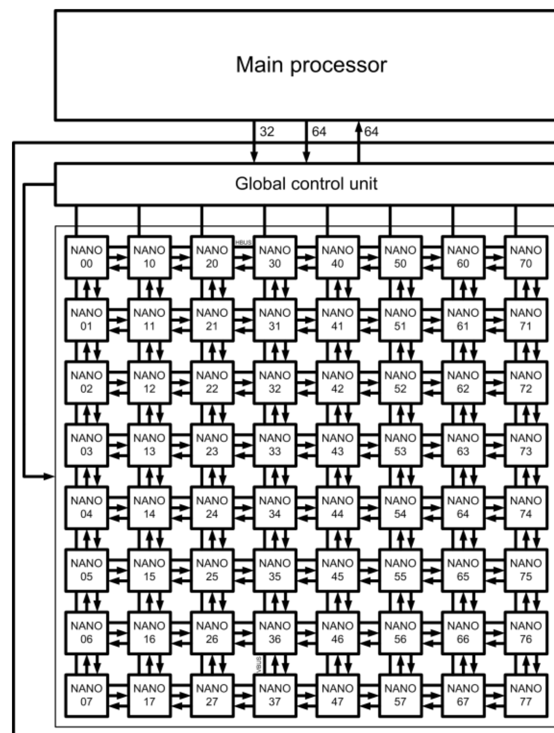

Figure 2-18 REMARC architecture internal structure [8]

Figure 2-17 shows the architecture layout while Figure 2-18 illustrates its internal construction of an 8x8 array of nano-processors and the global control unit. Each nano-processor has: 32 entry instruction RAM, a 16-bit ALU; 16-bit entry data RAM; 13x16-bit data registers; 4x16-bit data input registers; 1 instruction register; and a 16-bit data-out register.

Each nano-processor can communicate to the four adjacent nano-processors to the north, south, east and west. In addition, it can communicate with the processors in the same row and column through the 32-bit horizontal bus (HBUS) and the vertical bus (VBUS). The eight 32-bit VBUSes are also used for communication between the main the control unit and the nano-processors.

The nano-processors receive the program counter value from the global control unit, since it does not have its own program counter. Moreover, the function of the global control unit is to control the nano-processors and data transfer between the main processor and the nano-processors.

The REMARC architecture is a VLIW processor as its instructions consist of 64 operations. Its datapath is 16-bit and its targets multimedia applications such as image processing and video compression [8].

2.4.9 RaPiD Architecture

The reconfigurable Pipelined Datapath (RaPiD) is a coarse-grained FPGA architecture which is designed for computing intensive, repetitive tasks where configuration supports computation pipelines [9].

The architecture consists of an application-specific datapath and the program for controlling it. The interconnections are based on a linear array, in nearest neighbour style. The processor's functional units are placed in a linear array and formed of identical cells.

RaPiD-I is a prototype developed by the University of Washington, in which the cells or PEs consist of an integer multiplier, two integer ALUs, six general-purpose registers and three small local memories. The cells are interconnected with a series of buses as demonstrated in Figure 2-19. The inputs and outputs of the cells have multiplexers and de-multiplexers respectively to identify the specific buses to receive and send data.

RaPiD-I operates on 16-bit data. The datapath has registers which are used to store constant or intermediate values; however, these registers are costly in terms of area and utilisation. The control signals in this architecture are divided into static and dynamic signals. The former are used for initialisation and pipeline construction,

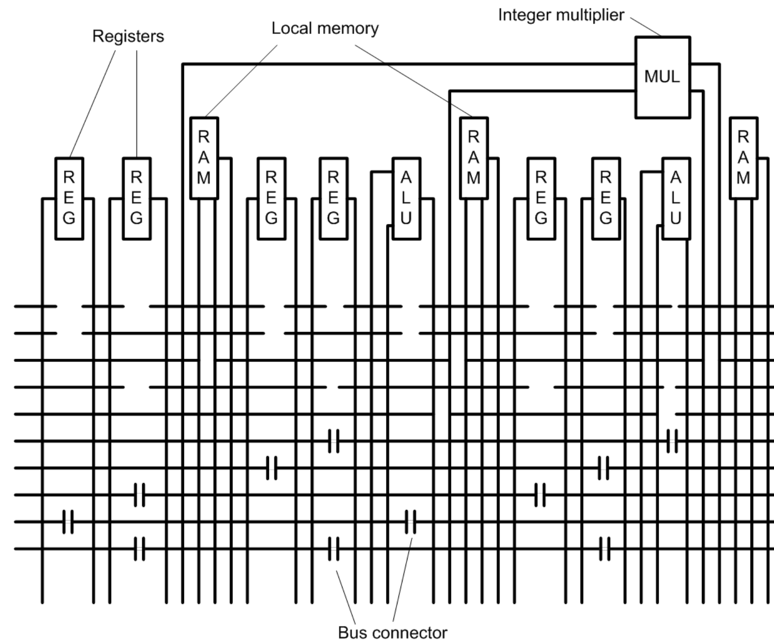

Figure 2-19 RaPiD-I basic cell structure [9]

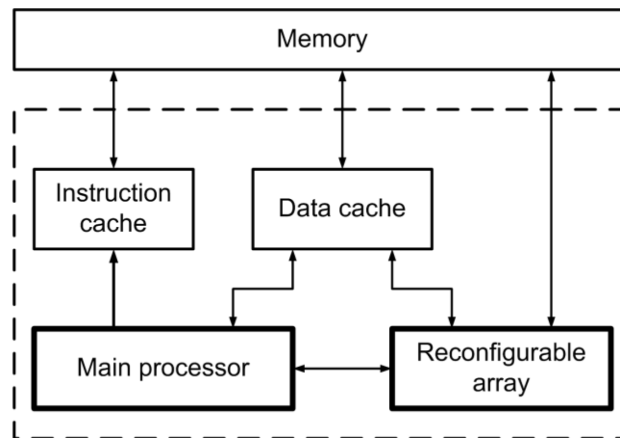

Figure 2-20 Garp architecture block diagram [18]

while the latter are used for scheduling computation information and are changeable in every cycle.

Around 34% of the cell control signals in the RaPiD-I are dynamic while the rest are static signals, giving a total of 230 control signals per cell.

The RaPiD is not suited for non-highly repetitive algorithms or those whose control flow is strongly dependent on data, such as in error correction, image processing or data encoding.

2.4.10 Garp Architecture

The Garp architecture is based on a combination of reconfigurable hardware with a standard MIPS processor [18]. This means that the Garp is reconfigurable architecture as a co-processor for executing certain parts of the code which are slower when running on the MIPS. A block diagram is presented in Figure 2-20.
The Garp's reconfigurable array is used to speed up functions, loops or subroutines that would be slower if run on the main processor. The reconfigurable array is fully controlled by the program running on the main processor.
The main processor instruction set has been extended for Garp. The processing element in the reconfigurable array is called a "block." The block is a logic unit which resembles the FPGA; however, at the start of the array row the first block is a control unit as illustrated in Figure 2-21. The array has a restricted 24 columns, while the number of rows is application-specific with a minimum of 32. The architecture's datapath is 2 bits, and thus to give a 16-bit operations at least eight logic blocks will be required. Usually these blocks are combined in a linear style in the same row.
The array has four vertical buses for interaction with the memory for the reception or transmission of data. In addition, these buses are used for array reconfiguration. There is an additional wiring network for data transfer between the different blocks. An innovative feature within this architecture is the availability of cache units within the array blocks, which hold recently used configurations in order to minimise costly memory access and allow faster switching between reconfigurations.

The Garp architecture has two clocks, one for the main processor and another for the reconfigurable array. The control block at the end of each row works as the interface control between the array and the main processor or main memory, and it can even interrupt the main processor. Each block requires 64 configuration bits in order to be fully configured. The configuration of the whole array is a lengthy and costly process in terms of energy and waiting time without execution. The latter is

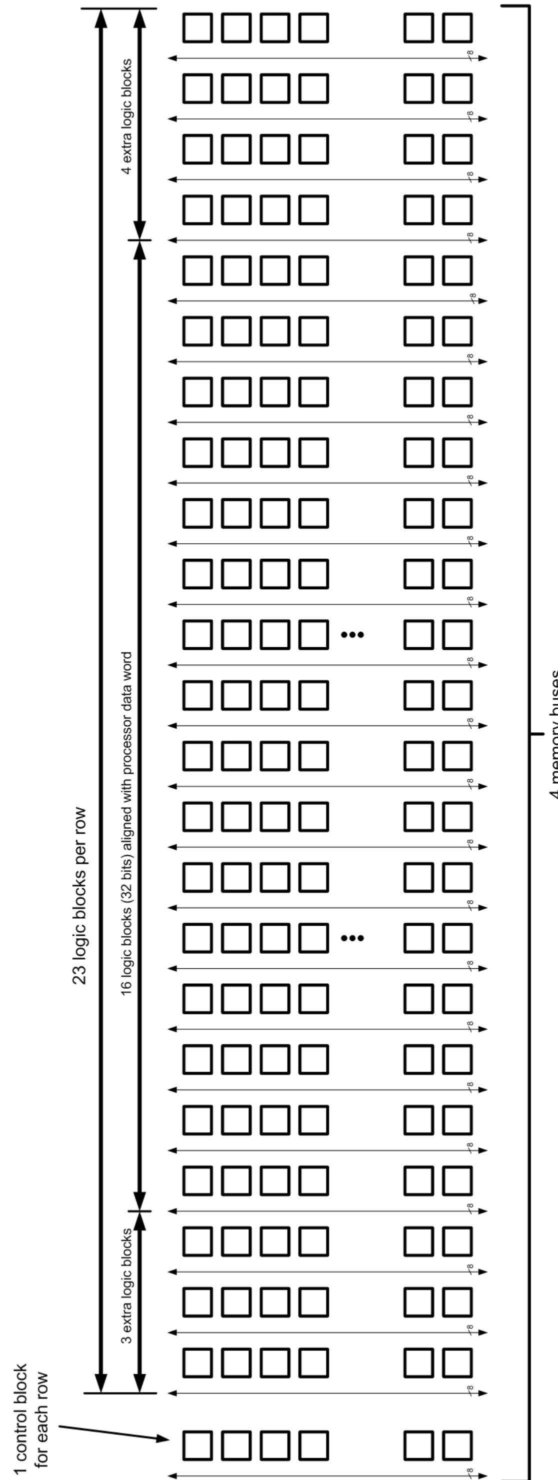

Figure 2-21 Garp array organisation [18]

assumed by the architecture's authors to be 50μs in order to complete the configuration load. An interesting aspect of this architecture is the ability of the

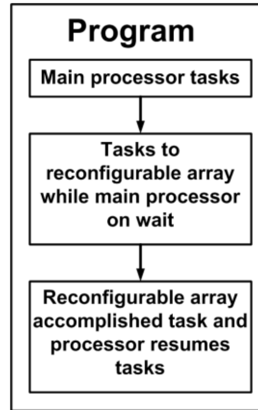

Figure 2-22 Garp program flowchart [18]

array to be partially reconfigured if only partial usage of the array is required. The architecture's minimum configuration is for a single row.

2.4.11 SRGA Architecture

The self-reconfigurable gate array (SRGA) architecture consists of an array of processing elements [11]. The processing element consists of a logic cell and memory block. The logic cell contains a 16-bit LUT and a flip-flop. The memory block can store one or several configuration contexts as well as data for the logic.

Processing elements are connected to their four nearest neighbours in addition to the mesh of the tree network. With this network context switching and memory access operations can be performed in a single clock cycle. Each PE has two switches a row switch and a column switch, as demonstrated in Figure 2-23.

Self-reconfiguration in this architecture is mainly used to allow each logic cell to modify its own configuration at run time. Therefore, instead of having an external or centralised reconfiguration controller, this architecture has a distributed reconfiguration capability integrated within each PE. This gives the device fast memory access and context switching.

To achieve self-configuration, a very complex interconnection network is required which consists of a logic interconnection network and a memory interconnection network, in addition to the switching network at each node.

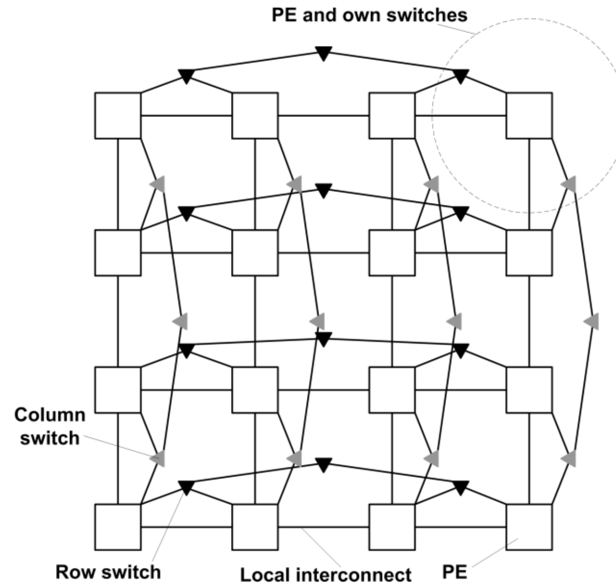

Figure 2-23 SRGA architecture interconnect mesh [11]

2.4.12 CHESS Architecture

The CHESS architecture was developed by Hewlett-Packard (HP) laboratories targeting multimedia applications. This architecture is intended to be an ASIC IP or a unit of the processor datapath [10].

The PE in this architecture is a 4-bit ALU with a set of 16 primary instructions. The instructions can be constant and stored within the configuration word or dynamic and generated through an external circuitry feeding into the instruction input of the ALU or PE.

The datapath of this architecture is 4-bit, allowing much flexibility and a wider range of datapath applications compared with 1-bit architectures (FPGA).

The architecture has a switch box which has a dual functions based on the mode of operation. First it may act as a cross-point switch, allowing 64 connections by connecting vertical and horizontal buses. Secondly, it may be a RAM of 16 words x 4 bits using the 64-bits configuration.

The architecture's layout has PEs arranged as a chessboard in a symmetrical fashion. This increases neighbourhood connectivity, and also reduces the routing network complexity due to the maximisation of the number of neighbours in close

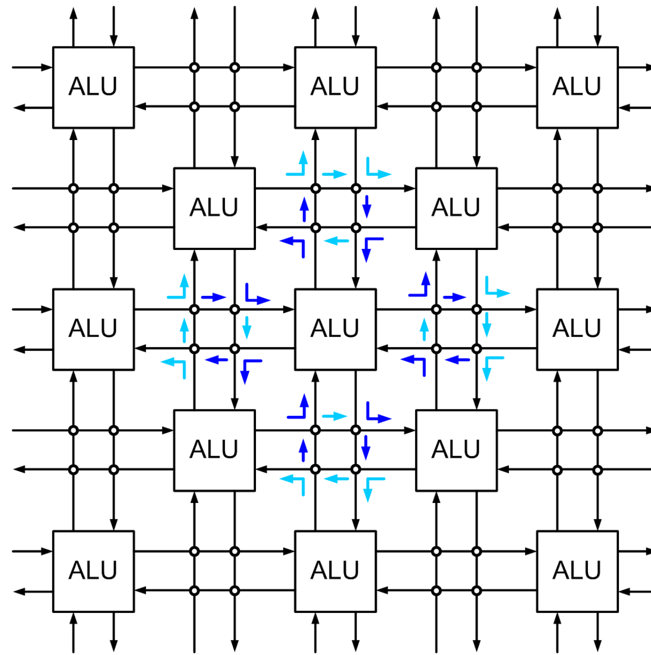

Figure 2-24 CHESS architecture layout and neighbouring interconnections [10]

proximity to each other, as illustrated in Figure 2-24. This proximity allows each ALU to have input and output buses on all four sides, enabling data transmission and reception from any of the eight surrounding ALUs. In addition, the architecture provides embedded evenly distributed block RAMs of 256W x 8-bits per 16 ALUs. The pipelining support increases throughput and efficiency of the architecture. There are two registers or buffers for each switchbox, and this is particularly helpful for long connections in order to avoid limiting the clock speeds of the entire architecture. Each PE has 100 configuration bits, which allows fast reconfiguration. The architecture does not support partial reconfiguration; however, it has the usual offline reconfiguration in addition to the ability to alter changes in functionality at runtime through feeding instructions into the ALU in a cycle-by-cycle approach.

One of this architecture's advantages is the capability of the switch boxes to act as memory (RAM) of a reasonable size. In other architectures, the conventional technique is to use the PE's own configuration bit as a memory, and this is usually small and thus limits the architecture's capability.

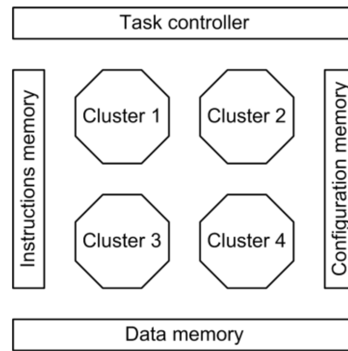

Figure 2-25 DART system level architecture

The PE or ALU can be combined with an adjacent switch box to provide a 16W x 4-bit memory. In this architecture the routing scheme is large since it uses 50% of the array area; however, this is less than in most FPGAs.

2.4.13 DART Architecture

The DART architecture is intended to be a reconfigurable architecture for telecommunications applications. The authors claimed that the architecture can handle complex processing tasks of third generation telecommunications systems in an efficient and low-power manner [12]. The architecture can be broken down into independent processing units named clusters. Those clusters can work independently or in cooperation with other clusters, as illustrated in Figure 2-25. The top level or cluster level architecture shows the main controller that is responsible for distributing tasks to specific clusters to execute. Then each cluster has its own embedded controller to manage the internal processing of the task allocated to it.

This architecture uses a hierarchical interconnect network, which is more suitable, smaller and less complex than a global interconnect network.

Figure 2-26 shows the interesting feature of the DART architecture that it has two PE types inside each cluster; a reconfigurable datapath (DPR) and an FPGA core.

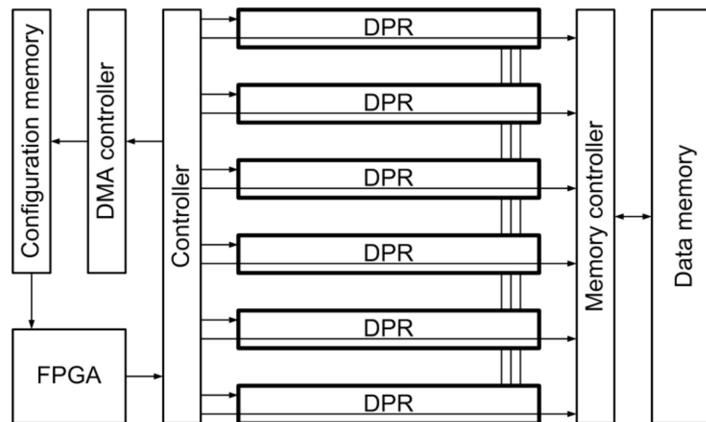

Figure 2-26 DART cluster construction [12]

Each cluster consists of one FPGA core and six arithmetic processing primitives (DPR). Each DPR has four functional units of two ALUs and two multipliers. The functional units are dynamically reconfigurable. As illustrated in Figure 2-26, there is an interconnection between the DPRs, so they can be configured to work together or work independently (in parallel).

In this architecture, there are three modes of reconfiguration. In dynamic reconfiguration mode, the interconnections within the cluster are reconfigured according to the calculation pattern. In hardware reconfiguration mode, this is the kernel configuration and it take four cycles and requires a large amount of data; this is the regular ongoing configuration process. Software reconfiguration mode concerns only the functionality of operators and is used for irregular processing where the DPR configuration needs to be changed.

This is clearly a very interesting architecture on various levels: firstly in its combination of FPGA and reconfigurable processing elements; secondly for the reconfiguration modes available, including dynamic reconfiguration; and thirdly given that its area of application is in the same domain as that addressed in this thesis.

2.4.14 DReAM Architecture

The DReAM (dynamically reconfigurable hardware architecture for mobile communication systems) is a coarse-grained architecture dedicated for wireless

communication applications [19]. This architecture has been designed to work within a system or system-on-chip, which means that the architecture would require a DSP, memory, controllers, and so on. The architecture has been developed at the Darmstadt University of Technology.

As illustrated in Figure 2-27, the DReAM architecture consists of an array of reconfigurable processing units (RPUs) interconnected through local and global connections. In addition, there are dedicated input and output ports at the borders of the architecture for data and control interfaces. Within the array, each RPU is able to connect directly to its nearest four neighbours through the local communication network. Moreover, within the array each four processing units share one configuration memory unit.

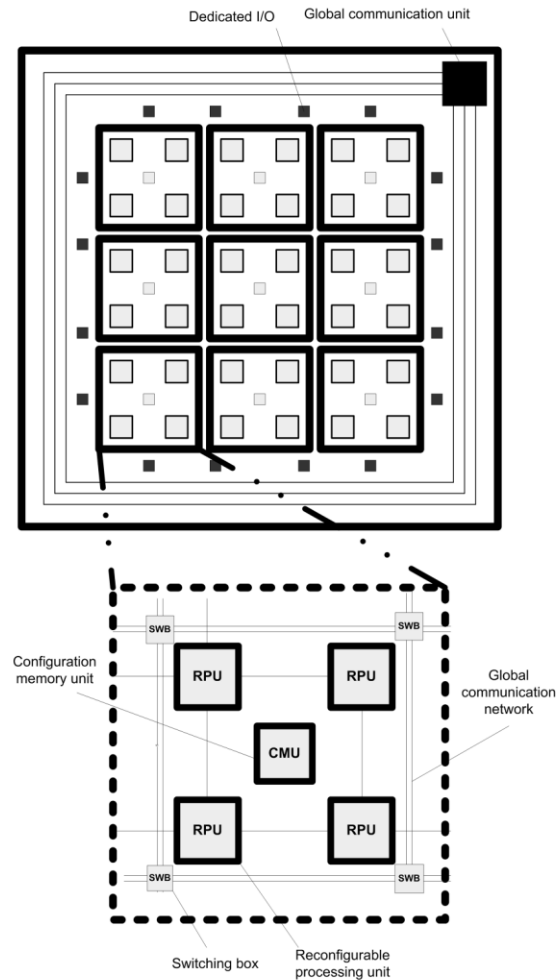

Figure 2-27 DReAM architecture structure

The RPU is able to execute data-flow arithmetic, data manipulations and finite state machines for the control-flow. Each RPU contains two reconfigurable arithmetic processing units, one spreading datapath unit, two dual-port RAMs and one communication protocol controller. An interesting aspect of this architecture is the dedication of a specific unit the spreading datapath unit to processing correlation operations for the communication standards quadrature phase shift keying (QPSK) and bi-phase shift keying (BPSK) required for code division multiple access (CDMA) systems. The correlation process is based on generating PN codes within the unit, which require a PN-code generator. This makes the architecture suitable for the communication applications targeted. However, having such dedicated and fixed units in every RPU within the array, despite the fact that it may be neither used nor required, represents a waste of resources in terms of area and power.

The RPU is the processing element in this architecture and the architecture's datapath is 8-bit based. Despite this 8-bit limitation, the PRU unit is capable of addressing a higher number of operands through the manipulation of the RAM as a LUT.

The architecture is built on a hierarchical concept where the global communication unit is the controller for the whole architecture as illustrated in Figure 2-28. The

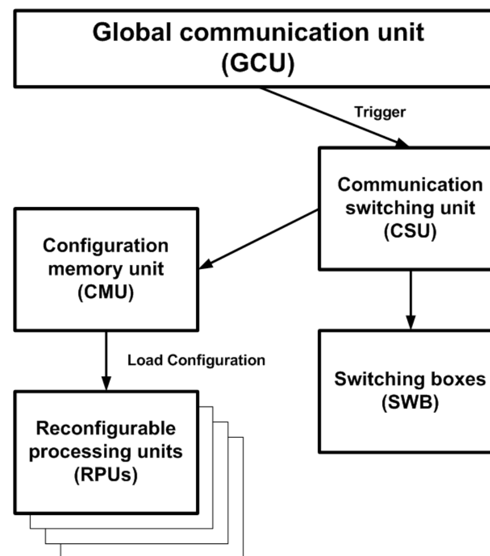

Figure 2-28 DReAM hierarchy control for dynamic reconfiguration [19]

designers claimed that they used this approach as it offers a trade-off between area and configuration performance. However, the hierarchical concept not only adds complexity to both the system and its control, but it also increases its area and power consumption. It is possible for the dynamic reconfiguration in this architecture to occur in different scenarios, including during run-time by conducting partial reconfiguration.

2.4.15 PADDI Architecture

PADDI stands for programmable arithmetic devices for high speed digital signal processing architecture. The PADDI architecture was first introduced in 1990 [20]. The architecture targets the rapid prototyping of high-speed data paths for real-time digital signal processing applications.

The PADDI architecture contains a cluster of eight EXUs or processing elements [21]. The interconnection between the EXUs is a crossbar-based network termed a switch, which is dynamically reconfigurable, as shown in Figure 2-29.

EXUs can be configured into two modes, 16 or 32-bit wide. Each EXU contains two register files each of which contains six registers. Registers are used for temporary data buffering, and the register files have dual ports for simultaneous read and write operations. Figure 2-30 represents the internal architecture of the EXU or processing element.

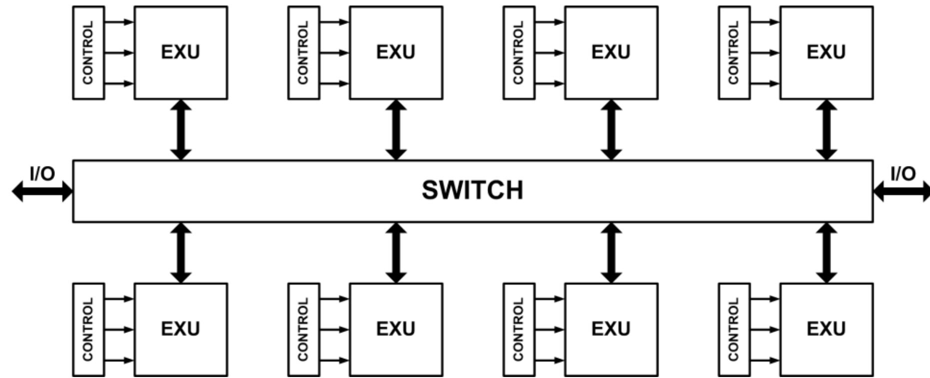

Figure 2-29 PADDI architecture structure [20]

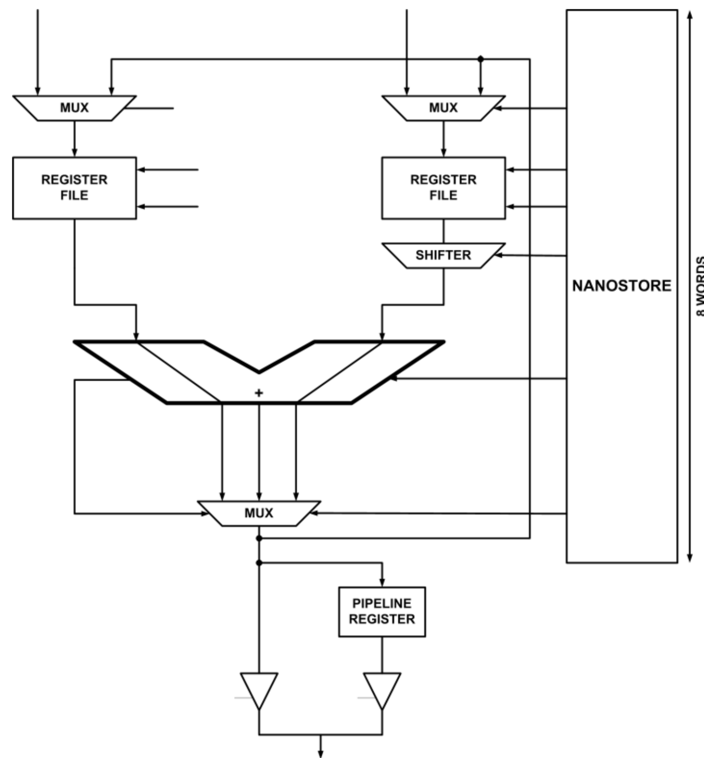

Figure 2-30 PADDI EXU architecture [20]

Each EXU needs a 53-bits control word. The Nanostore holds words of 8 bits necessary for controlling the eight EXU units. A global controller is responsible for feeding or loading the instruction word into each Nano-unit. The programming for this processor takes place using a high-level data flow language, “Silage” [22].

2.4.16 MorphoSys Architecture

MorphoSys is a reconfigurable architecture developed at the University of California targeting computation-intensive and high-throughput applications [25]. The architecture comprises a reconfigurable array, core processor and memory interface as illustrated in Figure 2-31. The reconfigurable array acts as a SIMD coprocessor and is responsible for exploiting the parallelism available in the application's algorithm.

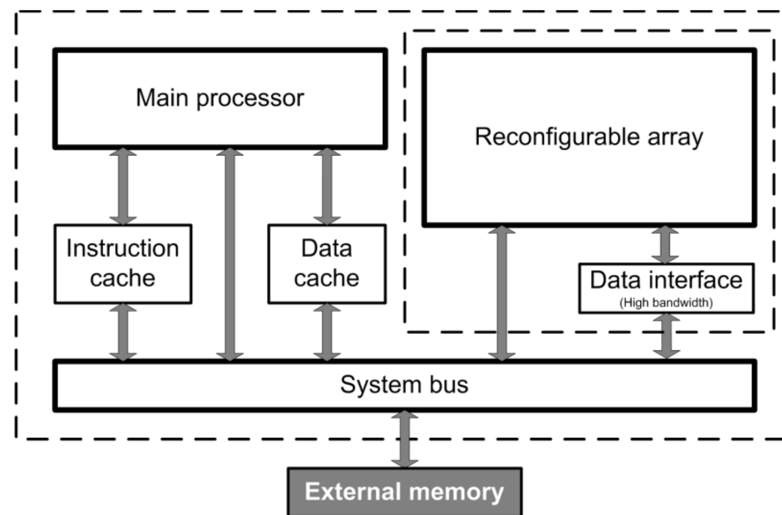

Figure 2-31 MorphoSys architecture layout [25]

The key component in this architecture is the reconfigurable cell array. The array consists of 8 x 8 reconfigurable cells (RCs), each of which consists of an ALU, a multiplier, a shifter and a register file and it is configured by a 32-bit context word stored within the array context memory. Each RC is connected to all of its neighbours in the same quadrant in both row and column directions in addition to the interconnection between the neighbouring quadrants as shown in Figure 2-32. All eight RCs in the same column or row are configured by the same context word; however, each operates on different data.

Dynamic reconfiguration is supported in this architecture and takes place through having context data loaded into an inactive part of the context memory without

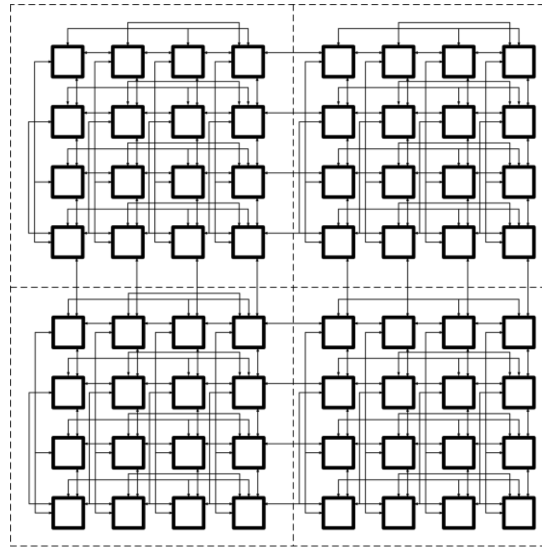

Figure 2-32 MorphoSys 8x8 reconfigurable cell array and row-column connectivity between each reconfigurable cell (RC) [25]

interrupting the array operation. Interconnectivity here within the array is based on the use of a 2D mesh and hierarchical bus network.

the MorphoSys system can operate on 8 or 16-bit data, despite the fact that the architecture's RISC processor is 32-bit.

This architecture has some clear features and advantages that have been highlighted earlier; however, there are also some drawbacks. The extensive use of multilevel memories has a significant effect on the processing times and power consumption.

Moreover, the processing element or RC is complex and sophisticated which implies a significant effect on the architecture's area and power consumption. In addition, all RC units in a single row or column have the same functionality. Although one of the reasons for this was to is try to limit the interconnectivity overheads; however, this dramatically limits the architecture capabilities and flexibility and leads to a reduction in the range of applications that can be executed.

2.4.17 PipeRench Architecture

The PipeRench architecture use pipeline reconfiguration as its c main concept [26]. The architecture is composed of a set of pipeline stages, rows or stripes, as illustrated in Figure 2-33. Each stripe consists of processing elements up to N and

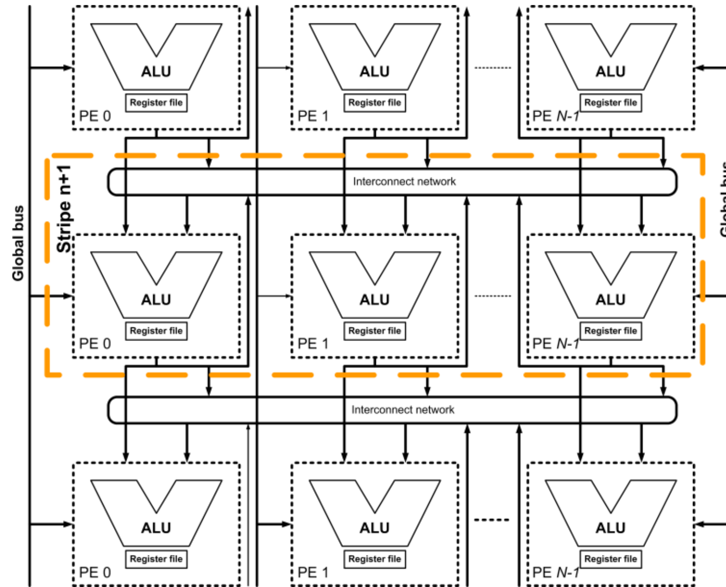

Figure 2-33 PipeRench architecture layout [26]

an interconnect network. Each processing element contains registers and ALUs. The ALU is based on a look-up-table, and. PEs can interact with each other within the same stripe but not adjacent ones.

This architecture has two types of interconnections: a local network where all PEs within the same stripe can share some data and have local transfer; and a global network where a PE in a stripe can read data from the output register of the above stripe. Each PE is 8-bits based while the whole stripe totals 128-bit in width. This means that there are 16 PEs per stripe.

The architecture's principle of operation in having two levels of configuration for the PE and the stripe is novel. However, it is too hardware-oriented and overly focused on the pipeline approach. This limits the applications that this architecture can handle. Moreover, it appears that the implementation of such an architecture will be costly in terms of area, power and performance.

2.4.18 rDPA Architecture

The processing elements in the reconfigurable datapath architecture (rDPA) are called datapath units (DPU) [27]-[29]. The rDPA is reconfigurable in-circuit and is scalable to large array sizes. The architecture has an added controller called the

reconfigurable ALU (rALU) which allows the architecture to be data-driven. The rALU provides the architecture with the capability for the parallel and pipeline computation of complex expressions.

The rDPA architecture consists of an array of reconfigurable processing elements or DPUs, as illustrated in Figure 2-34. The number of DPUs within the array can be up to 128. The elements are connected using a mesh type interconnect network. The architecture has two interconnection levels: global interconnection through longer lines and local interconnection through shorter lines.

As illustrated in Figure 2-34, the rALU consists of an rDPA control unit, an rDPA address generation unit and a register file. A data-driven reconfigurable ALU is the result of having rALU with rDPA within the architecture. The register file has 64 x 32-bit registers used for holding intermediate data in order to reduce the multiple reading of data from the memory. The execution of the whole architecture is data-driven, including the configuration process. The rDPA control unit holds the

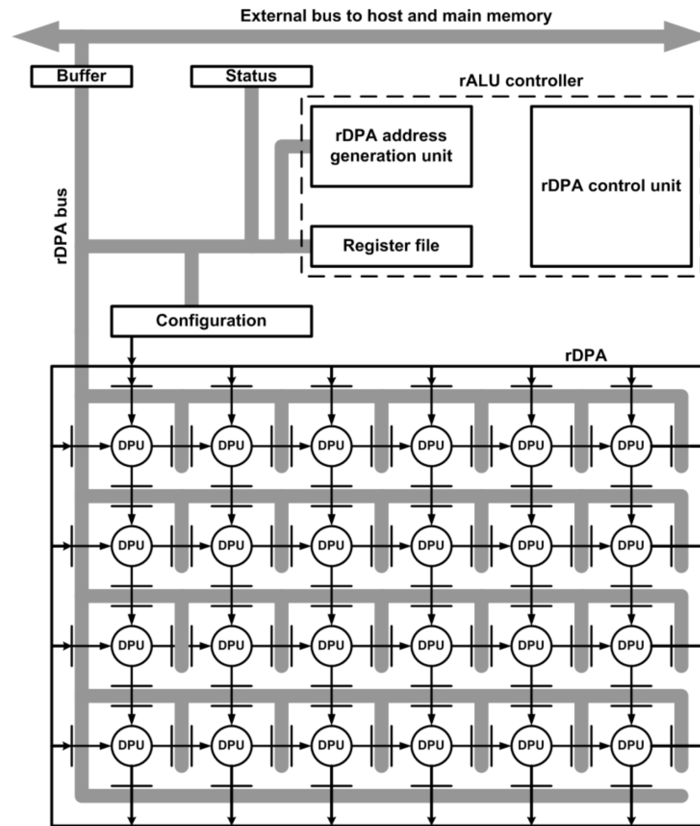

Figure 2-34 rDPA architecture with ALU controller [27]

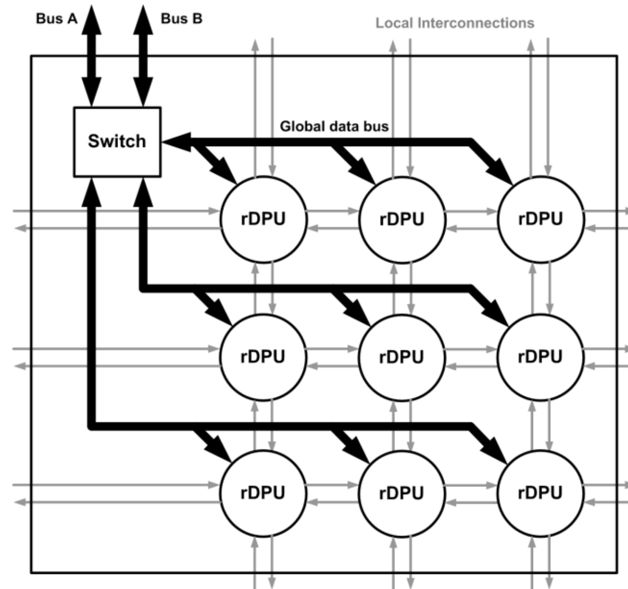

Figure 2-35 KressArray architecture [27]

instructions sets and delivers instructions to the designated DPU within the rDPA. The architecture is uses a single I/O bus to connect all datapath units using multiplexing; however, the designers hinted that the architecture would benefit from two buses to speed up I/O operations [27].

2.4.19 KressArray Architecture

The KressArray architecture or KressArray-III is a 32-bit-based coarse-grain architecture [27]-[29]. The processing elements of this architecture are called reconfigurable datapath units or rDPUs. Figure 2-35 represents the structure of the architecture's layout with its interconnection network. Local interconnections are used to feed data directly to the designated PE or rDPU or to read the resulting data. In addition, they can be used to pass intermediate results from one rDPU to another. The hierarchical interconnects allows this flexibility. The architecture has nine PEs and 32-bit duplex connections in four directions, north, east, west and south (NEWS). The direction of the dataflow through the connection is programmable.

An interesting feature in this architecture is its capability to allow the rDPU to be reconfigurable as a router. Usually architectures have dedicated switches to

reconfigure interconnections, and these switches act as routers. This is a compromise from the architectural design point of view, since having a complete rDPU acting like a router is a waste of valuable computation resources. However, the authors claimed that the rDPU can be split into a partial router (routing only limited number of connections; usually one) and a processing element. They suggest that, in partial mode, the processing capabilities remain intact and the PE can fully utilise them. Nevertheless; is unclear weather or not what has been proposed by the authors really is a full integration between the PE and the switch in the single unit called the rDPU. This may have implications for the area, power consumption and flexibility of the system, and could complicate the programming of the system as well.

Another interesting feature of this architecture is that the configuration memory consists of four independent layers. In addition, the register file within each rDPU has four configuration layers, as illustrated in Figure 2-36, to hold the four complete configuration sets. All rDPUs within the architecture simultaneously

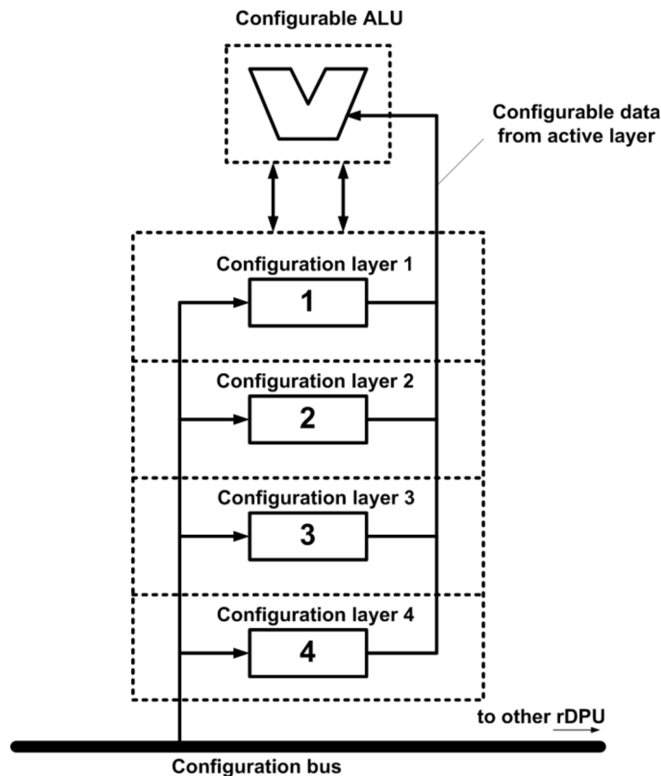

Figure 2-36 rDPU four configuration layers

change the actual configuration memory layer. The aim of these layers is to minimise or eliminate the reconfiguration time of the array, since there will be only one active configuration layer at any given time while the others are idle. The system can reprogram those layers, given that the configuration data and configuration control buses are independent. The elimination of reconfiguration time is a great step forward towards real-time reconfiguration.

A key drawback of having four layers is the need for four times the size of the configuration memory and registers, which will reflect negatively on the architecture area and, most importantly, power consumption. It is worth noting that the KressArray architecture is meant to be a co-processor or accelerator. Moreover, this architecture has a strong tool set that supports various optimisations for the algorithms implemented in seeking the best performance level.

2.4.20 MOVE Architecture

Any thorough review of reconfigurable architectures would usually consider some or all of those discussed above. However, transport-triggered architectures (TTAs) are usually missing from such studies. However, this is such an important type of architecture that it must be taken into consideration. Arguably, the TTA may or may not be considered to be a truly reconfigurable architecture; however, from the point of view of performance and flexibility it is significant and deserves to be considered among the reconfigurable architectures rather than as a general-purpose processor.

The transport-triggered programming paradigm was developed at the Delft University of Technology [30]-[32]. The paradigm was changed from ‘operation triggered’ to ‘transport triggered’, and the realisation of this paradigm is an architecture called the MOVE32INT. The key feature and main principle in this architecture is the reduction of the instruction set to only one operand, which is the ‘MOVE’ function; hence, the name of the architecture.

Figure 2-37 shows the architecture’s structure where the focus is the transport network and the processing elements or the functional units (FU) are distributed along the network. There are sockets that define the connection between FU and

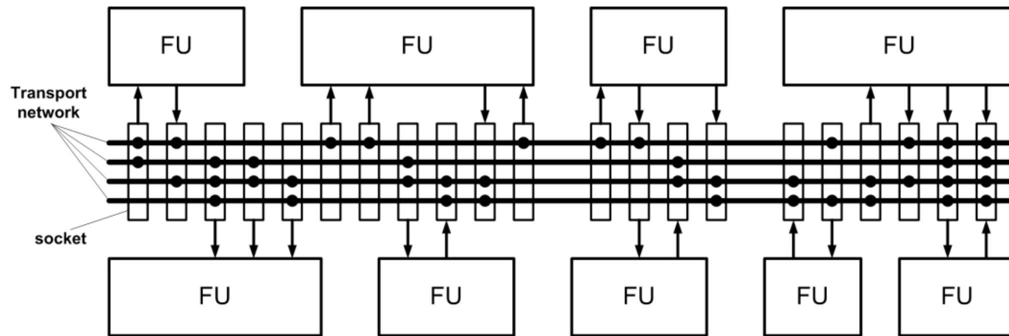

Figure 2-37 Move architecture structure [30]

the network which can be either input or output from the function unit. The functions of the FU can range from being a single operation unit (operand), to a complete ALU, and it can also accommodate internal pipelining. Each FU has a register at its output named the result register. An interesting aspect here is that the FUs in general are heterogeneous and designed to fit targeted areas of application or can be narrowed down to suit a specific algorithm.

It is worth mentioning that there is a clear separation in this architecture between operations and transport.

As can be seen from Figure 2-38, there are four types of registers within the architecture: ‘O’ is the operand register; ‘T’ is the trigger register, ‘R’ is the result register and finally ‘r’ is the general purpose register.

Operations within the FU will kick-start once the trigger register T is loaded. The cycle time of this architecture is determined by data transport.

The MOVE32INT is a 32-bit-based processor, which uses Harvard architecture with separate data and address paths to memory. The processor is capable of four concurrent data transports per clock.

It is clear that this architecture is conceptually interesting, allowing high enough flexibility for it to be a programmable processor. Despite all of its interesting features, however the architecture has a key drawback which is the complicated process of code compilation. When the MOVE was compared with VLIW in DSP applications [33], it was noted that MOVE has a much lower code density resulting in a larger code size. This is clearly one of the main drawbacks of the architecture.

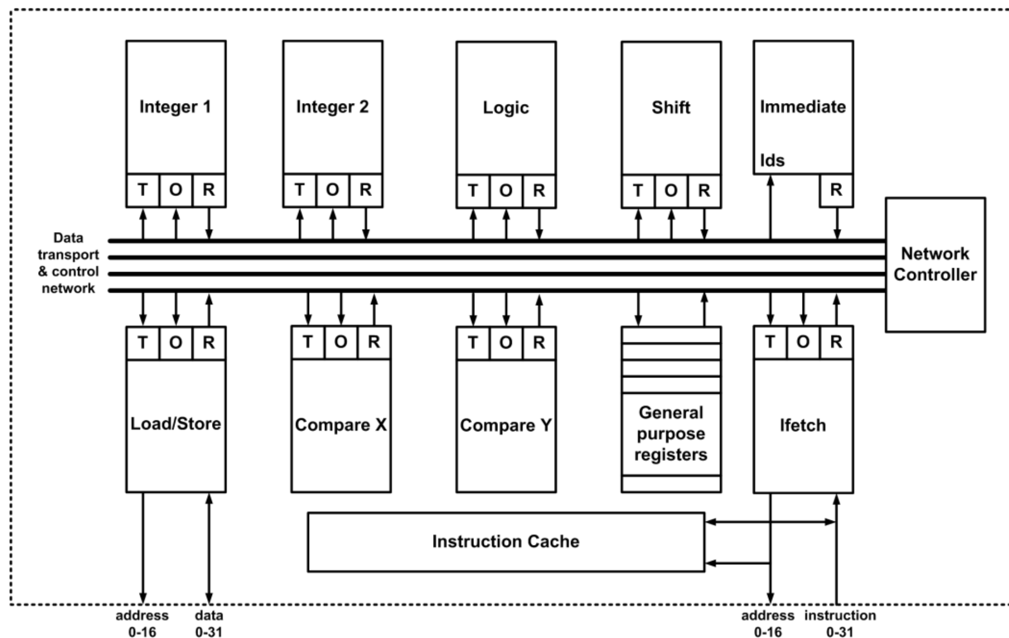

Figure 2-38 MOVE32INT block diagram [31]

Another upgraded processor has been developed to overcome some of these drawbacks, namely the MOVE-Pro [34]. This new processor is based on the TTA architectural concept; however, it is built with power savings as a major driver along with increasing code density.

Move-Pro promises significant dynamic power savings through the reduction of the number of accesses needed to the register file. Key changes from the earlier processor are the addition of jump and branch instructions to the instruction set. The authors concluded that the new processor achieves 80% savings in register file access and a total of 11% reduction in power consumption compared to the older version [34].

Despite the two processor versions, TTA has some clear advantages such as modularity, flexibility and scalability. TTA architectures may have a future as a reconfigurable architectures; however, this would require more time and effort from the developers. Instead of trying to compare it with RISC processor, it may be worth looking further into moving the TTA architecture into the reconfigurable architecture arena.

2.4.21 RICA Architecture

The reconfigurable instruction cell array (RICA) architecture was developed at the University of Edinburgh [35]. This architecture is based on having an array of customizable instruction cells or processing elements. A unique feature is that the architecture's processing elements or instruction cells are heterogeneous. Table 2-1 lists the different types of instruction cells.

Table 2-1 Main types of instruction cells types in the RICA architecture

Instruction Cell	Operations executed
ADD	Addition/subtraction
COMP	Compare two values
DIV	Signed/unsigned divisions
I/O REG	Register with access to I/O ports
JUMP	Branches/ step end
LOGIC	Logic operations (AND, OR, XOR, etc.)
MEM	Data memory READ/WRITE functions
MUL	Signed/unsigned multiplications
MUX	Multiplexer/simple branching
REG	Register
RRC	Reconfiguration rate controller
SHIFT	Shift logic operation

The main concept behind the RICA is the processor, which is able to handle the control and dataflow aspects of applications. This handling is flexible, maximises utilisation, and supports parallel processing and low-power consumption. Thus, the RICA architecture is characterised by a high performance high parallelism and has a processor with low power consumption and small area. It is highly flexible being coarse-grain and scalable which allows it to be adapted to the application required by using the most suitable combination of types and numbers of instruction cells.

From a programming point of view, this processor has the key advantage of DSP processors of being programmable using the ANSI-C programming language and its tool flow can be designed using GNU C-compilers, which is familiar to programmers. Therefore the architecture will not need a skilled hardware engineer with experience in HDL languages, but on the contrary C-programmers will be able

to efficiently use and program this processor, thus saving time and resources. When the algorithm program is compiled, the resulting assembly code is then sliced into blocks of instructions, and each block is executed in a single step. The step size is defined by the resources available within the architecture and the number of read/write operations included. Therefore, the algorithm will be executed in steps, which allows the architecture to be dynamically adaptable to each step. Each step can have a different critical path, where the clock that controls the program counter and memory is reconfigurable. The programmable clock allows the architecture to provide optimum performance, by providing the maximum performance level for each step, so that maximum performance or throughput for the whole application is guaranteed. The JUMP instruction is used as a trigger for the architecture to load the next configuration or step. If the step contains a full loop of instructions, this means that the processor will avoid any reconfigurations. Only the data will be loaded from the memory or registers, giving the processor a great advantage over other architectures. Figure 2-39 illustrates how a single C-code instruction is compiled and mapped on the architecture cells. This allocation or mapping changes with every code or set of codes.

Various applications have been implemented on the RICA and customised versions show high performance with significantly lower power consumption [36]-[39]. There are two features which the RICA and TTA architectures have in common, discussed earlier in section 2.4.20. Firstly, both are architectures based on heterogeneous processing elements. The RICA has heterogeneous instruction cells while the TTA architecture has heterogeneous functional units. Secondly, both are claimed to be processors or processor-like architectures with Harvard-like structures. In other words, both can run independently without the need for an external processor to fetch instructions or synchronise operations. However, the drawbacks of the TTA discussed earlier, including its complex register file structure, do not apply to the RICA. Moreover, the RICA is designed from the ground up based on two key features, which are low power consumption and an architecture which is easy to program.

C code

```
e = b1 * 6 - b7 * 27;
```

Compiled ASM:

```
LDR     a1,[sp,#0x24]
LDR     ip,[sp,0x3c]
MOV     a3,#6
MOV     a4,#0x1b
MUL     ip,a4,ip
MUL     a1,a3,a1
SUB     a1,a1,ip
```

Figure 2-39 Dynamic allocation of instruction cells into processing steps, scheduled within the GCC tool chain

2.4.22 CDDS Variable Datapath Architecture

A control-flow driven data-flow switching (CDDS) variable data architecture has been introduced at by Hokkaido University, which is characterised by flexibility and low energy consumption as cited by the authors [40]. The authors achieved the balance between performance and power consumption was achieved by limitating the scope of dynamic reconfiguration. This architecture is aimed at control-intensive programs, and its datapath is divided into static and dynamic sections. Only the dynamic part is allowed to be dynamically reconfigurable at run time. Figure 2-40 illustrate the split in dynamic reconfiguration the architecture datapath. The architecture is designed to work mainly as an accelerator beside the main processor.

The PEs in this architecture are ALU which are asymmetric to those of the main processor in the whole system, in order to streamline the architecture's programming and mapping.

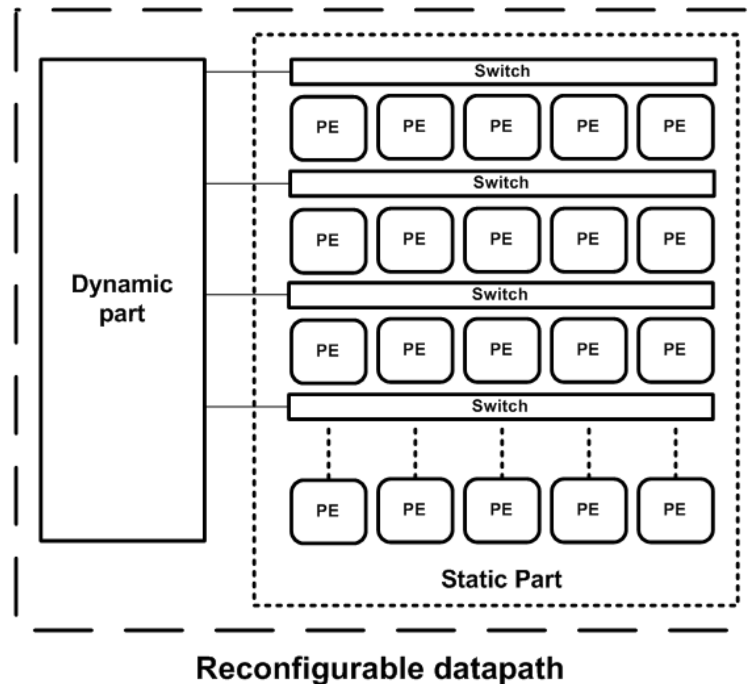

Figure 2-40 CDDS architecture's reconfigurable datapath separated into static and

This architecture is similar to RICA in that the key parameter for switching the dynamic reconfiguration is the branch, which in this case matches the RICA jump instruction.

The architecture has a clear novelty in restricting dynamic reconfiguration and memory access during execution for the static portions or PEs, while this is allowed for switching. This approach may have clear advantages in terms of power consumption; however, it has limitations in terms of being suitable for a restricted range of applications and needing a larger area in order to cater for the multiple branch options on the array so as to proceed with reconfiguration. This architecture is still in its early days, and may evolve when significant algorithms are implemented with it.

2.4.23 BilRC Architecture

The PEs of a execution-triggered coarse-grain reconfigurable architecture entitled BilRC [41] are inspired by the FPGA. The PEs in BilRC are of three types: ALU, memory and multiplier. The PEs are realised in columns of the same type as illustrated in Figure 2-41.

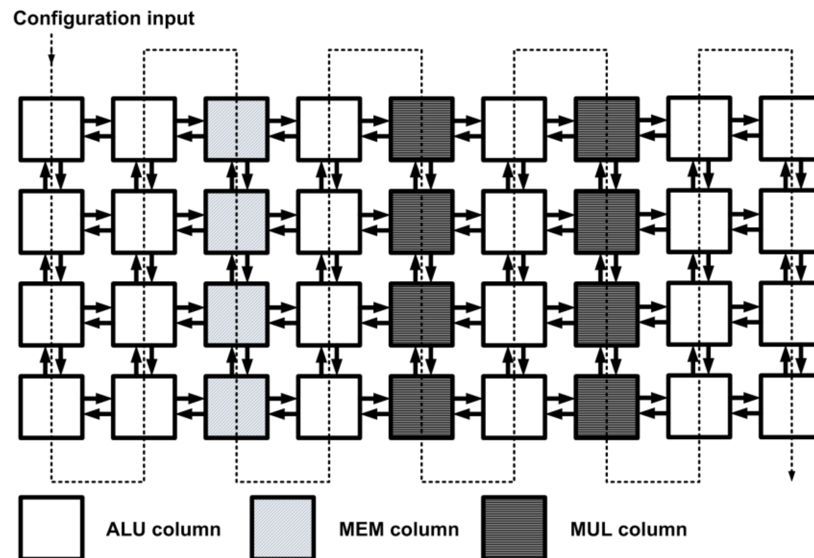

Figure 2-41 PE column based structure in BilRC [41]

The applications this architecture is intended for span in a wide range, from signal processing to telecommunications. However it is clear that PEs the distribution of PEs will change depending on the application domain targeted. An interesting aspect of this architecture is that the authors used the MUX instruction for transportation within the architecture, which is similar to the MOVE instruction in the MOVE32int architecture.

Key points of this architecture are that it is static and not dynamically reconfigured; moreover, it is programmed through a special language developed by the authors called LRC. Despite the high reported performance compared to FPGAs and dedicated DSPs, no comparison of power consumption was mentioned. This may be expected to be addressed in future publications.

2.5 Comparison and discussion

Reconfigurable architectures can be classified based on various criteria, including datapath width, type of PE interconnection, reconfiguration model, programming language, placement and routing.

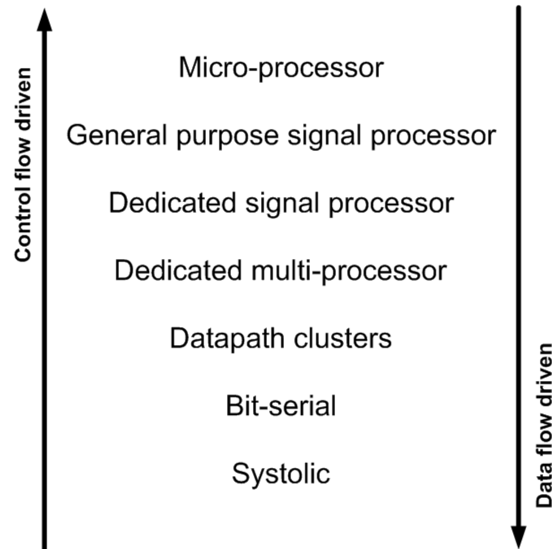

Figure 2-42 Classification based on control/arithmetic ratio [42]

One classification can be based on the control/arithmetic ratio. Brodersen and et.al. [42] categorised architectures based on the amount of sharing operations on an arithmetic unit. As seen in Figure 2-42, all operations in the micro-processor of the general purpose ALU are all operation time-multiplexed; hence, it is a control-driven architecture. On the other hand, a systolic array, for example, has each operation represented by separate hardware with minimal control. The best outcome would be the right balance between control and datapath for a given application and throughput.

For a reconfigurable architecture to be capable of carrying out the tasks of telecommunications system efficiently, various criteria would need to be satisfied. Datapath width or granularity is one of the key features of such architectures, and should be between 8 to 32 bits. This range would be suitable for telecommunications applications today and in the near future. It is anticipated that 64 bits would be desirable for future systems. Furthermore, it is desirable for reconfigurable architecture to have heterogeneous rather than homogenous PEs, for two reasons. Firstly, this will allow enough flexibility to accommodate the challenging functions of telecommunications systems. Secondly, it will most probably involve lower power consumption due to the high utilisation of the

resources in the architecture. Another key aspect to consider is the ability of the reconfigurable architecture to work as a standalone unit and not just as a co-processor or an extra functional unit for a main processor. This is the key for telecommunications systems, from the point of view of efficiency, optimisation and power consumption.

Various efforts have been made to compare reconfigurable architectures and in particular the coarse-grain architectures [43]. The present study focuses on reconfigurable architectures suitable for telecommunications systems.

A comprehensive comparison of the various architecture is provided in Table 2-2. Here the approach used with each processor can be clearly identified in terms of datapath width, and level of supported reconfiguration in terms of whether it

Table 2-2 Comparison of studies of reconfigurable architectures

Architecture	PEs Homogeneous/ Heterogeneous	Datapath Width	Reconfiguration. Dynamic/Static	Application	Role
BilRC	Heterogeneous	16-bit	Static	General	Standalone
CDDS	Homogeneous	32-bit	Dynamic	DSP	Coprocessor
CHESS	Homogeneous	4-bit	Static	Multimedia.	Coprocessor
Chimaera	Homogeneous	16-bit	Static	DSP	Coprocessor
CRISP	Heterogeneous	8-bit	Static	Multimedia.	Coprocessor
DART	Homogeneous	8-bit	Dynamic	Telecomm.	Coprocessor
DREAM	Homogeneous	8-bit	Dynamic	Telecomm.	Coprocessor
Garp	Homogeneous	2-bit	Static	DSP	Coprocessor
KressArray	Homogeneous	32-bit	Static	General	Coprocessor
MATRIX	Homogeneous	8-bit	Dynamic	General	Coprocessor
Morphosys	Homogeneous	8-bit	Dynamic	General	Coprocessor
MOVE	Heterogeneous	32-bit	Static	DSP	Standalone
OneChip	Heterogeneous	32-bit	Static	DSP	Coprocessor
PADDI	Homogeneous	16-bit	Dynamic	DSP	Coprocessor
PipeRench	Homogeneous	8-bit	Static	DSP	Coprocessor
Pleiades	Homogeneous	8-bit	Static	Telecomm.	Coprocessor
RaPiD	Homogeneous	16-bit	Static	DSP	Coprocessor
rDPA	Homogeneous	32-bit	Dynamic	Telecom	Standalone
REMARC	Homogeneous	16-bit	Static	DSP	Coprocessor
RICA	Heterogeneous	32-bit	Dynamic	DSP	Standalone
SRGA	Homogeneous	2-bit	Dynamic	General	Coprocessor
Systolic Ring	Homogeneous	8-bit	Static	General	Coprocessor
vCell Matrix	Homogeneous	4-bit	Static	General	Coprocessor

supports dynamic reconfiguration or is limited to static reconfiguration. In addition, it should be noted whether the architecture is capable of being standalone or if it requires an external processor. Moreover, key differentiation among PEs in an architecture is whether they are homogeneous or heterogeneous. The architectures considered were built in targeting specific applications, which are also indicated in table 2-2.

Most of the reconfigurable architectures are designed to act as co-processors, except for the RICA, MOVE, BilRC and rDPA. Another important criterion is the nature of the reconfigurable cells or PEs. Most of the architectures are based on homogeneous PEs, except for the RICA, CRISP, Pleiades, OneChip, MOVE and BilRC architectures which have heterogeneous PEs. Dynamic configuration is another key feature that is most desirable for telecommunications systems, as highlighted earlier. Several applications are highlighted in table 2-2 which support dynamic reconfiguration, either partially or fully.

2.6 Conclusion

A reconfigurable architecture which can meet the challenging requirement of communications systems has to have many crucial characteristics. Primarily it has to operate with low power consumption. In order to sustain this, the use of heterogeneous PEs is the best approach. PEs can be customised specifically to the system's needs, resulting in the highest utilisation, which will lead to lower power consumption and smaller area.

It appears from Table 2-2 that the most suitable architectures for telecommunications systems are the MOVE, BilRC and RICA. BilRC is disregarded here since it uses a new nonstandard programming language.

The MOVE and RICA are very different architectures; however, they are similar in that the PEs used are being heterogeneous, both are standalone and do not need external processors for control, and are programmable using C-language.

However, the RICA appears to be superior in terms of power consumption, the processor has been built with low power use as its core principle. The MOVE

designers only started to address power savings at a later stage, whereupon they deviated from having MOVE as the only instruction and added the two additional instructions - JUMP and BRANCH. In addition, the RICA is fully dynamically reconfigurable, which is a key feature lacking in MOVE. Hence, the RICA is the architecture chosen as the paradigm upon which the reconfigurable architectures in this work is built.

Chapter 3

MULTIRATE CONVOLUTION ENCODER

3.1 Introduction

The integration of a number of digital devices into a single device is of great interest in the present decade. In order to achieve this, a design of miniaturised devices with reduced power consumption accompanied with a degree of flexibility is the key to success. The recent IEEE 802.16 (also known as WiMAX) standard promises ultra long communication ranges over kilometers for wireless systems [44]. The communication distance supported by WiMAX is suitable for sensor node and cluster head communication [46]-[47] in the ESPACENET project, which is developing evolvable networks of intelligent and secure integrated and distributed reconfigurable system-on-chip sensor nodes for aerospacebased monitoring and diagnostics. The ESPACENET project involves the development of pico-size satellites (spacecraft) so that a networked group of them could functionally replace a micro-sized to large satellite. In order to achieve this, each node (i.e. each single pico-satellite) should support several communication standards. It has been shown that the most suitable standards are the IEEE 802.11 (WiFi) for short ranges and 802.16 (WiMAX) for longer ranges between the various nodes or cluster heads. Longer intermediate distance ranges are common in space applications. In a network, there are two levels. The nodes are the lowest level and the cluster heads manage groups of nodes. The pico-node is a platform characterised by limited area and power use. In order to integrate the WiMAX standard and all of its options, the design should prioritise minimal power consumption and size. In space applications, power and weight (size) need careful consideration in order to keep them to the minimum. Moreover, the design has to

accommodate all the standard modes, which need to be controlled by a node controller or processor. In this chapter two modules of the physical layer of a WiMAX transmitter are considered, which are the convolution encoder and puncturing configuration. These modules have been chosen due to their presence in almost all digital mobile radios, for example WiFi (802.11). These modules are used here to investigate the addition of reconfigurability and flexibility and the overheads associated with this.

In research as well as commercial products for wireless communication systems, the architecture used for the implementation of the convolution encoder and the puncturing configuration is based on their co-location as subsequent units. The fixed convolution encoder with a rate of 1/2 is followed by the puncturing configuration for other higher rates which are supported, shown in Figure 3-1. Generally, the design and/or implementation is based on a separate subsequent units [52]-[55]. Another study presents the programmable convolution encoder as based on the same architecture as shown in Figure 3-1, but the convolution encoder could provide a rate of 1/3 in addition to 1/2 [53].

In this chapter, a novel architecture is introduced which provides full integration of the convolution encoder and the puncturing configuration into a single IP (intellectual property) or module. As mentioned earlier, the WiMAX standard is used because it sets the most stringent requirements and is the most challenging of current standards. The challenge here is to identify the best architecture for reconfigurable integration. In the present research, the convolution encoder constraint length (k) is fixed at seven in order to allow the architecture to be flexible for various rates, while the targeted architecture could be used for other constraint lengths. The constraint length k of a convolutional encoder is the

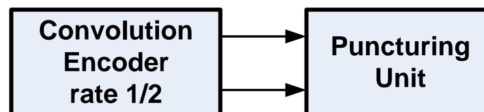

Figure 3-1 Convolution encoder and puncturing configuration as separate units in the transmitter of a wireless communication system

maximum number of symbols in a single output stream that can be affected by any input symbol, which reflects the number of memory units used [48]. Furthermore, k=7 is the most commonly used length, especially for receiver decoders in wireless systems. This is mainly because the decoder for the convolution is usually a Viterbi decoder, and constraint length of seven turns out to be the most suitable in terms of memory usage, efficiency and success in recovering transmitted information. For example, in one study [56] convolution encoder and Viterbi decoder were implemented on an Altera FPGA for the rate ½ and k=7 the authors implemented. The other constraint lengths are three, five and nine, which are not commonly found in realisations. The novel concepts presented here for the length constraint of seven will also be applicable to other constraint lengths, since the solution proposed is scalable. The flexibility arises here from having the convolution encoder being able to execute multiple rates instead of being fixed to a specific rate.

In the next section, convolution and puncturing are briefly explained. Section 3.3 shows the technique implemented in this study to achieve concatenation between both units. Section 3.5 explains the novel architecture, which results from integrating various combinations of all supported rates. Finally, the design and implementation results and the conclusions are presented.

3.2 Convolution Encoder and Puncturing Configuration

3.2.1 Convolution Encoder

A convolution coder is an error-correcting coder that processes information serially and continuously. The output symbols of a convolution encoder depend on the inputs as well as the previous inputs or outputs, which means that memory is required in which to save some of the history [48]. Convolution decoding is usually carried out using the Viterbi algorithm.

The convolution coder encodes each m-bit information symbol into an n-bit symbol, where m/n is the code rate ($n \geq m$) and this transformation is a function of

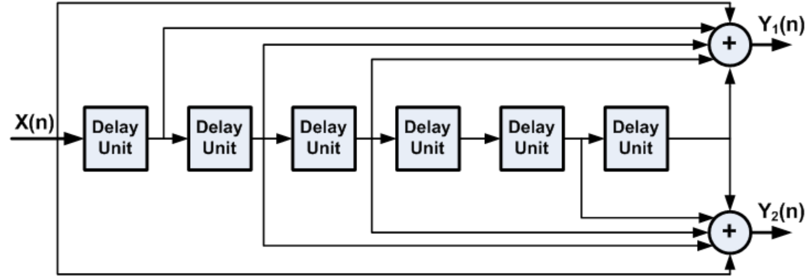

Figure 3-2. Convolution encoder (rate=1/2, *k*=7)

the last *k* information symbols, where *k* is the constraint length of the code or the memory required depth.

The binary convolution encoder for the WiMAX standard [44] is derived from a rate of 1/2 and constraint length *k*=7 [39]. The encoder uses the generator polynomials in Equations (3-1) and (3-2) to obtain the coded output. The generator polynomials are usually provided by the standard, and in this case WiMAX equations are used. The encoder is designed using delay units (registers) as shown in Figure 3-2. The number of delay units used is equal to the constraint length -1.

$$G_1 = 171_{OCTANT} \qquad \text{For } Y_1 \qquad (3\text{-}1)$$

$$G_2 = 133_{OCTANT} \qquad \text{For } Y_2 \qquad (3\text{-}2)$$

The encoder input is a continuous bit stream represented by X and the output by Y; and as the rate used is 1/2, then there will be two outputs Y_1 and Y_2, with a two-modulo2 adder (exclusive-OR) as shown in Figure 3-2.

Table 3-1 WiMAX puncture pattern configuration and resulting convolution code serialization

	Puncturing codes and patterns		
Rate	Y_1	Y_2	Y_1Y_2
1/2	1	1	$Y_1^1Y_2^1$
2/3	10	11	$Y_1^1Y_2^1Y_2^2$
3/4	101	110	$Y_1^1Y_2^1Y_2^2Y_1^3$
5/6	10101	11010	$Y_1^1Y_2^1Y_2^2Y_1^3Y_2^4Y_1^5$
7/8	1000101	1111010	$Y_1^1Y_2^1Y_2^2Y_2^3Y_2^4Y_1^5Y_2^6Y_1^7$

3.2.2 Puncturing Configuration

Puncturing is the process of systematically deleting, or not sending, some of the output bits of a low-rate encoder [48]. As stated in the WiMAX standard, it is required that the puncturing configuration adjusts the rate of 1/2 to higher rates by omitting some bits, as illustrated in Table 3-1 (where 1 indicates bit to be transmitted, while 0 indicates skipping). Y_1Y_2 is the resulting serial output with the assigned rates. The superscript numbers represent their sequence in time; for example, 1 indicates the first bit to exit the unit.

3.3 Technique for the Parallelization Punctured Convolution Encoder

Parallelization is usually a way of speeding up processing, increasing frequency rate or throughput. Benjamin [49] introduced a method of basically converting the convolution encoder with a normal 1/2 rate followed by a puncturing for the desired rate into a single parallel punctured convolution encoder in parallel form. The idea presented by Benjamin with the example of a rate of 3/4 is shown in Figure 3-3.

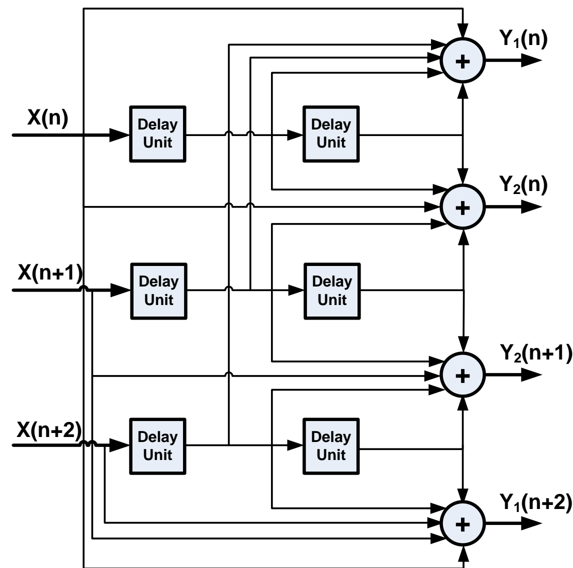

Figure 3-3. Parallel punctured convolution encoder [49]

In Figure 3-3 *X(*n*)*, *X(*n+*1)* and *X(*n+*2)* are the input bits in a parallel form, which need to be coded by the convolution encoder and then punctured to fit the rate of 3/4. *X(n)* is the first bit to enter, followed by *X(*n+*1)*, then *X(*n+*2)*. For the above unit to work properly all the three inputs must enter simultaneously, and thus a serial to parallel converter is required to provide those inputs. The output polynomial expressions are derived as:

$$Y_1(n) = X(n) \oplus X(n-1) \oplus X(n-2) \oplus X(n-3) \oplus X(n-6) \quad (3\text{-}3)$$

$$Y_2(n) = X(n) \oplus X(n-2) \oplus X(n-3) \oplus X(n-5) \oplus X(n-6) \quad (3\text{-}4)$$

$$Y_2(n+1) = X(n+1) \oplus X(n-1) \oplus X(n-2) \oplus X(n-4) \oplus X(n-5) \quad (3\text{-}5)$$

$$Y_1(n+2) = X(n+2) \oplus X(n+1) \oplus X(n) \oplus X(n-1) \oplus X(n-4) \quad (3\text{-}6)$$

Where n = 0, 3, 6 ... etc

Based on the above principle, diagrams and polynomial expressions for the different rates have been derived and are presented below. For the rate of 1/2 in Figure 3-2, the resulting polynomial expressions are:

$$Y_1(n) = X(n) \oplus X(n-1) \oplus X(n-2) \oplus X(n-3) \oplus X(n-6) \quad (3\text{-}7)$$

$$Y_2(n) = X(n) \oplus X(n-2) \oplus X(n-3) \oplus X(n-5) \oplus X(n-6) \quad (3\text{-}8)$$

Where n = 0, 1, 2 ... etc.

Meanwhile for a rate 2/3, the extracted design is shown in Figure 3-4, and the resulting polynomial expressions are:

$$Y_1(n) = X(n) \oplus X(n-1) \oplus X(n-2) \oplus X(n-3) \oplus X(n-6) \quad (3\text{-}9)$$

$$Y_2(n) = X(n) \oplus X(n-2) \oplus X(n-3) \oplus X(n-5) \oplus X(n-6) \quad (3\text{-}10)$$

$$Y_2(n+1) = X(n+1) \oplus X(n-1) \oplus X(n-2) \oplus X(n-4) \oplus X(n-5) \quad (3\text{-}11)$$

Where n = 0, 2, 4 ... etc.

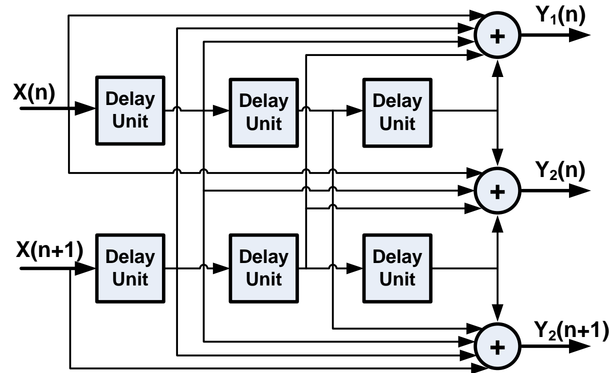

Figure 3-4. Punctured convolution encoder for the rate 2/3

Using the same procedure, polynomial expressions and designs for rates of 5/6 and 7/8 can be derived as demonstrated in Figure 3-5 and Figure 3-6 respectively.

For a rate of 5/6, the resulting polynomials are:

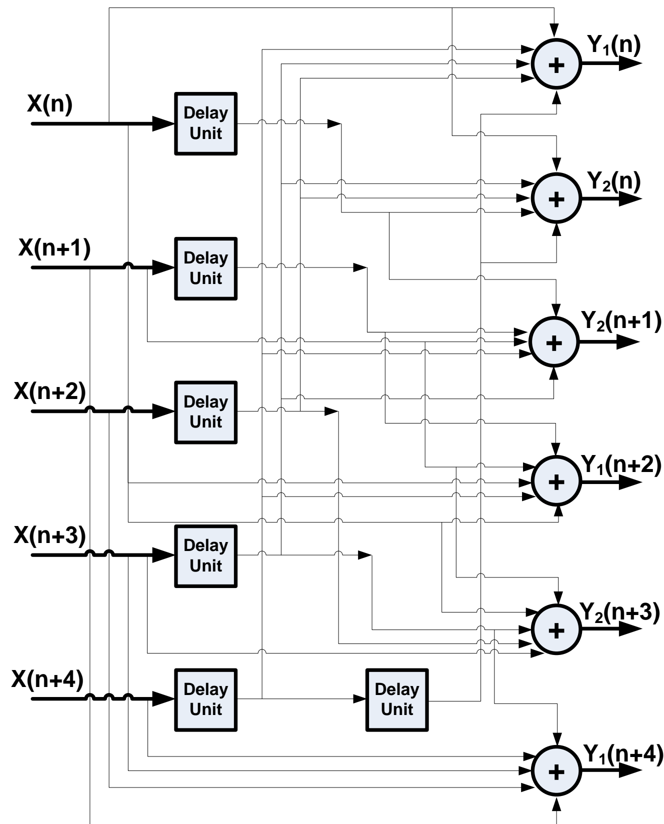

Figure 3-5. Punctured convolution encoder for the rate 5/6

$$Y_1(n) = X(n) \oplus X(n-1) + X(n-2) + X(n-3) + X(n-6) \quad (3\text{-}12)$$

$$Y_2(n) = X(n) \oplus X(n-2) + X(n-3) + X(n-5) + X(n-6) \quad (3\text{-}13)$$

$$Y_2(n+1) = X(n+1) \oplus X(n-1) + X(n-2) + X(n-4) + X(n-5) \quad (3\text{-}14)$$

$$Y_1(n+2) = X(n+2) \oplus X(n+1) \oplus X(n) \oplus X(n-1) \oplus X(n-4) \quad (3\text{-}15)$$

$$Y_2(n+3) = X(n+3) \oplus X(n+1) \oplus X(n) \oplus X(n-2) \oplus X(n-3) \quad (3\text{-}16)$$

$$Y_1(n+4) = X(n+4) \oplus X(n+3) \oplus X(n+2) \oplus X(n+1) \oplus X(n-2) \quad (3\text{-}17)$$

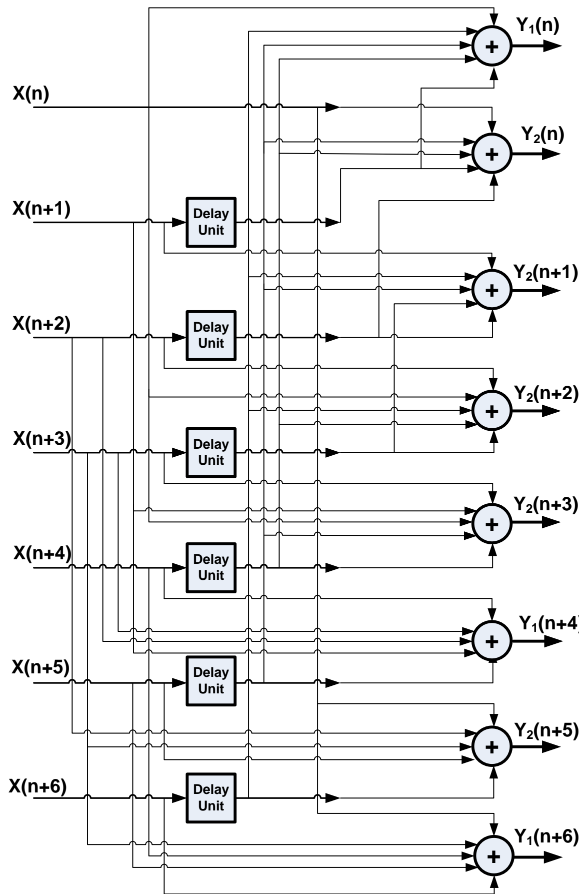

Figure 3-6. Punctured convolution encoder for rate 7/8

$n = 0, 5, 10$ ... etc.

Polynomial expressions for the rate of 7/8 are:

$$Y_1(n) = X(n) \oplus X(n-1) \oplus X(n-2) \oplus X(n-3) \oplus X(n-6) \quad (3\text{-}18)$$

$$Y_2(n) = X(n) \oplus X(n-2) \oplus X(n-3) \oplus X(n-5) \oplus X(n-6) \quad (3\text{-}19)$$

$$Y_2(n+1) = X(n+1) \oplus X(n-1) \oplus X(n-2) \oplus X(n-4) \oplus X(n-5) \quad (3\text{-}20)$$

$$Y_2(n+2) = X(n+2) \oplus X(n) \oplus X(n-1) \oplus X(n-3) \oplus X(n-4) \quad (3\text{-}21)$$

$$Y_2(n+3) = X(n+3) \oplus X(n+1) \oplus X(n) \oplus X(n-2) \oplus X(n-3) \quad (3\text{-}22)$$

$$Y_1(n+4) = X(n+4) \oplus X(n+3) \oplus X(n+2) \oplus X(n+1) \oplus X(n-2) \quad (3\text{-}23)$$

$$Y_2(n+5) = X(n+5) \oplus X(n+3) \oplus X(n+2) \oplus X(n) \oplus X(n-1) \quad (3\text{-}24)$$

Where n = 0, 6, 12 ... etc.

3.4 Reconfigurable concatenated convolution-puncturing architecture

According to WiMAX standard, support is required for all rates mentioned earlier in Table 3-1. In general, two possible architectures can be used to design the system supporting all these rates.

The first architecture is based on using a convolution encoder rate of 1/2 followed by a puncturing unit as shown in Figure 3-7. Each of the rest of the rates should have its own puncturing unit. Here each puncturing unit is considered as a separate IP core or module [50]. It is noted that, in this architecture, each puncturing unit needs its own parallel to dual converter in order to have the ability to maintain the ongoing data stream.

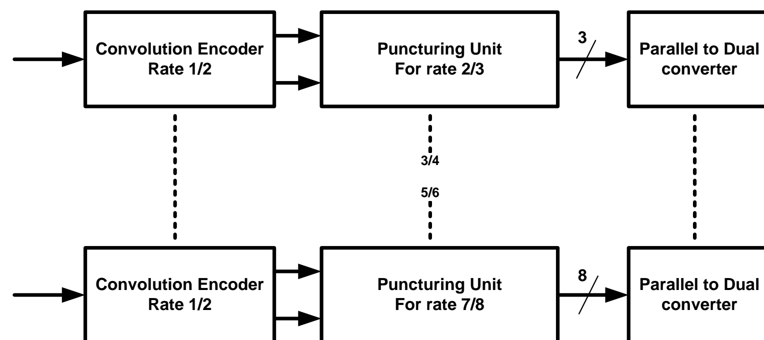

Figure 3-7. First conventional approach for implementing punctured convolutional encoder for different rates

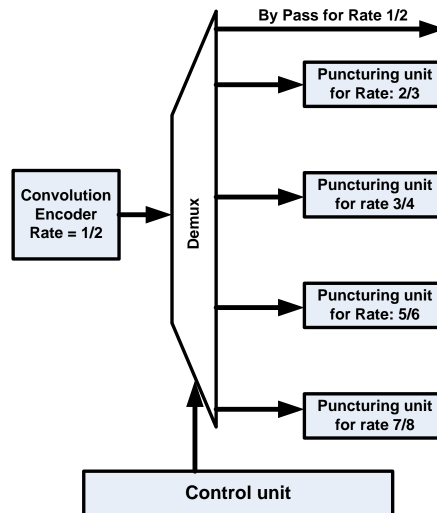

Figure 3-8. Second conventional approach for implementing punctured convolution encoder for different rates

The second possible architecture is an optimised version of the first. Now, instead of having separate parallel paths, it uses a single convolution encoder followed by a de-multiplexer to switch to the desired rate, as illustrated in Figure 3-8.

Both architectures have a timing issue, since time management and synchronisation will be sophisticated due to the different time spans required by each puncturing unit and the different data sizes that they handle. This leads to the need for the inputs to be buffered. A buffer is required and in this case the introduction of a FIFO (first-in first-out memory) or a memory buffer may be useful. The FIFO requires two different clocks for writing and reading; but this will increase the design complexity, area and power consumption.

In order to overcome the drawbacks of classical optimisations and to achieve a multi-rate convolution encoder, a novel architecture is introduced here which is far more optimised and fully integrated compared to the architectures mentioned earlier.

The novel proposed architecture is based on employing the parallel convolution with puncturing mentioned earlier in section 3.3 with the full integration of all the rates. The proposed architecture presented in Figure 3-9 comprises four units: the reconfigurable convolution unit, a reconfigurable serial to parallel unit, a reconfigurable parallel to dual channel unit (whose output is two parallel bits) and a

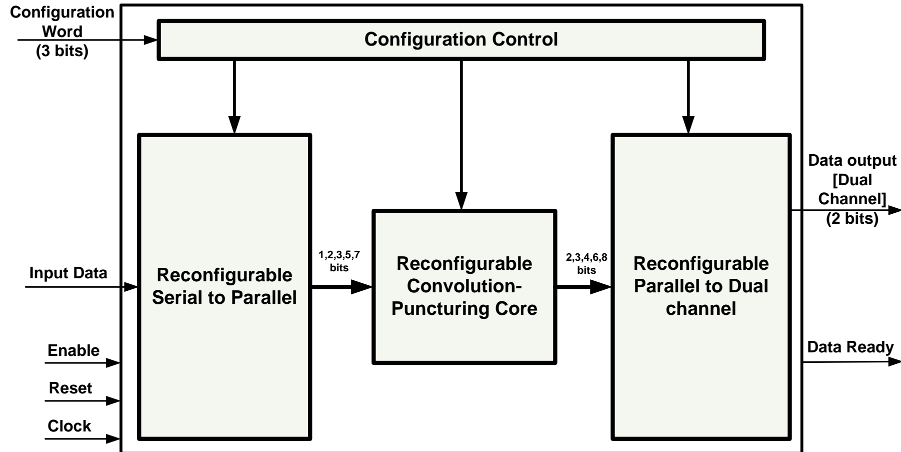

Figure 3-9 Proposed top-level architecture for low power reconfigurable concatenated convolution-puncturing module for 802.16

configuration controller unit. This architecture provides punctured convolutionally encoded data for all the rates in a continuous manner; in addition, it avoids the need for dual clocks.

The architecture has been designed and verified by simulating all possible rates and verifying the resulting outputs.

In this architecture, all rates have been integrated, including the original convolution encoder rate of 1/2. All the rates are based on six delay units only

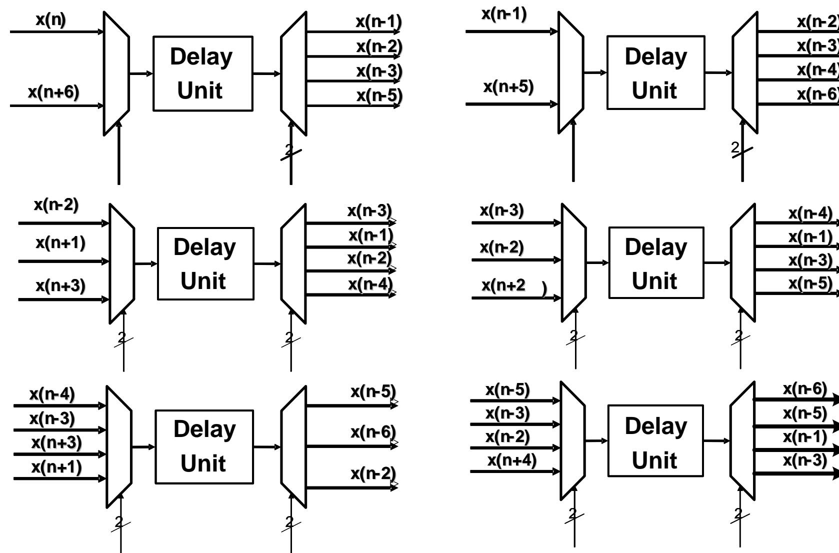

Figure 3-10 Reconfigurable interconnections for the convolution-puncturing core (delay units are implemented as registers)

(with the number of registers, k = 7), while the number of modulo-2 additions (exclusive-OR) is the same as the number of output bits for each rate. This new convolution puncturing architecture uses only six registers along with reconfigurable interconnections, while existing architectures require at least 30 registers.

The interconnection network between the registers and modulo two adders is to be configured using multiplexers and de-multiplexers, as illustrated in Figure 3-10. The reconfigurable core shown in Figure 3-10 represents the reconfigurable interconnection network between the registers (for clarity, the modulo two adders and their connections are not shown). The configuration controller provides the necessary control signals.

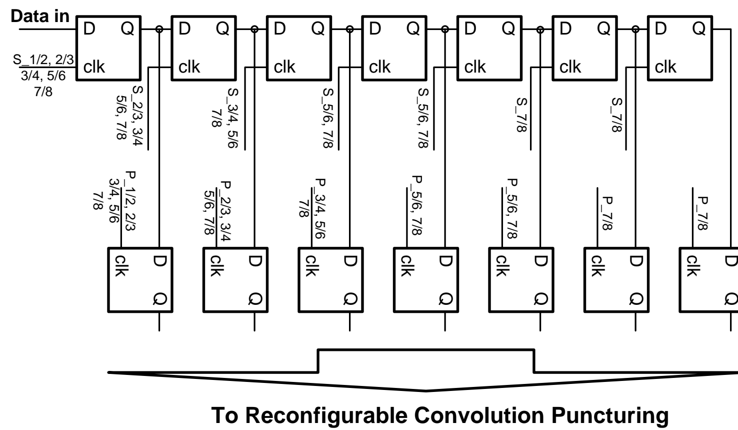

Figure 3-11: Reconfigurable serial to parallel

As shown in Figure 3-9, the input to the configuration controller is a 3-bit configuration word that configures the whole architecture based on the selected rate.

Table 3-2 shows the configuration words used in the architecture and their associated rates. In addition, the table lists the 19-bits internal configuration words that result from an embedded decoder in the configuration control unit. Configuration words provide the required signals for all units in the architecture in order to configure or program the interconnections for each selected rate. The

Table 3-2: External and internal configuration words

Rate	Configuration words	
	External	Internal (using decoder)
1/2	000	000_0000_0000_0000_0000
2/3	001	001_0101_0100_0011_0101
3/4	010	010_0110_1001_0111_1001
5/6	100	011_0111_1011_1010_1110
7/8	110	100_1111_1010_1111_1111

system can be reconfigured or reprogrammed in real-time to switch between the rates.

The reconfigurable serial to parallel converter (RSPC) unit converts the serial input stream into parallel output in order to feed the convolution puncturing encoder. The width of output parallel bits varies according to the selected rate and is assigned by the configuration word. The width varies from two parallel bits in the case of a rate of 2/3 to 7 bits for a rate of 7/8. It is worth noting that, for rate 1/2, this unit is bypassed and the incoming bit stream enters directly to the convolution puncturing encoder unit.

The RSPC is based on a shift register and parallel registers as per Figure 3-11. The enabling of these registers depends on the assigned rate. All enabling signals are controlled by an embedded controller (not shown in Figure 3-11). It decides when to enable each register to support the continuous operability of the whole unit and avoiding the usage of additional memory or the need for data holding.

The reconfigurable parallel to dual channel (RPDC) unit's output provides the inverse function for serial to parallel. The operation of this unit is specified by the selected rate and the control signals. For a rate of 1/2, the controller will adjust the output from the core to bypass this unit.

3.5 Results

The reconfigurable architecture design achieved for the multi-rate convolution encoder is the result of several design iterations. The target was to accommodate all required rates and to verify the correctness of all data. The final architecture iteration is the one discussed in section 3.4. The architecture has been simulated

Table 3-3. Area and estimated power for all supported rates

Rate	1/2	2/3	3/4	5/6	7/8
Power (μW)	157	185	407	327	302.9
Area (mm²)			0.012		

and synthesised using UMC 180 nm CMOS technology. The design proves its workability for different data rates, as shown by the results obtained which are listed in Table 3-3

The resulting area and estimated power consumption is a good indication of the capability of the designed architecture. The results have been obtained after applying power reduction techniques, such as the gated clock technique, to minimise the switching activity [51]. As expected, the rate of 1/2 provides the lowest power consumption and this is clearly due to the automatic disabling of the reconfigurable serial to parallel and parallel to dual modules.

Table 3-4 presents the resulting throughputs obtained by the architecture. The first row shows the results for the core unit only (the concatenated convolution puncturing), while the second row shows the throughput for the whole module including the serial to parallel and parallel to dual converters. It was difficult to find any published data regarding the power consumption for convolution encoders or puncturing configurations in order to establish comparisons; however, the first flexible dynamically reconfigurable fabric that provides multi-rate support for convolution coders is introduced here.

3.6 Conclusion

A novel reconfigurable architecture that provides a punctured convolutional coder

Table 3-4. Throughput for all supported rates for the core unit and whole module

Rate	1/2	2/3	3/4	5/6	7/8
	Throughput in Mbps (Delay in ns)				
Core only	200 (10)	300 (10)	400 (10)	600 (10)	800 (10)
Core+RSPC+RPDC [whole module]	66 (30)	33 (60)	28 (70)	20 (100)	15.4 (130)

has been introduced in this chapter. This architecture can be used in wireless communication system incorporating both convolution and puncturing. The convolution-punctured multi-rate architecture has achieved superior throughput between 200 and 800 Mbps. Although the main element of the architecture is the core, which provides the concatenated convolution-punctured code, the reconfigurable input and output interfaces were added to broaden the usability of this fabric. The main advantage of this architecture is that a single clock cycle is sufficient to provide the parallel convolution punctured code for its parallel inputs, which can be used to maximise the throughput of the whole transmitter system. This work demonstrates an example of dynamically reconfigurable architecture for a module within the communication systems. The proposed architecture is capable of dynamically reconfiguring the module rate through the programming code.

Chapter 4

RECONFIGURABLE INTERLEAVER ON DYNAMICALLY CELL-BASED ARCHITECTURE AND AS A FABRIC

4.1 Introduction

In this chapter, a novel reconfigurable block interleaver/de-interleaver is introduced. The block interleaver and de-interleaver are part of the IEEE 802.16 WIMAX transceiver.

The interleaver is widely used in wireless communication systems, such as IEEE 802.11 and 802.16 [44], and also in coders such as turbo coding [58]. The interleaver is one of the main modules of turbo codes [60].

This chapter focuses on the design and implementation of the interleaver /de-interleaver for the WiMAX transceiver. This work is part of the ESPACENET project [46], in which WiMAX has been selected as the preferred communication standard for communication between sensor nodes and cluster heads in space pico-satellite networks. The same concepts and strategies for WiMAX and its usage apply to both mobile devices and pico-satellites. Both have constraint in common, which are mainly size and power.

Integrity in digital devices is an important current challenge in minimising size (area) and reducing power consumption (to give longer battery life). This can be achieved using various strategies. Most published strategies are implementations of novel architectures and/or use advanced technologies with lower voltage supplied or which consume less energy.

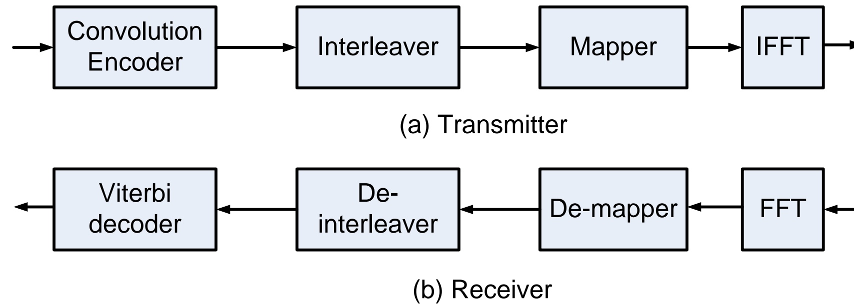

Figure 4-1. The interleaver and de-interleaver in OFDM WiMAX baseband

The interleaver has been designed into a reconfigurable fabric architecture and a dynamically reconfigurable instruction cell-based architecture (RICA).

4.2 Interleaver/De-interleaver

The interleaver is a module in communications system which increases the reliability of transmitted signals by rearranging the in order into various subcarriers. In addition, it ensures that the coded bits are distributed in such a way as to prevent low reliability in the long run. Meanwhile the de-interleaver performs the opposite functions in the receiver, by gathering the distributed coded bits back so that they are adjacent, in order to recreate the same originally transmitted codes. Figure 4-1 illustrates the interleaver's position in the transmitter subsequent to the convolution encoder, and the de-interleaver's place in the receiver next to the de-mapper.

The interleaver is defined by a two-step permutation. The first ensures that adjacent coded bits are mapped onto nonadjacent sub-carriers, while the second permutation ensures that adjacent coded bits are mapped alternately onto less or more significant bits of the constellation, thus avoiding long runs of bits of low reliability [44]. The interleaving in the OFDM WiMAX is based on the data block size used, and the interleaving should correspond to the number of coded bits per allocated sub-channel per OFDM symbol [44]. Table 4-1 presents the various interleaving block sizes that correspond to the number of sub-channels. The interleaving is carried out through two stages or permutations.

Table 4-1 Block sizes of bit interleaver for WiMAX [44]

	N_{cbps}*				
	Number of sub-channels				
	1	**2**	**4**	**8**	**16**
BPSK	12	24	48	96	192
QPSK	24	48	96	192	384
16-QAM	48	96	192	384	768
64-QAM	72	144	288	576	1152

* N_{cbps}: *coded bits per OFDM symbol*

In the transmitter, all encoded data bits will be interleaved by the block interleaver and the size of the block is determined by the parameter N_{cbps}. This represents the number of coded bits per allocated sub-channel per OFDM symbol, and the values are presented in Table 4-1. In digital modulation, an analog carrier signal is modulated by a digital bit stream. The modifications to the carrier signal are chosen from a finite number of alternative modulation symbols. The PSK (phase shift keying) modulation technique is based on using a finite number of phases. For example, BPSK (binary phase shift keying) uses two phases, while QPSK (quadrature phase shift keying) uses four phases. On the other hand, quadrature amplitude modulation (QAM) uses a combination of amplitude and phase. For example, 16-QAM uses four different amplitudes and four different phases, while 64-QAM uses eight different amplitudes and eight different phases [45].

It is obvious from Table 4-1 that, in order to function as an interleaver, various block sizes from 12 up to 1152 bits need to be handled. In order to achieve flexibility in the communication transceiver system a reconfigurable interleaver is necessary to switch between the different data blocks based on the mode of operation.

The interleaver's parameters are determined by specific equations that are responsible for the scrambling of the coded bits. When dealing with a standard, the specific interleaving equations are provided in order to give identical interleaver and de-interleaver functions globally in devices.

From the WiMAX standard [44], specific parameters are defined for the interleaver: N_{cpc} and *s*. N_{cpc} is the number of coded bits per subcarrier, while *s* is

defined by Equation (4-1). The values of both N_{cpc} and the calculated values of s are presented in Table 4-2.

$$s = ceil(N_{cpc}/2) \tag{4-1}$$

The parameters for an interleaver of block size N_{cbps} bits are:

k: index of the coded bit before the first permutation;

m_k: index of that coded bit after the first permutation; and

j_k: index after the second permutation.

The first permutation is calculated using Equation (4-2):

$$m_k = \left(\frac{N_{cbps}}{12}\right) \cdot k_{mod12} + floor\left(\frac{k}{12}\right) \tag{4-2}$$

$$k = 0,1,\ldots,N_{cbps} - 1$$

The second permutation takes place through Equation (4-3)

$$j_k = s \cdot floor\left(\frac{m_k}{s}\right) + \left(m_k + N_{cbps} - floor\left(12 \cdot \frac{m_k}{N_{cbps}}\right)\right)_{mod(s)} \tag{4-3}$$

$$k = 0,1,\ldots,N_{cbps} - 1$$

The de-interleaver performs the inverse operation and is also defined by two permutations. The parameters for a received block of N_{cbps} bits are:

j: index of a received bit before the first permutation;

m_j: index of that bit after the first permutation; and

k_j: index of that bit after the second permutation, just prior to delivering the block to the decoder.

The de-interleaver's first and second permutation are found by Equations (4-4) and (4-5) respectively:

Table 4-2 Value of the *s* parameter in the interleaver/de-interleaver equations

	BPSK	QPSK	16-QAM	64-QAM
N_{cpc}	1	2	4	6
s	0	1	2	3

$$m_j = s \cdot floor\left(\frac{j}{s}\right) + \left(j + floor\left(12 \cdot \frac{j}{N_{cbps}}\right)\right)_{mod(s)} \tag{4-4}$$

$$j = 0,1, \dots, N_{cbps} - 1$$

$$k_j = 12 \cdot m_j - \left(N_{cbps} - 1\right) \cdot floor\left(12 \cdot \frac{m_j}{N_{cbps}}\right) \tag{4-5}$$

$$j = 0,1, \dots, N_{cbps} - 1$$

It is clear from the above equations that the interleaver and deinterleaver execute the same functions but in the opposite manner. The first permutation in the de-interleaver is the inverse of the second permutation in the interleaver and vice versa. As they are identical from a computational point of view, the interleaver will be the focus of this work, and any results obtained will be the same for the de-interleaver.

4.3 Reconfigurable Interleaver

An interleaver is generally implemented in a receiver for a fixed predefined coded data block size. In order to allow scalability in communication systems, integrated modules should be able to process all types of data defined by the standards. In other words, the interleaver should be capable of dealing with all block sizes in that specific system. This will allow the receiver to be capable of dealing with all modulation types thereafter.

Traditionally the block interleaver is based on a LUT ROM (read-only memory) storing the interleaving sequence. The obvious drawback of this method is that the memory locations are the sum of all supported block sizes, which requires a significantly large memory size. However, its clear advantage is the simplicity of the architecture. One multi mode interleaver architecture has been introduced [58] which is based on having a matrix memory or two-dimensional memory, where data is written as rows and read out as columns. The authors realised the design using an 802.11 block interleaver where only four different block sizes were supported. In the case of WiMAX, the interleaver has to deal with various block

sizes from 12 up to 1152 bits of coded data, as mentioned earlier in Table 4-1, which means that twelve different block sizes need to be supported. Overall, there are twenty different modes based on the block size and the associated modulation or constellation type. In order for the interleaver to have this capability, it has to be reconfigurable. Using a configuration word, the interleaver should be able to self-reconfigure toward the desired block size. This is the approach used in this work.

4.4 Reconfigurable Interleaver Fabric

A reconfigurable interleaver fabric is introduced here as a solution which allows the incorporation of all data block sizes. This allows the communication system to be capable of using all modulation techniques and the various block sizes needed by the communication system.

As mentioned earlier in Table 4-1, the interleaver has to deal with various block sizes. It can be noted in the table that the same block size can have different numbers of sub-channels and an altered modulation, as illustrated by the markings in Table 4-3. This means that the interleaver should not only deal with different data block sizes, but also the same data block size which has different interleaving parameters. A conventional notion for solving such a problem is to use parallel fixed interleavers. Based on the desired modulation and number of sub-channels,

Table 4-3 Block sizes of bit interleaver for Wimax. Circle marks show similar block sizes with different modulations and N_{cbps} [1]

	N_{cbps}*				
	Number of Sub-channels				
	1	**2**	**4**	**8**	**16**
BPSK	12	24	48	96	192
QPSK	24	48	96	192	384
16-QAM	48	96	192	384	768
64-QAM	72	144	288	576	1152

* N_{cbps}: *coded bits per OFDM symbol*

Table 4-4 External configuration word for interleaver/de-interleaver

	Modulation type	No. of sub-channels	Block size	Configuration word (bits)
1	BPSK	1	12	00001
2	BPSK	2	24	00010
3	QPSK	1	24	10010
4	BPSK	4	48	00011
5	QPSK	2	48	01011
6	16-QAM	1	48	10011
7	64-QAM	1	72	00100
8	BPSK	8	96	00101
9	QPSK	4	96	01101
10	16-QAM	2	96	10101
11	64-QAM	2	144	00110
12	BPSK	16	192	00111
13	QPSK	8	192	01111
14	16-QAM	4	192	10111
15	64-QAM	4	288	01000
16	QPSK	16	384	01001
17	16-QAM	8	384	11001
18	64-QAM	8	576	01110
19	16-QAM	16	768	10000
20	64-QAM	16	1152	01100

the result would be twenty different interleavers.

A further optimised version of the architecture can be obtained through the reuse of the same interleavers block sizes leading to only twelve parallel interleavers.

Reconfiguration can provide the solution to give a single interleaver capable of carrying out these functions in an optimised approach. In order to configure the interleaver for the various modes, a configuration word is necessary. As there are twenty different combinations, a 5-bit configuration word would be sufficient as shown in Table 4-4. In addition, to provide a seamless interface with other modules in the system, the use of reconfigurable serial to parallel and parallel to serial converters may be appropriate in order to allow a common interface of 2 bits for the input and output of the interleaver. A block diagram for the architecture is presented in Figure 4-2.

Table 4-5 Internal configuration word: 4 bit

	Internal configuration word	Coded data Block Size
1	0001	12
2	0010	24
3	0011	48
4	0100	72
5	0101	96
6	0110	144
7	0111	192
8	1000	288
9	1001	384
10	1010	576
11	1011	768
12	1100	1152

In Table 4-4 it should be noted that the configuration words assigned are not in the form of the usual ascending or descending code. The optimisation of the configuration word used is an essential aspect of reconfigurable architectures.
Configuration in general adds an overhead to the system. Thus, it must be optimised efficiently in order to minimise the overall system overhead. In addition,

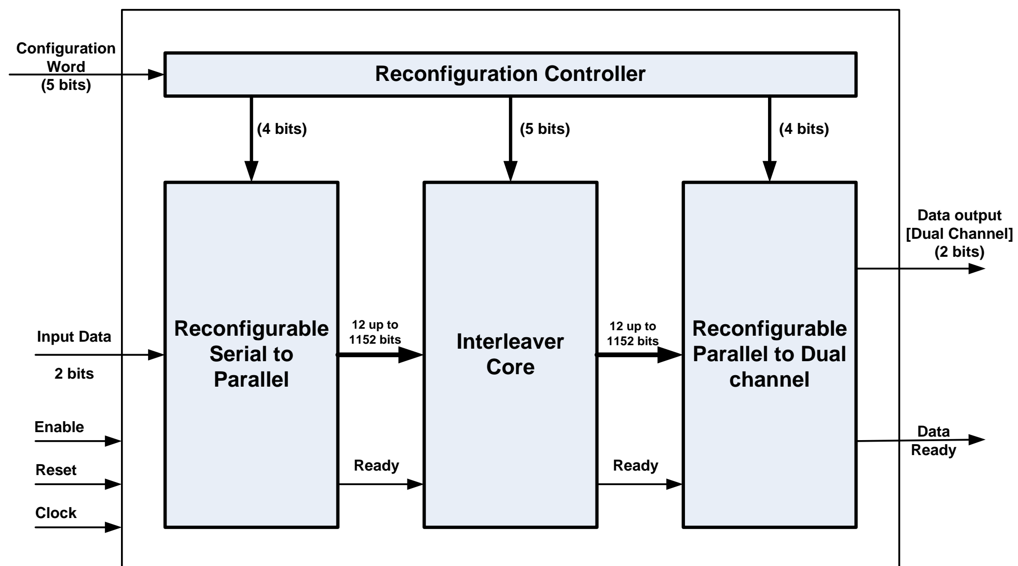

Figure 4-2: Proposed reconfigurable interleaver

optimizing the number of bits used will reduce switching activity, thus maintaining the minimum dynamic power consumption.

The code words in Table 4-4 have been arrived at after several types of assessment. Those codes have been specifically chosen in order to eliminate the usage of a decoder to generate the internal configuration words (4 bits) presented in Figure 4-2. The internal configuration words listed in Table 4-5 are used to configure the serial to parallel and parallel to serial converters. In other words, the external configuration words are oriented to block size.

It is worth mentioning that the configuration word 11111 has been reserved as a software reset for the entire interleaver. The interleaver is based on two shuffling stages, and therefore a Matlab model has been designed and programmed in order to solve and simulate the two main equations (4-4) and (4-5) of the interleaver. Some of the Matlab models are presented in Appendix A. It is concluded that it will be too complicated and unrealistic to implement the equations themselves the Interleaver fabric. A more practical approach is to create an array of input indexes and the resulting indexes for each case. These indexes result from Matlab calculations for the combined permutation stages. Thus, the interleaver core is configured through the configuration word, knowing the coded data block size, number of associated sub-channels and the modulation used and then placing them in to the pre-calculated indexes.

From Figure 4-2, the reconfiguration controller is a straightforward controller to pass the specific configuration bits to specific units. The Interleaver core is based on having a single array of registers (memory locations) of a size of 1,152 bits, which is the maximum data block size required. In contrast, a traditional realisation would need at least 4,836 bits [58].

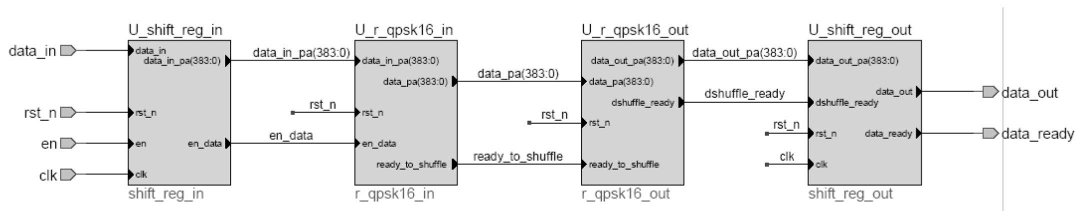

Figure 4-3. WIMAX interleaver for QPSK modulation

4.5 Interleaver on a Dynamically Reconfigurable Processor

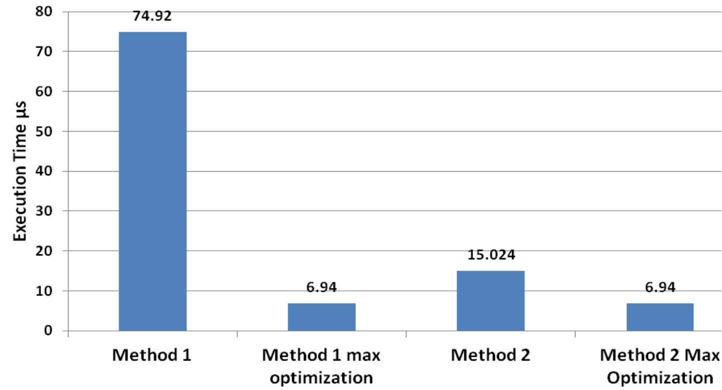

Figure 4-5. Architecture's execution time in µs for interleaver 576 64-QAM with two design methods and their optimisations

The majority of VLSI architectures employ a pipelined data path to gain timing advantages. The concept used in pipeline acceleration is to reconfigure pipelines or parts of pipelines onto a reconfigurable architecture. The reconfiguration allows one stage of the pipeline path to be configured in every cycle, while concurrently executing all other stages. The reconfiguration is usually conducted at run time (dynamic) in which the time for reconfiguration is kept as short as possible. For the interleaver design here, the reconfigurable instruction cell array (RICA) as

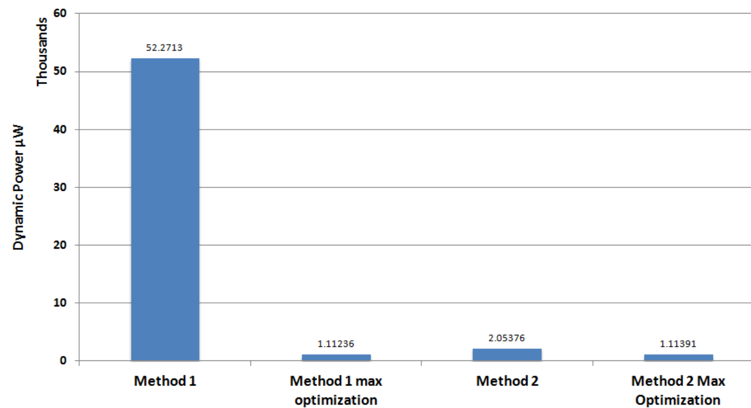

Figure 4-4 Architecture Cells dynamic power in µW for interleaver 576 64-QAM with two design methods and their optimisations

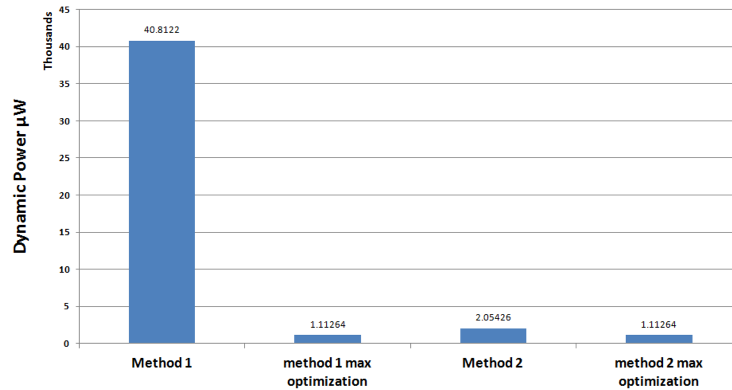

Figure 4-6. Architecture cells dynamic power in µW for interleaver 768 16-QAM with two design methods and their optimisations

discussed in section 2.4.21 is used for the realisation.

The interleaver is based on two shuffling stages as shown in by the two de-Interleaver main equations (4-4) and (4-5). Two implementation methods are used here to design the interleaver into the RICA. The first is the direct implementation of the equations on the reconfigurable architecture (RICA), and this is called Method 1. This method benefits from the fact that the architecture is C-programmable. The other method is similar to the design of the reconfigurable fabric in hard-coded implementation, and this is called Method 2. The hard-code implementation is based on calculating the new bit positions offline using Matlab and storing these values in an array which will be used on the RICA to reorder the input data blocks.

A series of optimisations has been carried out using both methods with different block sizes. The optimisations include partial loop unfolding (LUF), full loop unfolding (FLUF) and parallelism extraction. Both methods have to be designed in the C language and compiled by the processor compiler, optimising the assembly code and analysing the resulting code and the associated performance report.

Figure 4-5 and Figure 4-4 show the execution time and dynamic power consumption respectively for interleaver 576 64-QAM. The maximum optimisation in the figures indicates the optimum results for the designated method.

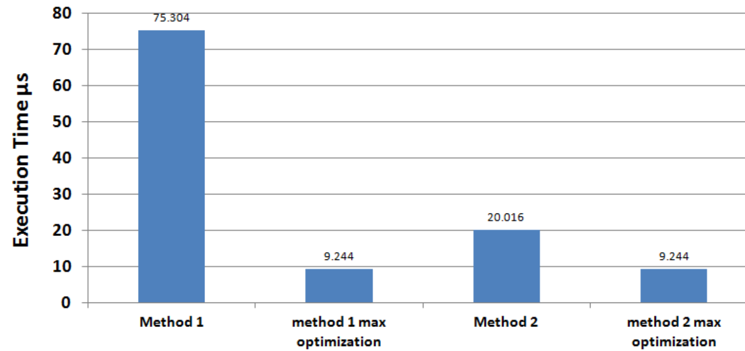

Figure 4-7. Architecture's execution time in µs for interleaver 768 16-QAM with two design methods and their optimisations

Figure 4-6 and Figure 4-7 show the resulting execution time and dynamic power for respectively the design of the 768 16-QAM block interleaver on RICA.

From Figure 4-5 to Figure 4-7, it is clear that both methods achieved almost the same performance. From the analysis of the results of the two implementation methods for the Interleaver, the two methods merged into one giving the same results after the full optimisation implemented. Therefore, it turns to merge to the same concept of the second method. This explains why the maximum optimisation in both methods gave approximately the same results.

4.6 Results and analysis

In this chapter, the reconfigurable block interleaver has been designed for fabric and dynamic reconfigurable architecture realisations. Various optimisations and approaches have been discussed and the results have been presented. In this section, only the best results for all designs are presented.

For the reconfigurable fabric, and as expected, the best throughput and execution time were achieved in comparison with the RICA architecture. For example, for the interleaver 576 bit 64-QAM, the throughput achieved on the reconfigurable fabric (ASIC) is 99.8 Mbps (5.77 µs) compared with 83 Mbps (6.94 µs) for the

reconfigurable architecture. This means that the reconfigurable architecture overhead compared to ASIC is around 16.8% of the execution time or throughput.

From a power consumption standpoint for the same interleaver, the RICA's

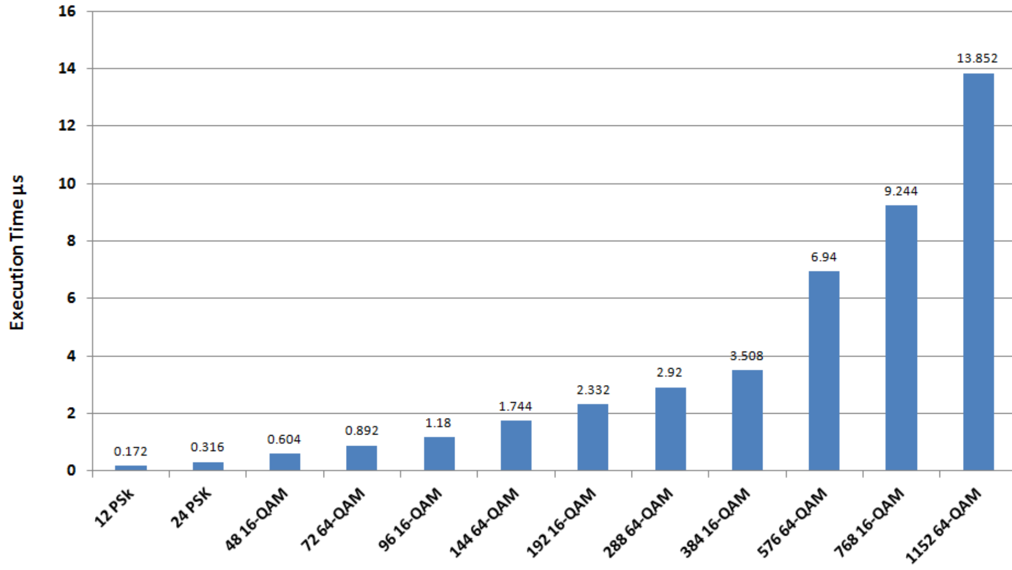

Figure 4-8. Reconfigurable interleaver execution time in µs for the various modes on the dynamically reconfigurable architecture RICA

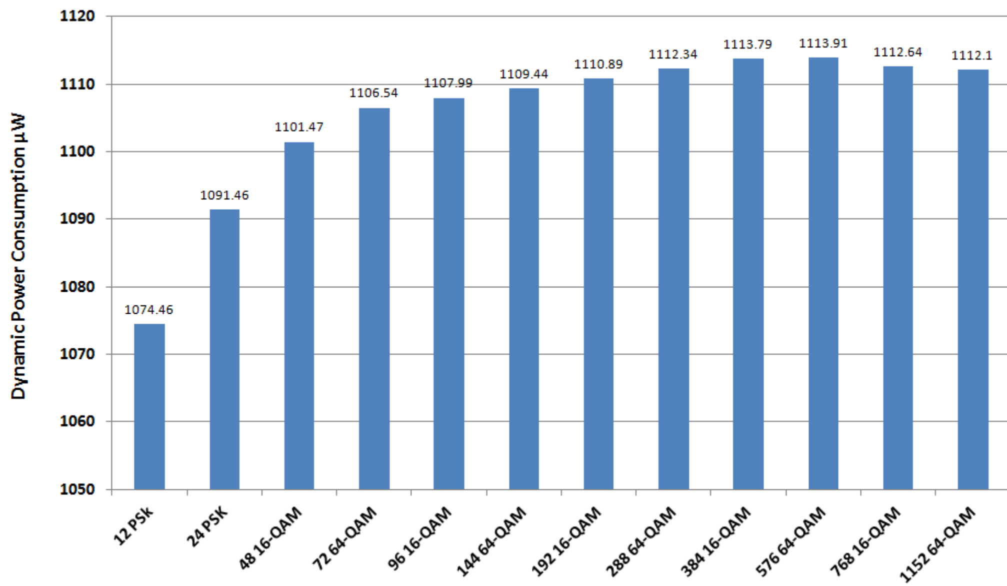

Figure 4-9. Reconfigurable interleaver dynamic power consumption in µW for the various modes on the dynamically reconfigurable architecture RICA

dynamic power consumption is in the range of 1.11 mW[1]. It is expected that the ASIC would be superior in power consumption; however, the power achieved is around 50.15 mW. The reason for this high value is that the ASIC library uses registers (up to 5760) instead of RAM memory, which negatively affects the value achieved and this is clearly not representative of real ASIC performance.

The focus of this work is the dynamically reconfigurable interleaver that would support all block sizes with adequate performance. Figure 4-8 presents the execution times of all the block sizes for the interleaver on RICA. As expected, the execution time is proportional to the block size executed, with the highest being the 1152 block size.

Figure 4-9 presents the dynamic power consumption for all block sizes. It is expected that the power consumption will increase slightly with the size of the block of data, and it almost stabilises around 1110μW, which is as expected. This represents the full utilisation of the processor's resources. In a previous study, an ASIC chip was designed for a full baseband processor for 802.11a [61]. The interleaver and de-interleaver are less sophisticated in 802.11a due to the limited block sizes supported. The authors reported that the power consumption of the

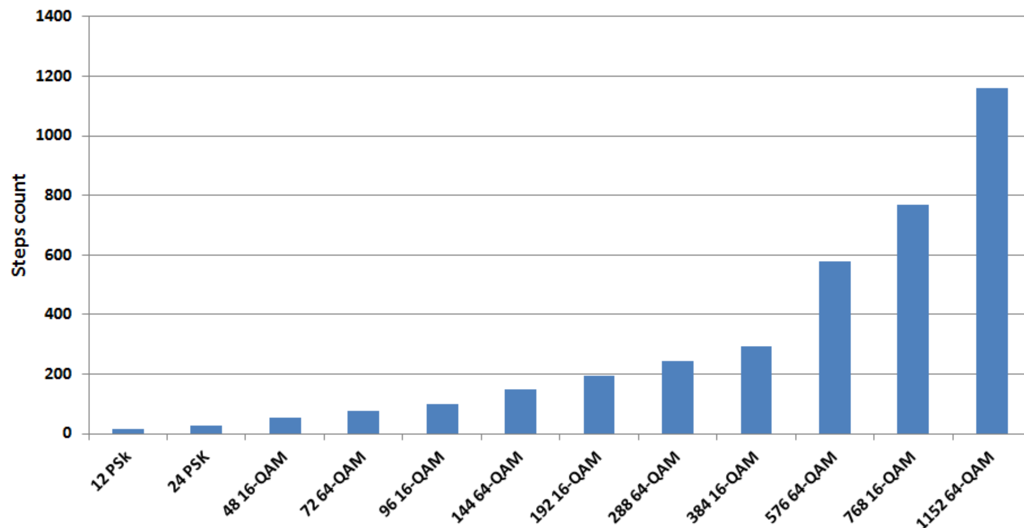

Figure 4-10. Reconfigurable interleaver "steps" count for the various modes on the dynamically reconfigurable architecture RICA

[1] Memory access and interconnection power are not included

interleaver and de-Interleaver was 13 and 21 mW respectively. This compares to the RICA reconfigurable interleaver which gives ultra-low power consumption at 1.1 mW.

Furthermore, the designed interleaver/de-interleaver reconfigurable fabric area is 0.529 mm^2, while in a previous study [61] the areas reported were 0.501 mm^2 for the interleaver and 1.786 mm^2 for the de-interleaver.

Figure 4-10 shows the numbers of steps used for each block size on RICA. A step is the number of reconfigurations required to accomplish the full interleaving or de-Interleaving process. As anticipated, the number of steps is proportional to the block size processed.

4.7 Conclusion

A novel reconfigurable interleaver has been presented in this chapter. The target application was the 802.16 standard with its sophisticated block size requirements. The interleaver has been researched and designed into a reconfigurable fabric architecture and a dynamically reconfigurable instruction cell-based architecture (RICA). The interleaver throughput as a reconfigurable fabric exceeds the standard requirement (up to 70Mbps), having a throughput of 99.8 Mbps at 576 block size for 64-QAM modulation. Meanwhile on RICA the throughput as well as the dynamic power consumption were superior to the fabric realisation and other ASIC realisations. These results are a good step forward towards a fully reconfigurable baseband telecommunications system. Moreover, the results are a promising step in integrating all WIMAX modules on a dynamically reconfigurable architecture.

Chapter 5

DYNAMICALLY PROGRAMMABLE REED SOLOMON PROCESSOR

5.1 INTRODUCTION

With the rapid progress of communication technologies, various communication standards have emerged [61]-[63]. Quality of service (QoS) is one of the most important factors in mobile communication networks. Reductions in delays, including those due to processing time and error correction, are proposed to achieve higher levels of QoS [64]. Reed-Solomon coding is one of the most important schemes for error detection and correction. The Reed-Solomon codes are named after their originators [65] and are widely used in digital communication systems. There is a great demand for present and future devices to integrate various applications and communication standards in the same device. Such integration can be accomplished by, for example, having GSM, WiFi and WiMAX communication capabilities on the same unit in addition to video reception through DVB-H. This is a tough challenge for battery-powered handheld devices. Achieving this level of integration will require an ultra-low-power platform with a less complex design flow that provides a shorter time-to-market. Well-known platforms are the ASIC and FPGA. ASIC technology faces several limitations, such as lack of flexibility and relatively slow time-to-market. On the other hand, FPGAs have high flexibility but are not suitable for handheld devices due to their high-energy consumption. Hence, the development of implementation methods has targeted systems based on digital signal processors (DSP) [62].

This chapter presents a low-power reconfigurable Reed Solomon (RS) processor which can support various communication standards such as WIMAX [44] and

DVB-H [73]. The proposed reconfigurable RS processor is intended to be programmable for different communication standards.

Reed Solomon codes are constructed and decoded using a type of finite field arithmetic which is known as the Galois Field (GF) in honour of its inventor. A finite field of q elements is usually denoted as GF(q) [66]. RS code is constructed in GF, which has its own calculation theorem. Special calculation elements are needed for the implementation of its coding and decoding, such as the GF multiplier and GF adder. For the GF multiplier to be implemented on FPGAs, it a large number of adders and shifters or look-up tables on normal fine-grained hardware platforms may be required. This may lead to excessive delays [66]-[67] as well as high energy consumption. Some DSP processors, such as the TI C64x, already have their own embedded GF multipliers, but their high-energy consumption cannot be ignored in mobile communication systems. Moreover, an architecture that supports multi-standard wireless communication systems should support programmability. Thus, a real time programmable Reed Solomon coding processor is investigated here.

In this chapter, the endeavour is to design an architecture that is capable of supporting the computational requirements of Reed Solomon coding, while also maintaining flexibility and the capability for integration with other systems. The processor architecture is based on the reconfigurable cell-based array architecture approach [35]. The RS processor is based on an array of heterogeneous cells, each of which supports a primitive operation such as addition, multiplication, shift, logic operation, write memory, read memory, multiplexing and so on.

The chapter is organised as follows. Section 5.2 introduces the RS algorithms, while section 5.3 presents the novel processor architecture. Section 5.4 addresses the RS codec implementation and Section 5.5 presents a novel Galois Field multiplication cell design and implementation. Section 5.6 then discusses the implementation and optimisations of the novel RS processor, and the results and comparisons are given in Section 5.7, while the chapter's conclusions are listed in section 5.8.

5.2 RECONFIGURABLE RS-CODEC ALGORITHMS

RS codes are based on adding redundancy symbols to data that will in turn allow the encoder to code a block of data and the decoder to correct up to (t) error symbols. The number of redundancy symbols added is equal to 2t. The implementation of the RS coder is based on GF (Galois Field) polynomials and exponential representations of the field elements. The number of elements in a finite field must be in the form of p^m, where p is a prime integer and m is a positive integer. A primitive element is a root of a primitive polynomial $p(x)$.

The order of an element α in GF(q) is the smallest positive integer m such that $\alpha^m = 1$. GF(q) always contains at least one element, called a primitive element, that has the order (q - 1). Let α be the primitive in GF(q). Since (q - 1) consecutive powers of α, $\{1, \alpha, \alpha^2, \ldots, \alpha^{q-2}\}$ must be distinct, and they are the (q - 1) nonzero elements of GF(q). The 'exponential representation' of the nonzero elements in the field provides an obvious means for describing the multiplication operation: $\alpha^x \bullet \alpha^y = \alpha^{(x+y)}$. A primitive element is a root of a primitive polynomial $p(x)$. The exponential representation for the nonzero elements of GF(q) is given by the reduced modulo of the primitive polynomial to obtain a 'polynomial representation,' which is used in the addition operation [66].

The addition operation is to be carried out using the 'polynomial representation' of the field elements. This polynomial representation is obtained by having the nonzero elements of GF(q) in exponential form as the reduced modulo of the primitive polynomial [66]. For the RS decoder, the chosen algorithm is composed of the Berlekamp Massey algorithm (BMA), the Forney algorithm and the Chien search. A detailed description of the encoder/decoder algorithms has been given elsewhere [66] and [81].

The RS codes have several parameters to be programmed targeting several applications, especially multi-standard wireless communication. The present approach was targeted the RS encoder for the GF(2^8), where the symbol width is

fixed to 8-bits in order to minimise power consumption and maximise throughput. $GF(2^8)$ is also common to all of the applications mentioned earlier. The programmable parameters n and k are the key in the encoding process, in addition to the primitive polynomial used. There can be sixteen different fields defined over $GF(2^8)$ through 16 primitive polynomials, where the default primitive polynomial is $x^8+x^4+x^3+x^2+1 = 285_{decimal}$, while the other 15th are (decimal values): 299, 301, 333, 351, 355, 357, 361, 369, 391, 397, 425, 451, 463, 487 and 501.

5.3 RECONFIGURABLE RS PROCESSOR

The aim in this work is to design a low power architecture that is capable of supporting the computational requirements of the RS codec. In order to cater for these requirements, the processor used is based on the reconfigurable instruction cell architecture (RICA) [35]. Thereafter the processor is specifically tailored to be reconfigurable for the RS codec, so that it can be easily configured and programmed real-time in a dynamic manner. The processor is DSP-like, hence keeping the advantage of DSPs of high-level programmability. Despite being DSP-like, it is an ultra-low-power architecture due to the ability to dynamically reconfigure highly optimised data paths at given instants in time.

The RS processor cells are flexible enough to work with 32, 16 and 8-bit data types. Compared with the field programmable gate array (FPGA), the RS processor can be dynamically reconfigured so that unallocated cells can be eliminated (disconnected) at each step; thus, the energy consumption can be limited rather than using the majority of available transistors to provide flexibility as in the FPGA.

The processor is based on an array of heterogeneous cells as shown in Figure 5-1. Each cell supports a certain operation. Various operations are included in the processor, such as addition, multiplication, shift, logic, write memory, read memory and multiplexing. These cells are based on 32-bit operands interacting with a distributed memory of 16 banks with an 8-bit memory bank width. The 32-

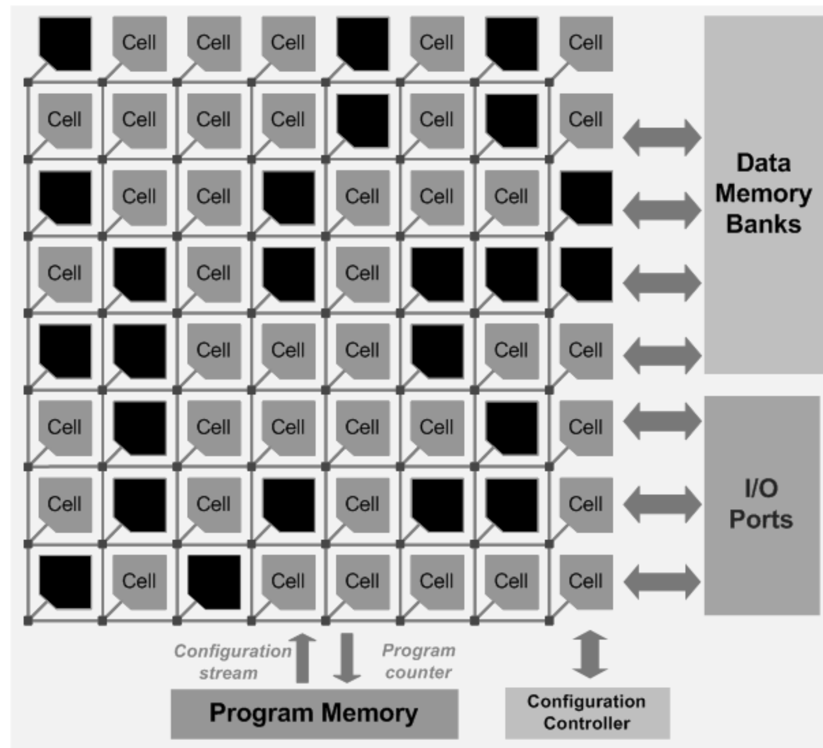

Figure 5-1 Reed-Solomon processor based on dynamically reconfigurable heterogeneous cell array.

bit architecture is used as this provides a greater capability for implementing parallelism in the 8-bit based Reed Solomon algorithm.

The processor has a reconfigurable data-path that implies non-fixed cycles, but is based on a ‘step’ concept. A ‘step’ is a combination of instructions, or physically a single datapath interconnection configuration for a group of cells in the processor. Step size is determined by the resources available in the processor, a conditional branch (Jump) and the length of the critical path. The step concept allows the maximum exploitation of parallelism for the implemented application instead of having fixed data-paths as in traditional processors.

RS codes are defined by two main parameters: the number of overall symbols after encoding (n) and the number of data symbols before encoding (k). the number of data symbols that can be corrected is t, which equals (n-k)/2.

The encoder in the present work uses the GF(256) with a symbol width of 8 bits. For the purpose of research, validation and comparison, two different RS block

sizes (255,239 and 204,188) which are used in WiMAX and DVB-H respectively are targeted.

This includes the famous RS(255,239), as well as other combinations, such as the RS(204,188) for DVB (Digital Video Broadcasting).

5.4 RS Encoder and Decoder Implementation on Novel RS Processor

5.4.1 RS Encoding and Novel Design

For the ASIC and FPGA, the classic implementation of RS is based on the use of linear feedback shift registers (LFSRs). Figure 5-2 (a) shows the ASIC/FPGA-based classical architecture for an RS encoder. Figure 5-2 (b) presents a novel design modification for the RS encoder that includes parallel parity symbol outputs. In classical implementations, the system requires an additional $2t$ clock cycles to generate the calculated parity symbols after the k clock cycles used for parity calculation, resulting in a total number of clock cycles of (k+2t). However, with the new design the number of clock cycle is reduced to only k, since the parity bits are output in parallel without the need for additional clock cycles. This implementation strategy increases the throughput of the whole communication system by n-k-1 times. Moreover, power savings are achieved due to the reduction in the number of clock cycles used, as indicated below.

Time reduction = 1- [k / (2t + k)] • 100% per data block,

For example, for the RS(255,239), the time reduction is 6.3%, while for the RS(204,188) it is 7.8%

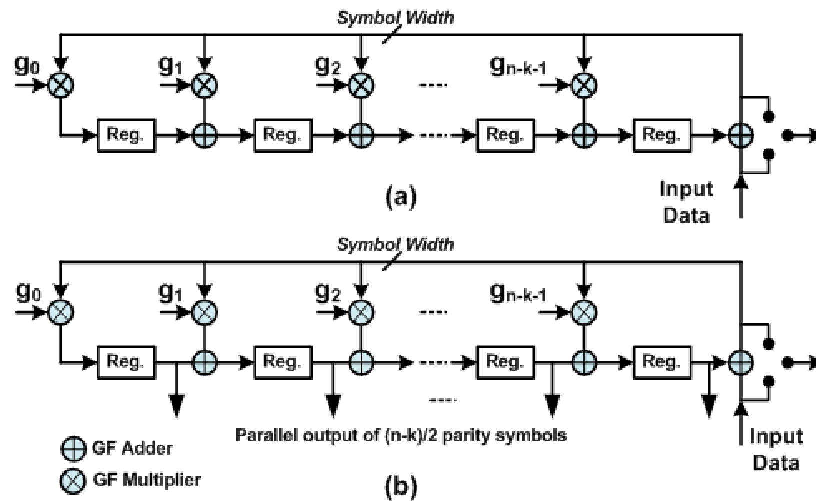

Figure 5-2. Reed-Solomon encoder using linear feedback shift register with n-k stages: (a) classical RS encoder architecture; (b) novel design of parallel parity output

The complex part of the LFSR implementation is the GF multiplier. The multiplication process changes radically when the primitive polynomial is changed. For the efficient implementation of the real-time programmable RS encoder on a C-programmable architecture, a hardware algorithmic technique has been implemented. Various optimisation techniques need to be applied to the architecture in order to achieve the best possible results. This approach is named a 'hardware-approach' since it resembles ASIC/FPGA implementations and is based on the novel architecture shown in Figure 5-2 (b). An advantage of applying this approach is the reduction of processing time by 7.8% which will result in a reduction in the total number of operations and memory accesses required. Moreover, the omission of the modulus function is an additional advantage. There is a drawback associated with the implementation using this approach, which is the need for a new cell type to be added to the processor: a GF multiplier.

5.4.2 RS Decoder on RS Processor

The RS decoding algorithm can be divided into five main steps, as illustrated in Figure 5-13.

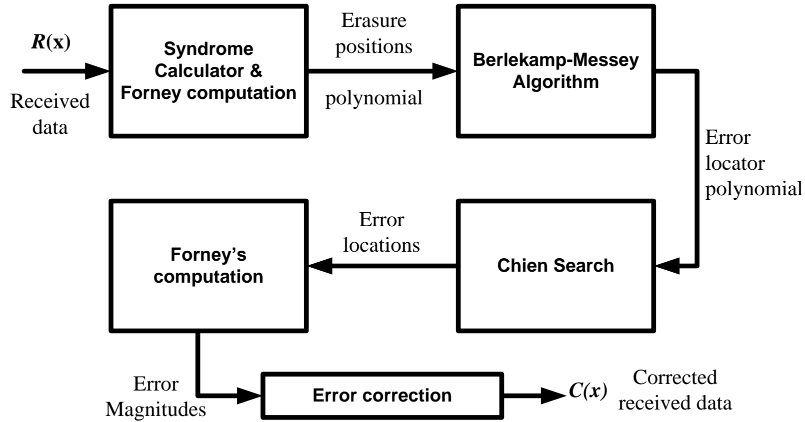

Figure 5-3. Reed Solomon decoder main algorithms

5.4.2.1 Syndrome computation

in syndrome computation the syndrome polynomial $S(x)$ is calculated, which is denoted as $\sum_{i=1}^{2t} S_i x^i$. By definition, $S_i = R(\alpha^i)$, where $R(x)$ is the received codeword (in polynomial form) and α^i are the roots of the codeword-generating polynomial for i=1, 2 … 2t [76].

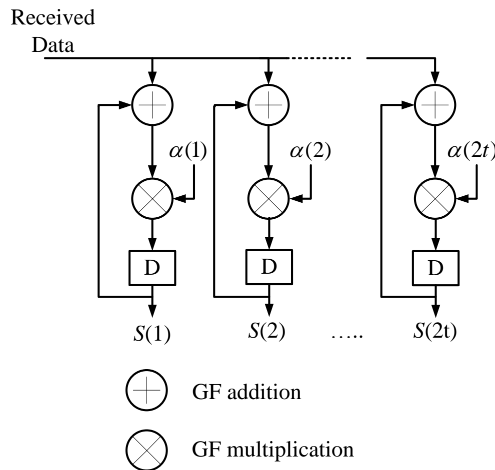

Figure 5-4. Syndrome computation architecture

The algorithm can be expressed as

$$S_i = R_0 + R_1\alpha^i + R_2\alpha^{2i} + \cdots + R_{n-1}\alpha^{(n-1)i} \tag{5-1}$$

Horner's rule is a method for reducing the number of multiplications in polynomial computation [76]. According to this rule, the expression of syndrome computation can be written in a nested multiplication form:

$$S_i = (\cdots((R_{n-1}\alpha^i + R_{n-2})\alpha^i + R_{n-3})\cdots)\alpha^i + R_0 \tag{5-2}$$

The syndrome computation architecture is shown in Figure 5-4. It employs a recursive construction with GF adders and GF multipliers.

5.4.2.2 *Forney computation*

If erasures exist, an erasure position polynomial as in Equation (5-3) would be generated, in which num_{era} is the number of erasures, and ε^i means the positions of the erasures which have occurred. In this case, the syndrome polynomial should be updated as in Equation (5-4)

$$\Lambda(x) = \prod_{i=1}^{num_{era}} (1 + \alpha^{\varepsilon_i} x) \tag{5-3}$$

$$S'(x) = S(x)\Lambda(x) \bmod x^{2t} \tag{5-4}$$

5.4.2.3 *Key equation calculation*

If the syndrome polynomial $S(x)$ is non-zero, mean errors or erasures are detected. Key Equation (5-5) is generated in order to obtain the error location polynomial $\sigma(x)$ and the error value polynomial $\omega(x)$

$$S(x)\sigma(x) = \omega(x) \bmod x^{2t} \tag{5-5}$$

For this nonlinear Key equation, $2t$ simultaneous equations need to be computed [66]-[78]. Two main algorithms can be utilised: Euclid's algorithm and the Berlekamp Massey algorithm (BMA) [79]. In this work, the BMA is employed because it is considered to entail less hardware complexity [80]. This is due to its nature as an algorithm that will find the shortest linear feedback shift register (LFSR) for a given binary output sequence. After $2t$ iterations of BMA, the error

location polynomial $\sigma(x)$ can be obtained. If any erasure occurs, the erasure information should be added into $\sigma(x)$.

5.4.2.4 Chien search

A Chien search concerns finding the roots $\gamma_l (1 \le l \le t)$ of the error location polynomial $\sigma(x)$ [76][79]. The basic idea of the Chien search is to evaluate the error location polynomial with 255 possible roots of GF and to check if the result is zero. If it is, this indicates that a root has been found [81]. With the Chien search, both the location and number of errors can be obtained.

5.4.2.5 Error evaluation polynomial

The error evaluation polynomial is calculated from the syndrome and the error polynomial. It can be expressed by Equation (5-6).

$$\omega(x) = S(x)\sigma(x) \bmod x^{n-k} \tag{5-6}$$

5.4.2.6 Forney algorithm

The Forney algorithm in Equation (5-7) is used to compute the error values from the error location polynomial and error value polynomial:

$$e_l = \omega(x) / \sigma'(x),\ x = \gamma_l \tag{5-7}$$

where $\sigma'(x)$ is the odd term of $\sigma(x)$.

5.4.2.7 Error correction

After the roots of both $\sigma(x)$ and $\omega(x)$ have been calculated, the data received can be corrected by a simple XOR operation of the error value and the received symbol at the corresponding error position.

5.4.2.8 Single instruction multiple data (SIMD)

The key optimisation target is to maximise step size and minimise the resources and memory access used. In the RS (255,239) decoder design, the maximum numbers of GF adders and multipliers required by each functional block have been calculated and are shown in Table 5-1.

Table 5-1 Number of additions and multiplications in the Reed-Solomon decoder

Block	Additions	Multiplications
Syndrome computation	4080	4080
Forney computation	Number of erasures	Number of erasures
Key equation calculation	16 iterations, depending on the number of errors and erasures	16 iterations, depending on the number of errors and erasures
Chien search	4335	4335
Forney algorithm	depending on the number of errors and erasures	depending on the number of errors and erasures
Error correction	8	0

NB. *Calculations are based on the RS (255,239)*

5.5 Galois Field Multiplier Cell for RS Processor

As discussed above, there is a need for a GF multiplier (GFMUL) custom cell to be designed and implemented within the processor in order to support the RS encoder and decoder optimised algorithms and to enhance their performance on the architecture.

In order to implement the cell in a programmable RS processor, the cell itself has to be programmable. The objective here is to focus on the RS coding for the GF(2^8), where the symbol width is fixed at 8-bits so as to minimise the energy consumed and to maximise throughput. Moreover, GF(2^8) is common to all of the applications

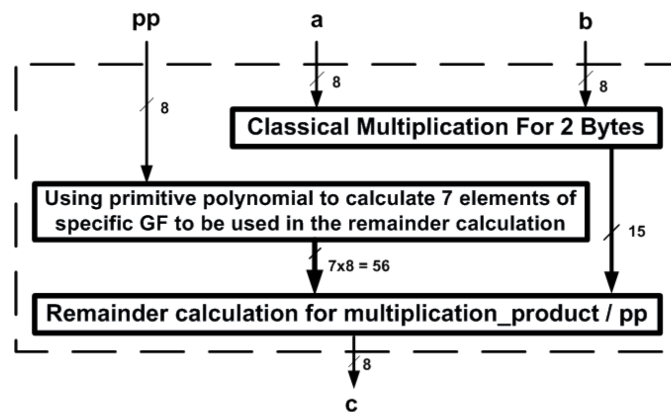

Figure 5-5. The internal architecture of a single GF multiplier for 8-bit data width

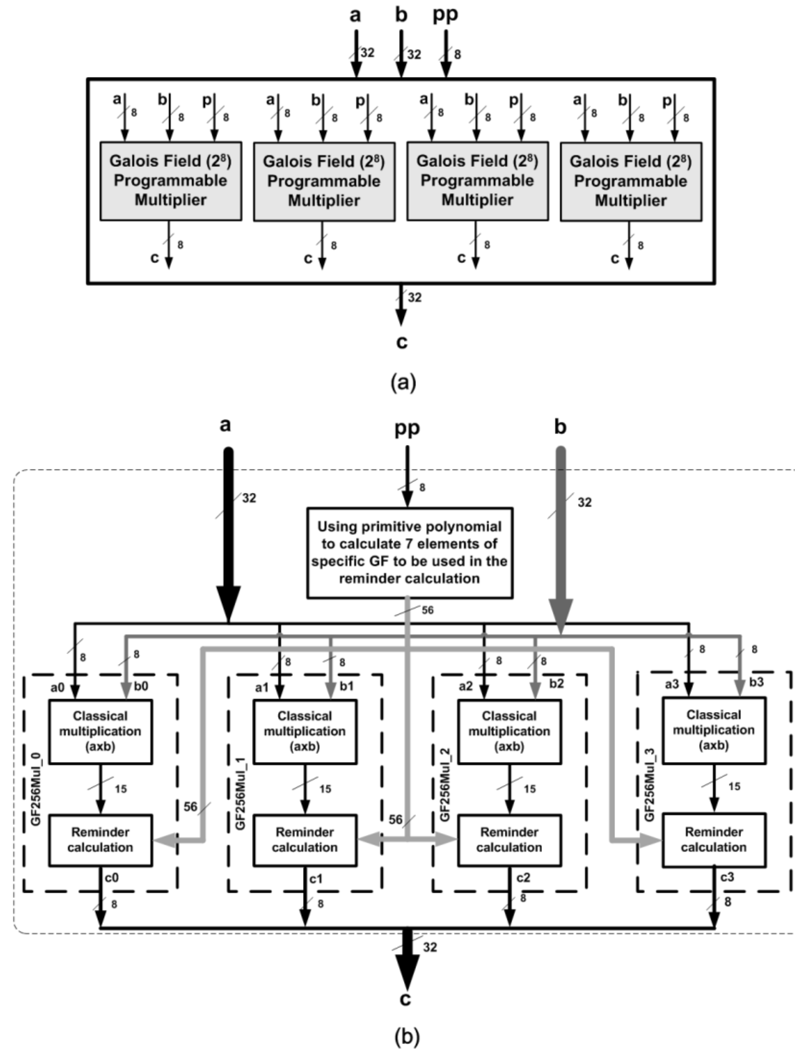

Figure 5-6. A novel 8 bit GFMUL cell with four embedded GF multipliers maximising the throughput by applying the SIMD technique (a) GF multiplier cell layout, (b) GF multiplier internal structure

mentioned earlier in Section 5.1. There are up to sixteen different fields defined over GF(2^8) by sixteen different primitive polynomials. Programmability here from an algorithmic point of view can be divided into two types: real-time and offline. Real-time programmability is used for switching between the different primitive polynomials used; while offline programmability refers to when the values of n and k need to be changed.

A combinatorial GF multiplier is designed, as this will suit the RS cell-based processor. The cell function is modelled and verified in Matlab. Then it has been designed, simulated and verified in both pre- and post-routing using Cadence VLSI

design EDA tool set using the UMC 0.18μm (UMCL18U250D2_2.4) CMOS technology library (which is the same technology as that used for the processor).

This enables the performance and characteristics of the new cell to be tested, verified and extracted. Thereafter, the cell and its characteristics were embedded within the overall processor model. The necessary adjustments have been added to the processor compiler in order to add the GFMUL into the processor instruction set as a new added instruction that will invoke a GFMUL custom cell instead of initiating a series of shift and logic instructions or cells.

Figure 5-5 shows the internal architecture of the 8-bit multiplier, with ‘pp’ as the primitive polynomial reconfigurable input. The RS Processor is 32-bit based; and this is why it is preferred to make use of full data-bus width. Hence, instead of having only an 8-bit-based cell, a novel 4 x 8-bit-based GFMUL cell has been designed. The cell is based on four concatenated 8-bit GF multipliers, as shown in Figure 5-6. This optimisation enabled a cell area reduction of approximately 23% with an associated reduction in energy consumption since only a single remainder calculation unit is needed instead of four in the case of four independent GF multipliers, as shown in Figure 5-5. In addition, this allows the single instruction multiple data (SIMD) technique to be applied in the implementation of the algorithms, in order to maximise throughput and resource utilisation, due to the concurrent use of the four multipliers within the cell. The results for the novel cell are presented below in Section 5.7.1.

5.6 RS Processor Implementations and Optimisations

5.6.1 Architecture Specific Optimisations

The RS processor is based on the dynamically reconfigurable architecture paradigm as presented in Figure 5-10 (b). The processor executes the codes in ‘steps’ instead of using a single instruction at a time. At each step, the instructions are loaded into

the processor's 'configuration controller', which introduces configuration latency. The architecture is structured to support only one 'Jump' instruction per step. Following the Jump, a new configuration will take place to reconfigure the processor cells and interconnections and load a new set of inputs based on the algorithm and the sequences of its instructions. If an algorithm fully or partially generates or includes more than one Jump, then the code in between will be placed in separate steps. The main reason for optimisation is to maximise processor utilisation by reducing the number of steps as well as the length of the critical paths. If the entire algorithm can be placed in a single step, then the configuration latency will be almost eliminated. If Figure 5-10 (b) is considered as a stand-alone case, and assuming that the entire algorithm has been encapsulated in a single step, then the processor will run on a single configuration, and the only ongoing changes will be in the data handled through input and output ports. This is the ultimate goal in optimising processor performance, since this can be expected to provide the best performance.

In initial implementations, the configuration time and performance results are not usually the best that might be achieved, since long critical paths and huge computational resources are required. There is a trade-off between configuration time, step size and the number of cells in the processor. In general, maximising the resource utilisation is a common target, as this leads to better performance by increasing processor throughput and eliminating redundant cell resources.

Memory access delay is another key performance parameter which needs to be monitored. This parameter can be a major bottleneck for performance in embedded systems [75]. Memory access optimisation can be achieved in three steps: a) algorithm-level optimisation by modifying the algorithm in order to optimise and limit the need for frequent memory access; b) increasing 'local register' cells that can hold intermediate calculated results; and c) applying pipelining and using register files. Conversely, there is a trade-off when using registers, since their excessive usage can have an adverse effect on throughput and, most importantly, on the power consumption. Parallelism and pipelining are explained further below.

5.6.1.1 Parallelism

The RS processor can execute both dependent and independent instructions in the same processor step, so that parallelism is utilised. For the encoder, 16 redundant code-words can be calculated in parallel to enhance performance. For the decoder, syndrome computation, the Chien search, error evaluation and the Forney algorithm can be implemented in parallel as well, which leads to a throughput enhancement 25%, and 23.3% reduction in the energy consumption memory access

5.6.1.2 Kernel and pipeline

The RS processor is specifically optimised to execute large steps that loop back onto themselves (termed 'kernels') [35]. In a kernel, the processor can fetch and store the complete set of configuration instructions only once, instead of fetching the same instructions repeatedly, so that configuration latency and energy consumption are greatly reduced. A software pipelining technique can be utilised for kernels to give for further performance improvements. The kernel step will be automatically pipelined into multiple stages with a special mark-up added in the software code.

In this case, the critical path of the kernel will be shortened. Therefore, the overall execution time will be reduced and hence throughput will be improved. This pipelining technique is especially efficient for the decoder. After building the kernels and pipelining, the critical path has been shortened, and thus the processor's performance has been improved.

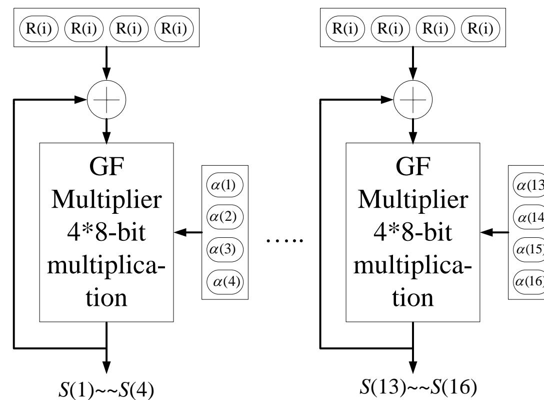

Figure 5-7. SIMD architecture for syndrome computation

5.6.1.3 Dedicated GF MUL and SIMD

It is obvious that the syndrome computation and Chien search require the most resources. In Reed-Solomon code, the width of a symbol codeword is 8-bits. The Single instruction multiple data (SIMD) is employed here to give the maximum usage of the processor's resources.

For example, in syndrome computation, all the terms of $S(x)$ are independent. With the SIMD technique, the received four 8-bit data and four 8-bit α^i can be combined together to form two 32-bit operands. Thus the four multiply operations can be calculated with a single 4 x 8-bit GF multiplier. The output $S(x)$ can also be segregated into four 8-bit operands. Figure 5-7 illustrates the architecture of syndrome computation with the built-in SIMD technique. Compared with the classical one method without SIMD shown in at Figure 5-4, the number of GF multiplications are reduced by 75%, and the memory access energy is reduced by 48%.

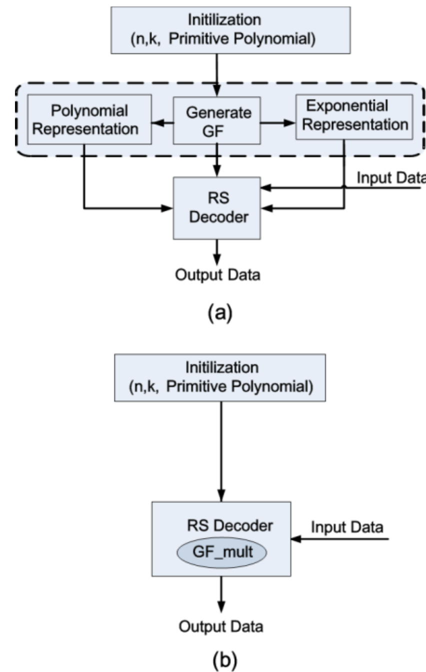

Figure 5-8 Reed Solomon decoder (a) classical software approach of RS decoder; (b) new approach with GF multiplier cell

5.6.1.4 Architectural approach to RS decoding

The RS decoder requires more GF multiplications than the encoder. Figure 5-8 (a) illustrates the architecture of a classical software RS decoding implementation which uses a LUT (look up table) to utilise the GF multiplication. With the use of the LUT, enormous numbers of memory accesses will be introduced into the application, which will lead to bring considerably increased execution time and energy consumption. Therefore, a GF multiplication cell is essential for both the RS encoder and decoder.

Figure 5-8 (b) illustrates the new RS decoder approach with the GF multiplier cell. Compared with the classical software approach, the architecture with the GF multiplier cell is less complex, and the large arrays that were used for holding the GF elements are eliminated. By reducing the number of memory access operations, the GF multiplier custom cell will greatly improve performance in terms of reducing the execution time and energy consumption.

5.6.1.5 Targeted processor's architecture

As explained earlier, the RS processor is based on the RICA paradigm. Hence it is worth explaining here in greater detail the layout of the architecture and its modes of operations. The processor is C-programmable and based on heterogeneous cell

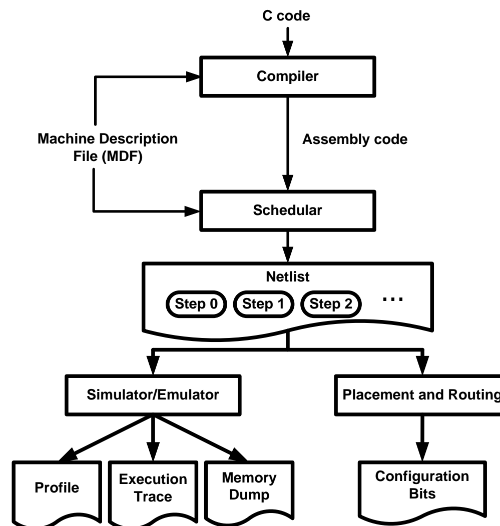

Figure 5-9 RICA architecture tool flow

arrays. Figure 5-10 demonstrates the three main operation modes of the processor. Figure 5-10 (a) represents the initial configuration mode, where the configuration controller is active in loading and configuring the cells and their interconnections, and the dark-coloured cells in this illustration represent active cells. Figure 5-10 (b) represents the main working operational mode of the processor, where actual RS encoding/decoding takes place. Here it can be noted that the input and output ports are active, indicating continuous data streams coming in to and going out from the processor. Finally, Figure 5-10 (c) shows the processor have completed the main function and now ready to be configured for the following operation, which could be another algorithm. Processors based on the RICA paradigm have a dedicated tool flow developed for them which comprises a compiler, scheduler, placement and routing, and emulator, as illustrated in Figure 5-11. The compiler transfers the high level C code into assembly language format, which is based on the instructions in the processor cells. Information for the processor cells is provided by the machine description file (MDF) which holds the functions and capabilities of those cells. The assembly file generated will be passed to the scheduler, which in turn will produce a netlist of a series of steps to configure the processor dynamically. The scheduler takes into account the resources from the MDF file, such as cells, interconnections and timing constants, in addition to the optimisation.

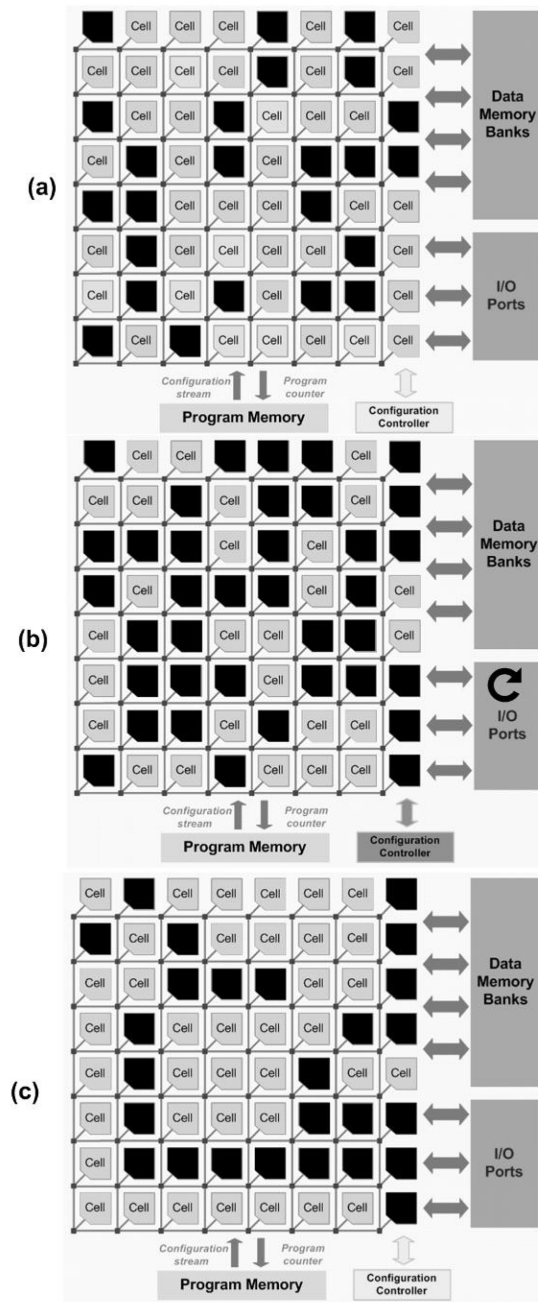

Figure 5-10 Programmable Reed Solomon processor architecture based on heterogeneous cell array (dark coloured cells represent active cells in a certain configuration). The three cases are: (a) initial configuration; (b) intermediate configuration, in which certain cells are configured to code/decode data from/to input/output ports; and (c) final configuration, flushing remaining data out, and preparing for the subsequent configuration.

5.6.2 RS Encoder Implementation and Optimisation

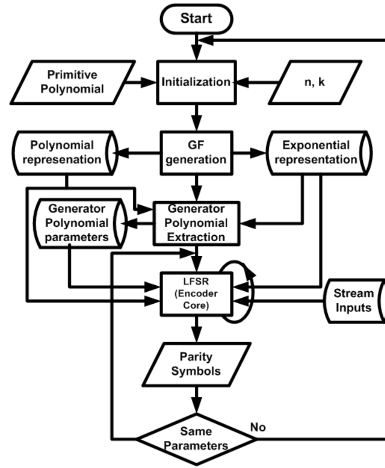

Figure 5-11 Reconfigurable RS encoder data-flow graph

For the efficient implementation of a real time RS encoder on the processor, the encoder has been implemented based on the data-flow graph presented in Figure 5-11. This implementation is based on generating the whole Galois Field required through its primitive polynomial, and records these in two separate arrays: one for polynomial representation and the other for exponential representation. The GF multiplication is carried out by calculating the exponentials of the two operands. The generator polynomial is generated and stored in an array, while the calculation of the encoder parity symbols takes place using the LFSR (linear feedback shift register), as demonstrated in Figure 5-2 (a). The throughput obtained in realising

Results before optimisation RS(255,239):

One time execution* of RS encoder = 335.83 msec
Step count = 13,751
Dynamic energy# = 5.67 mJ
Throughput = 6.1 Mbps

** One time execution = one complete coded block of data*
 # Represent all energy except interconnections energy

this approach is, however, far from the target of at least 70Mbps (which is the optimum requirement for WIMAX). This can be seen from the performance summary report below.

By analysing the resulting performance report and intermediate files, it is concluded that the obtained low throughput is due to the excessive use of read/write operations from/to data memory. In order to overcome this problem, several optimisation techniques, as mentioned earlier, were applied. The maximum throughput achieved then increased to 10.1Mbps, as can be seen from the report results below.

Results after optimization RS(255,239):

One time execution of RS encoder = 202.486 μsec
Step count = 6,042
Dynamic energy: 4.7 μJ
Throughput: 10.1 Mbps

The optimisation techniques applied resulted in an enhancement of 40% in the

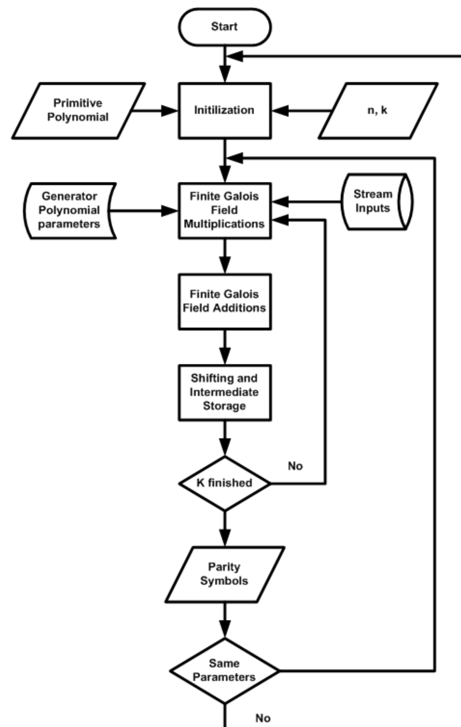

Figure 5-12 Reconfigurable RS encoder modified data-flow graph using GFMUL cell

throughput and a reduction 17% of in the dynamic energy. As the required throughput (70Mbps) was not reached, the algorithm has been further investigated, leading to the implementation of the algorithm using a hardware-inspired technique in order to reduce the memory access overhead.

Figure 5-12 presents the proposed modified data flow graph, which provides some advantages, such as greater reductions in the total number of operations and the number of memory accesses required, in addition to the elimination of the modulus cell/function. On the other hand, the modified flow-graph requires a new cell, the Galois Field multiplier (GFMUL), to be added to the processor. The cell developed supports run time reconfiguration for the GF primitive polynomials. This cell and its design have been discussed in detail in Section 5.5.

5.6.3 RS Decoder Implementation and Optimisation

The Reed-Solomon decoder architecture and the interaction of its various functions are illustrated in Figure 5-13. The decoder has been implemented based on the techniques of the data-flow graph shown in Figure 5-11, which has two arrays: the polynomial and exponential representations. The results obtained are similar to

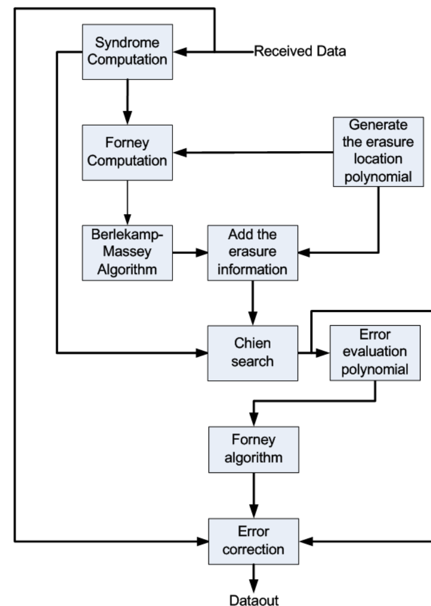

Figure 5-13. Reed-Solomon decoder algorithm design

those for the encoder and a follow-up implementation of the decoder was then conducted using the same techniques of the modified data-flow graph (Figure 5-12) with the aid of the GFMUL cell. Compared with the data-flow graph, the architecture with the GFMUL cell is less complex, and the large arrays required for storing the GF elements are no longer necessary. Considering the memory optimisation mentioned earlier, there would be four memory read/write operations per GF multiplication in the data-flow graph technique, as well as other operations such as modulo-2 adder, logic, modulus, and so on. By reducing the number of memory access operations and using the GF multiplier cell in the modified data flow graph approach, the performance improves significantly in increasing throughput and decreasing energy consumption. The usage of the GFMUL cell leads to a reduction in memory access energy of 72.4%, which represents a significant performance improvement. Parallelism is another important optimisation technique in maximising resource utilisation. Every functional block of the RS decoder has been examined to decide whether parallelism in the code can be utilised to increase performance. For example, syndrome computation requires 4080 8-bit adders and 4080 8-bit multipliers. As the elements of $S(x)$ are all independent, it can be computed with 16 parallel data paths with recursive multiplication and addition [76]. In addition, the performance of the parallel architecture can be further improved by employing the SIMD technique, as explained earlier. The architecture of the Forney computation's depends on the number of erasures in the received codeword. There are no efficient parallel architecture techniques that could be utilised for all cases of the Forney computation, because the number of additions and multiplications differ according to every symbol frame. The Berlekamp-Massey algorithm is based on iteration, and all calculations are computed in sequence depending on the number of errors and erasures, so that no parallel optimisation could be implemented. The Chien search is used to evaluate the error location polynomial with all 255 possible roots, so that parallel implementation is possible here since they are all independent. The error evaluation polynomial makes use of the syndrome and error polynomial. It has an architecture similar to that of syndrome computation, so that all of its 16 elements

could be computed in parallel in 16 paths, and in addition, the SIMD technique could be employed. The Forney algorithm determines the actual error and erasure value from the error location and value polynomials, and for the calculation of each symbol parallelism could be implemented. Error correction can be easily implemented in parallel simply with a GF adder (XOR - logic cell). With all the parallelism optimisations described, the processing time and memory access energy consumption are reduced further by nearly 25% and 23.3% respectively.

5.7 Performance, Comparison Analysis and Results

5.7.1 GF Multiplier Cell

The results in terms of area and calculation delay for the implemented GFMUL as either a single multiplier or the proposed novel concatenated four multipliers are presented in Table 5-2. The multiplier is combinatorial, hence its key parameter is delay time along with power consumption. In Table 5-2, two implementations have been highlighted: firstly a single 8-bit multiplier entitled the 'One programmable 8-bit GFMul cell;. and the second with four 8-bit multipliers entitled 'Four programmable 8-bit GFMul cell.' Both are programmable, and the results for the single multiplier have been listed to allow comparison with other studies while the cell with four multipliers is the main cell integrated within the processor. It is clear how the maximum delay has been kept to a minimum at 3.88 ns. These results were

Table 5-2 Implementation results for the GF multiplier cell
(180nm Technology)

$GF(2^8)$	**Area (μm^2)**	**Max delay (ns)**
One programmable 8-bit GFMul cell	6,265.40	3.34
Four programmable 8-bit GFMul cell	19,345.10	3.88

obtained post-layout and have been modelled within the RICA development environment in order to give accurate modelling of the RS coding. The GF multiplier has been developed as an extension of the instruction set for the Sandblaster Micro-architecture [69]. By comparing the results for the developed GFMUL with [69], the GFMUL multiplier proposed here achieves an area reduction of 45%. On the other hand, the Sandblaster multiplier delay is half of that obtained with the GFMUL; however, the GFMUL is programmable whilst the Sandblaster multiplier is not. The difference in delay is considered an acceptable overhead given the advantages in flexibility and does not affect the performance or functionality of the either multiplier or the whole processor. In addition, the 45% reduction in area will be accompanied by a reduction in power consumption, leading to the possibility of attaining an ultra-low-power processor design.

5.7.2 RS Codec Processor

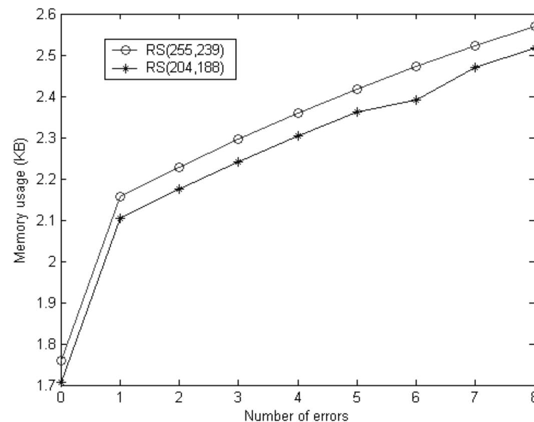

Figure 5-14 Memory usage for RS decoding on RS processor

The performance results for the processor are obtained from the RICA software tool flow based on the accurate modelling of a fabricated processor. For the RS (204,188), the encoder throughput reached 202 Mbps, while for the RS (255,239) the encoder throughput reached 200 Mbps. This is a significant improvement over the intermediate results reported earlier; moreover, it exceeds the application specifications. The following are the results for the RS encoder:

-Results after optimisation *RS(255,239)*:

Throughput: 199.9 Mbps

Memory usage = 359 bytes

-Results after optimisation *RS(204,188)*:

Throughput: 202.5 Mbps

Memory usage = 308 bytes

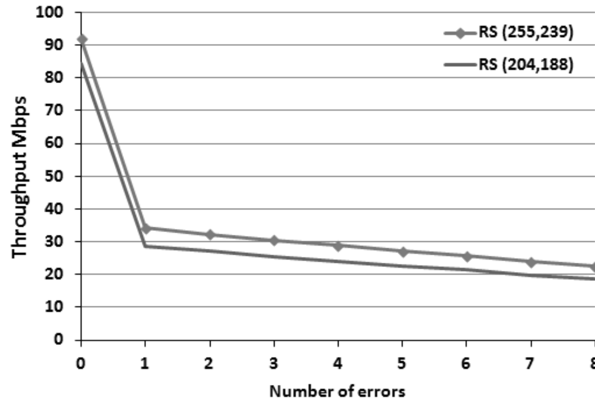

Figure 5-15 Throughput of RS decoding on RS processor

For the RS decoding, the results obtained after the series of optimisations can be represented graphically. Decoding results are usually represented by best, average and worst case results. The best case is when there are no errors, the average case is when there are four errors to be detected and corrected, and the worst case is when

Table 5-3 Performance comparison of the novel RS processor and StarCore 140 for RS(255,239)

	RS Encoder:				
Processor	**RS Processor***			**StarCore 140 [70]**	
Cycles count	257			6359	
Time (µs)	10.202			264.96	
Throughput (Mbps)	199.9			7.26	
	RS Decoder:				
Processor	**RS Processor***			**StarCore 140 [70]**	
	Best case	**Average case**	**Worst case**	**Average case**	**Worst case**
Cycles count	1423	3,245	4,463	14,298	14,428
Time (µs)	22.174	68.105	90.376	264.778	267.185
Throughput (Mbps)	73	29.95	22.57	7.70	7.64

**RS Processor chip is based on 180nm CMOS technology*

there are eight errors to be detected and corrected. Figure 5-14 illustrates the memory usage with different numbers of errors. The result show a considerable improvement of an 85.6% reduction in worst case for memory usage compared to previous work [70], and hence the memory access energy will be reduced. The decoder throughput with different numbers of errors is illustrated in Figure 5-15. The results show an overall advantage compared with [71] and moreover the proposed processor offers considerable lower energy consumption than FPGA implementations.

After applying all of the optimisation techniques discussed above in deploying the GFMul and applying the SIMD technique, the results were enhanced significantly. The encoder throughput increased from 10 to 200 Mbps, while decoder throughput from 2 to 92 Mbps in the 'best case' for the RS(255,239), where the best, average and worst cases are shown in Table 5-3. This represents a significant improvement, and these results also prove that the reconfigurable RS processor introduced here can accommodate the demanding standards and applications expected in the future.

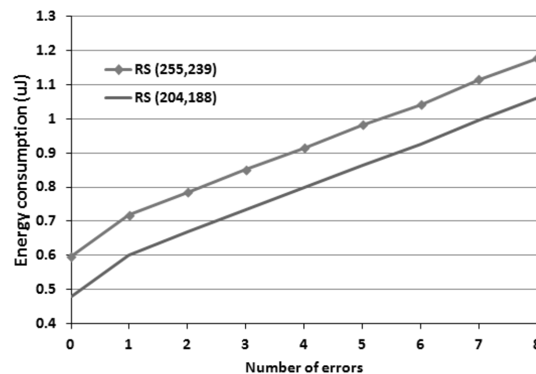

Figure 5-16 Energy consumption for the RS decoder on RICA

The methodology used for StarCore 140 RS encoder/decoder implementation [70] performs the additions in binary representation and multiplications in exponential representation, where conversions between the two representations are accomplished with the aid of look-up tables. Table 5-3 shows the superiority of the novel processor over the StarCore processor for all parameters. It is worth

mentioning that the StarCore is a dedicated industrial DSP for communication applications. The energy consumption is demonstrated in Figure 5-16 for the cases of both the RS(255,239) and RS(204,188). In [71] the maximum throughput achieved for the RS decoding is up to 15Mbps for RS(255,239), while in the proposed RS processor throughput reaches 92.5Mbps and moreover it offers considerable lower power consumption than the FPGA implementation.

5.8 Conclusion

A novel RS encoder architecture with parallel parity output has been introduced in this chapter. A novel high-speed and low-power 32-bit GF multiplier cell embedded within the novel low power processor for programmable Reed Solomon coding has been introduced along with its design, optimisation and implementation. The real-time programmable RS encoder and decoder processor supports several communication standards such as WiMAX and DVB-H. In addition, the processor can be used in a wireless local area network IEEE 802.11 to improve its range [67]. There are other possible applications for RS codes, such as deep space communication [68]. A number of approaches and optimisation techniques have been implemented in order to enhance the performance of the processor. The processor achieves high throughput and provides significant improvements in performance and energy consumption.

The GF multiplier cell leads to a reduction of 72.4% in memory access energy, which improves the processor's performance. Different design approaches and optimisation techniques have been applied in order to enhance the processor's throughput and to reduce its energy consumption. The throughput achieved is up to 200 Mbps and 92 Mbps for the encoder and decoder respectively. Associated dynamic energy consumption is in the range of 0.34 to 0.6 µJ, leading to the conclusion that this is a design suitable for present and future handheld devices.

Chapter 6

GPS DIGITAL MATCHED FILTERS USING DYNAMICALLY RECONFIGURABLE ARCHITECTURE

6.1 INTRODUCTION

The first GPS (global positioning system) was declared fully operational in 1995, with 24 satellites in orbit. Its importance to civilian users was recognised immediately and in 2000 the 'selective availability' function was discontinued, allowing users to receive non-degraded signals. This accelerated the system's adoption in civilian applications on land, in the sea and in the air and led to a revolution in personal navigation devices. Today, GPS applications are embedded in various gadgets such as mobile phones. This has allowed an extended range of applications ranging from sat-nav devices to social networking applications.

The GPS utilises a constellation of satellites in medium Earth orbit which provide positioning, navigation and timing information to compatible receivers on Earth [83]. Signal acquisition or correlation is by far the most computationally demanding module of a GPS receiver. In addition the correlator has strict time constraints and should be able to continuously execute the real-time correlation process. Thus the correlator is a key to achieving low-power GPS receivers [84].

This research work focuses on developing a novel correlation engine through the design and implementation of various GPS correlation architectures exploiting the RICA processing paradigm. Performance evaluations are conducted in terms of time, energy and memory usage.

Two main types of correlator architecture have been introduced in the literature. The first performs the direct correlation of the locally generated code replica with the sequence received in the time-domain. The second utilises frequency-domain techniques relying on the DFT (discrete Fourier transform) and FFT (fast Fourier transform). The FFT can reduce the computational complexity of the correlator, but still requires complex algorithms which are not easy to implement and are power-hungry. Furthermore, this approach approximates some calculations to reduce the overall noise tolerance of the correlator. For these reasons, the present research focuses exclusively on time-domain architectures.

This chapter is organised as follows: correlation architectures are presented in section 6.2, while section 6.3 describes the engine architecture developed in this work. Section 6.4 details various matched filter algorithms and their implementation on the architecture and the optimised correlator engine designs. The results are analysed in section 6.5 and conclusions are presented in section 6.6.

6.2 Correlation Architectures

A matching filter is an important basic building block in wireless communication systems such as the GPS, WLAN, CDMA and WiMAX. It is used in signal acquisition and tracking, and requires a significant amount of system resources. The design of the matching filter is a crucial factor in system performance in terms of rapid signal acquisition and tolerance of interference. Acquisition is the most time-consuming function of a GPS receiver and various acquisition algorithms have been developed in order to speed up computation [84]. A number of algorithms can be used for correlation. As mentioned earlier, the focus here is on time domain correlation, of which four main types are discussed.

The correlation algorithm multiplies the incoming signal with all possible coarse/acquisition code (C/A code) combinations and then integrates and sums the results. This means that the receiver is not affected by possible phase differences between the locally generated and the actual carrier frequencies. The correct phase

is found once the correlator output exceeds a predefined threshold. These parameters can then be passed on to the tracking circuit.

The GPS C/A code sequences belong to a family of pseudo-random noise (PRN) codes discovered by R. Gold in 1967 [85]. They are also known as ‘Gold’ codes and their most important characteristic is their correlational properties. Cross-correlating two different 1023-bit Gold codes (C_1, C_2) can be represented as in Equation (6-1).

$$R(k) = \sum_{i=0}^{1022} C_1(i) \cdot C_2(i+k) \approx 0 \tag{6-1}$$

where k is the phase-shift of C_2 relative to C_1. It is known that the C/A codes are almost uncorrelated with each other at any phase difference. If C_1=C_2, then the auto-correlation result of the above equation reaches a peak value of 1023, when C_1 and C_2 have the same phase (k=0).

The signal received by the RF front-end and quantised by the A/D converter is a combination of the signals transmitted by all visible satellites. If N satellites are visible at a specific moment, then the received signal $S(t)$ is the summation of all visible satellite signals as in Equation (6-2).

$$S(t) = S_1(t) + S_2(t) + \ldots + S_N(t) \tag{6-2}$$

The acquisition algorithm must identify whether or not a specific satellite is currently visible and find the carrier frequency of the signal and phase of the C/A code. The carrier frequency of the transmitted signal is already known, but since the satellite is continuously moving, and possibly the receiver is too, the carrier frequency of the received signal will differ from its nominal value by a small Doppler shift. Furthermore, the code phase will be random and the purpose of acquisition is to locate the beginning of the C/A code in the received signal.

Each GPS satellite is located its own unique code sequence. The C/A codes used are a chain of 1,023 bits within a period of 1ms. The GPS receivers generate the same code sequences internally. These are compared with the received signal from

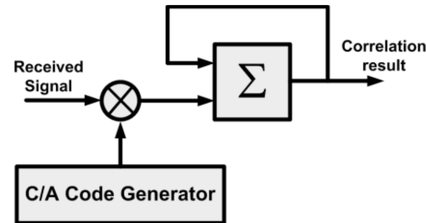

Figure 6-1 Serial search correlation

the satellites. If correlation is not achieved, then the local code will be shifted by one bit and compared again. This process is repeated until a match occurs, while the number of shifts will be represented as a delay value to be used in further calculations in the GPS receiver. If all 1,023 bits have been tried without successful correlation, then an offset to the phase value need to take place in the receiver using the frequency oscillator, and the previous process has to be repeated. The next section discusses various matched filter algorithms and their adoption for GPS correlation in the proposed processing engine.

6.2.1 Serial Search Correlation

The simplest and most conventional algorithm to implement is the serial search correlator. Figure 6-1 presents its architecture. It is used in the commercial Zarlink's GP2021 correlator implementation [86], and is a relatively simple algorithm to implement which has minimal hardware requirements. Its main drawback is the extensive period required to perform acquisition. The signal samples are multiplied consecutively with the locally generated C/A code samples and then the results are integrated.

It can be seen that the only hardware parts required are a multiplier, an adder and an accumulation register. To perform complete acquisition using this architecture would require 1,023 multiplications and 1,022 additions for each phase, since the length of a full C/A code is 1,023 bits. Therefore, for the 1,023 phases, it would require in total:

$1{,}023 \times 1{,}023 = 1{,}046{,}529$ multiplications and $1{,}023 \times 1{,}022 = 1{,}045{,}506$ additions.

In the GPS, each complete C/A code has a period of 1 ms (1 × 1.023MHz), and hence the serial search correlator requires 1.023 seconds to complete the acquisition of 1,023 phases.

6.2.2 Conventional Digital Matched Filter

The conventional digital matched filter (CDMF) correlator presented in Figure 6-2 uses a tapped delay line to correlate the incoming signal with the locally generated code replica [87]. Its main advantage is that a full 1023 phase search can be completed in the time required for one 1023-bit C/A code sequence to be received (1 ms). On the other hand, it requires an additional 1023-register buffer to store the incoming signal samples. The buffer must be initially filled before any correlation results can be obtained, so a further 1 ms should be added to the total acquisition period. Its computational complexity is the same as that of the serial search method, as it requires the same number of additions/multiplications to complete a 1,023-phase acquisition.

The algorithm operates by storing successive incoming signal samples in the shift registers (buffer), and calculates the correlation of phase *i* when the buffer becomes full. As a new data sample is received, it is shifted-in the buffer while at the same time the oldest stored sample is shifted-out. Then the correlation of phase *i+1* is calculated. Since the majority of incoming data samples are kept in memory, the

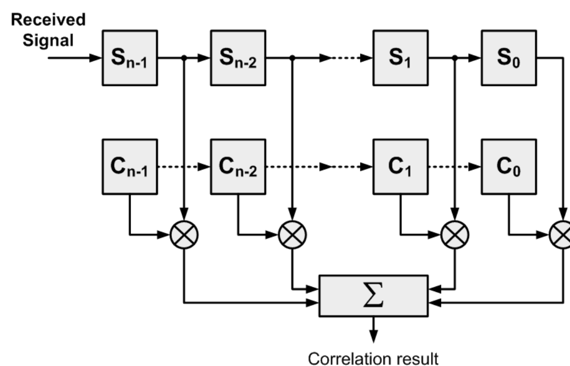

Figure 6-2 Conventional digital matched filter (CDMF)

correlation results for subsequent code phases can be calculated quickly, requiring 977 ns for each phase.

6.2.3 Differential Digital Matched Filter

A differential digital matched filter (DDMF) correlator improves on the CDMF by eliminating most of the multiplications and reducing the number of additions by 1/2 [88] and [89].

The sample values of the local code replica will be either +1 or -1. If these values are incremented by +1, then they will become +2 or 0. Statistically, approximately half of the code values will be -1 (or 0 after incrementing); so their multiplication and addition can be eliminated.

After adding +1 to the values of the code replica, their expression becomes $M_n = C_n + 1$ and therefore the correlation result will be:

$$Result(i) = M_{n-1}S_{n-1} + M_{n-2}S_{n-2} + \cdots + M_1S_1 + M_0S_0 - Sum(i) \quad (6\text{-}3)$$

where $Sum(i) = \sum_{L=0}^{n-1} S_L(i) = Sum(i-1) - S_{i-1} + S_{n-1+i}$

$$Sum(i) = \sum_{L=0}^{N-1} S_L(i) = Sum(i-1) - S_{i-2} + S_{n-1+i} \quad (6\text{-}4)$$

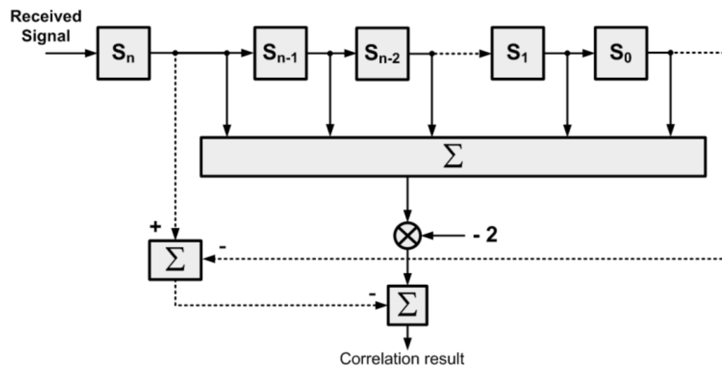

Figure 6-3 Differential digital matched filter (DDMF)

From Equation (6-4) *Sum(i)* is the sum of all of the values stored in the buffer-register *S* in the current phase iteration *i*. It can be calculated by adding the most recent signal value shifted-in to *Sum(i-1)* and subtracting the most recent signal value shifted-out as illustrated in Figure 6-3. Thereafter, the signal samples corresponding to the non-zero values of the code replica need only to be multiplied once by 2 after they have been summed together. Thus the DDMF effectively cuts in half the computational load compared to the CDMF algorithm.

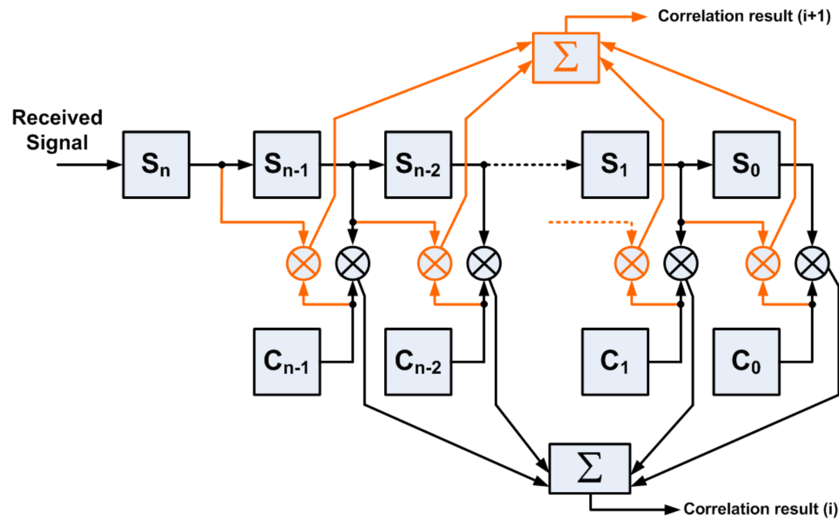

Figure 6-4 Segment processing digital matched filter (SPDMF) with *K*=2

6.2.4 Segment Processing Digital Matched Filter

The segment processing digital matched filter (SPDMF) algorithm optimises the DDMF by eliminating some of its computational redundancy [90]. Its architecture is illustrated in Figure 6-4. By examining the correlation results of two consecutive iterations *i* and *i+1* using the DDMF algorithm, $Sum_{00}(i)$ is the sum of all the *S* values that are multiplied by '0' on the first *(i)* and second *(i+1)* iterations, while $Sum_{01}(i)$ is the sum of *S* values that are multiplied by '0' on the first and '1' on the second iteration, and so on. The advantage of this method is that $Sum_{01}(i)$, $Sum_{10}(i)$,

$Sum_{11}(i)$ only need to be calculated once, and can then be use to produce two consecutive correlation results. Furthermore, the limitation processing the correlation results only in groups of two no longer applies, and it can be increased. It has been shown [90] that the algorithm reaches its maximum performance when groups of 8 or 9 are used, in which case the SPDMF becomes 3 times more efficient than DDMF. But the benefits should be noticeable even for groups of 3 phases. While this method effectively eliminates some computational redundancy, additional hardware is required in the form of extra registers and controls.

6.2.5 Algorithmic Comparison

The computational requirements of the algorithms mentioned above are summarised in Table 6-1. In all cases it is assumed that acquisition is performed on 1 ms or 1023 bits of C/A input signal samples, searching for 1023 code-phases. Furthermore, the SPDMF is assumed to process data in groups of 3 (K=3). All the equations used for calculating computational complexity can be found elsewhere [90]. Clearly the SPDMF algorithm, with just 357,027 additions needed to complete acquisition, compares favourably with the other architectures.

6.3 Engine Architecture

The processor proposed in this research is based on the reconfigurable instruction cell array (RICA) paradigm. The RICA paradigm is based on a variable set of

Table 6-1 Comparison of theoretical computational complexity for various correlation algorithms

Correlation algorithm	Additions	Multiplications	Time required to complete acquisition
Serial Search	1,045,506	1,046,529	1,023ms
CDMF	1,045,506	1,046,529	1ms*
DDMF	525,311	1	1ms*
SPDMF(*K*=3)	357,027	1	1ms*

**There is an additional 1ms for register preloading*

heterogeneous cells connected through a reconfigurable interconnection network [35] and [91]-[92]. In general, the cells perform primitive operations, such as addition, multiplication, shifting, logic, and multiplexing, and in addition other cells control handling and branch operations. The contribution and number of cells could be tailored depending on the application or set of applications required to run on the processor engine.

In order to have a specific engine for processing GPS correlations in general, several aspects should be considered. These include the exploitation of various correlation algorithms, the analysis of their requirements and subsequent optimisations. The engine's prime design consideration is minimum energy consumption, while still meeting the correlation requirements of the GPS in terms of processing capability and time constraints.

A correlation engine is proposed in this study based on the RICA paradigm. The engine inherits various capabilities and advantages that are characteristic of the architecture [90]. The engine is digital signal processor (DSP)-like, hence retaining the advantage of the DSP in terms of high level programmability. In addition, the engine has a reconfigurable data-path which implies that it does not have fixed cycles, but is based on a 'step' concept. A 'step' is a combination of instructions that run simultaneously on a single configuration, or physically a single data-path interconnection configuration for a group of cells in the processor. Steps are determined by the resources available in the processor, conditional branches (jumps) and the critical path length. The step concept allows the maximum exploitation of correlation parallelism. In general, the higher the utilisation of architectural cells, the lower the number of steps required for the application and vice versa. In addition, optimal performance is usually obtained through reducing the number of steps, and this is achieved by applying various optimisations to the correlation algorithms or the engine architecture. The engine has been designed in order to be capable of efficiently processing the correlation algorithms mentioned earlier. After analysis of the above correlation techniques, it was decided that the engine design would include the following cells: 64 adders, 13 logics, 20 shifts and 32 constants, this is in addition to the engine memory and control cells which are

responsible for the dynamic configuration aspects of the processor and managing the execution of ‘steps’.

6.4 Analysis and Optimisations

The design, optimisation and analysis in this work was carried out in three tiers. The first is based on implementing various correlation algorithms on a first processing engine. The types and quantities of engine cells have been optimised in order to save power consumption while a sufficient degree of parallelism is exploited for processing the various correlation techniques. It is worth mentioning here that the correlation algorithms and techniques are optimised to fit the specific engine architecture. In the second tier further optimisations are applied to the correlation algorithms based on the findings from the first tier. In the third tier the engine architecture is optimised in order to obtain optimal performance and the lowest energy consumption.

The aim of optimisation is to achieve minimum engine size and maximum engine utilisation for the target requirements. This can guarantee the achievement of lower energy consumption with the efficient implementation of the algorithm.

6.4.1 Tier 1: Correlation Implementation

Here the correlation techniques and algorithms mentioned earlier are implemented on the proposed novel correlation engine.

1) *Serial search correlator implementation*

A serial search correlator has been designed, programmed in the C-language and implemented on the correlation engine. Several optimisation techniques are applied in order to exploit the resources of the processor and to extract the algorithm parallelism. Such optimisations include loop unrolling and step size maximisation to increase the degree of parallel execution by the engine.

The correlation time obtained for a complete search process is equal to 94ms for 1023 phases, with each phase consisting of 1023 samples. This implies that the time required for each frame is equal to 91.88 μs, while the real-time requirement is 1ms. Due to the serial nature is of this engine and the fact that it is fully dependent on the received signal speed, it is impossible to achieve any improvements beyond the 1023 ms required for a full correlation. This means that the engine satisfies the algorithm's requirements in theory; however, it is slow and time consuming to wait for 1.023 s to fix a single satellite signal. Present and future applications require GPS receivers with quicker correlations and shorter times to the first fix.

2) Conventional digital matched filter implementation

The CDMF is a parallel search correlator that requires shift registers. After applying similar optimisations to the serial search, a total correlation time of 88.6 ms has obtained. By analysing the generated assembly codes, intermediate files, and resulting kernels, it was apparent that the implementation of the algorithm resulted in the excessive use of the processor memory, which led to increased numbers of steps. It is worth mentioning that there is a restriction on the usage of the processor memory to a maximum of four memory read or write operations in any single 'step'. This means that, with excessive memory usage, the number of steps will increase significantly. This leads to a reduction in the processor's utilisation of resources and increases the correlation time.

In order to enhance these results, a number of techniques were exploited. These include the merging of some of the steps and a reduction in memory access by maximising the usage of the local registers available in the processor. In addition, the C/A replica code required in the correlation process is generated in real-time by a C/A code generator embedded in the processor; here, an experimental procedure was conducted in order to have the C/A replica code hard-coded. The correlation time achieved after optimisation with hard-coded C/A code is 47.8 ms, and 29 ms with the C/A code generator. In general, the hard-coded technique provides better performance; however, in this case it incurs a penalty through increasing memory access activity in the processor. This is due to the location of hard coded data being in the processer memory which must be recalled back whenever required. On the

other hand, the generator inputs results directly into the processor registers without any interaction with the processor memory. The resulting architecture is approximately 3 times faster than that with the serial search, but is still far from meeting the requirements of the system.

3) Differential digital matched filter implementation

Using the same optimisation techniques described earlier, the correlation time achieved is 13.6ms. Furthermore, it is clear that the DDMF correlator algorithm outperforms the CDMF. Moreover, in implementation and after further optimisation, the DDMF outperforms CDMF to give a reduction in the correlation time of almost 50%. However, this is still far from achieving the requirement of 1ms or less.

4) *Segment processing digital matched filter implementation*

The SPDMF correlator with K=3 has been designed and implemented on the selected engine after similar optimisations as mentioned earlier. The result achieved is 57.6 ms. The SPDMF algorithm outperforms CDMF and DDMF in terms of computational complexity, while it lags behind in terms of implementation. The main reason for this result, almost double the CDMF timing is that the control part of the architecture has been increased in size in order to cater for the three simultaneous correlation processes.

6.4.2 Tier 2: Architecture Optimisations

After analysing the results obtained in Tier 1, and further exploring the architecture, it is clear that better results are required in order to overcome the time constraints. Modification to the correlation algorithms is considered in order to fit there optimally on the processor. A modified a modified architectural design for the digital matched filter algorithms mentioned earlier is proposed. The design is based on minimising memory access the switching activity via applying the circular buffering techniques by eliminating shifting the shifting of the data received [93]. This will take place by allocating sufficient memory locations for in the processor

memory two complete data frames (blocks) of total size *2n*. In addition, a unit is introduced to distribut the incoming signals to their correct fixed locations in the memory. In the following subsections, the proposed optimised designs are introduced along with the results associated with their implementation on the engine.

1) *Modified Conventional Digital Matched Filter*

The modified CDMF design is shown in Figure 6-5. As mentioned earlier, the allocated memory size is 2*n,* from M_0 to M_{2n-1}. A distribution switch is used to distribute/write the incoming signals into the correct memory locations. Then there is the slider control, which is linked to the distribution switch in order to continuously adapt to the addresses of the most recently received signals. After applying the same optimisation techniques mentioned earlier, the correlation time was reduced to 14.8 ms, representing a significant improvement. The MCDMF achieved a reduction of 51% in the correlation time compared to CDMF. Although the MCDMF architecture is designed to target the specific proposed engine, it could also reduce power consumption in ASIC or FPGA implementations.

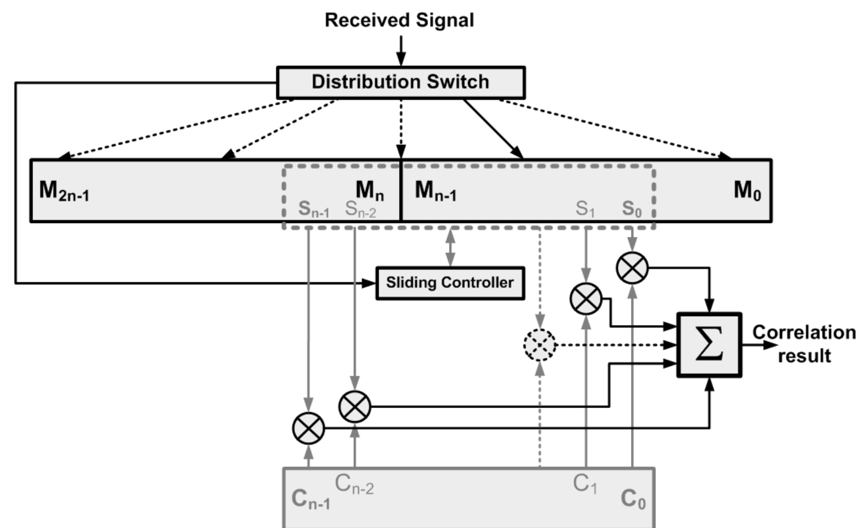

Figure 6-5 Modified conventional digital matched filter (MCDMF) architecture

2) *Modified differential digital matched filter:* The MDDMF architecture design is illustrated in Figure 6-6. One extra unit has been added, which is the '*C* value checker' (not shown in the diagram). If *C/A code*=0, then all subsequent operations will be eliminated (bypassed) in order to reduce computation, and hence reducing the correlation time. The MDDMF achieves time of 9.7ms, which represents a 29% reduction in correlation time compared to the DDMF.

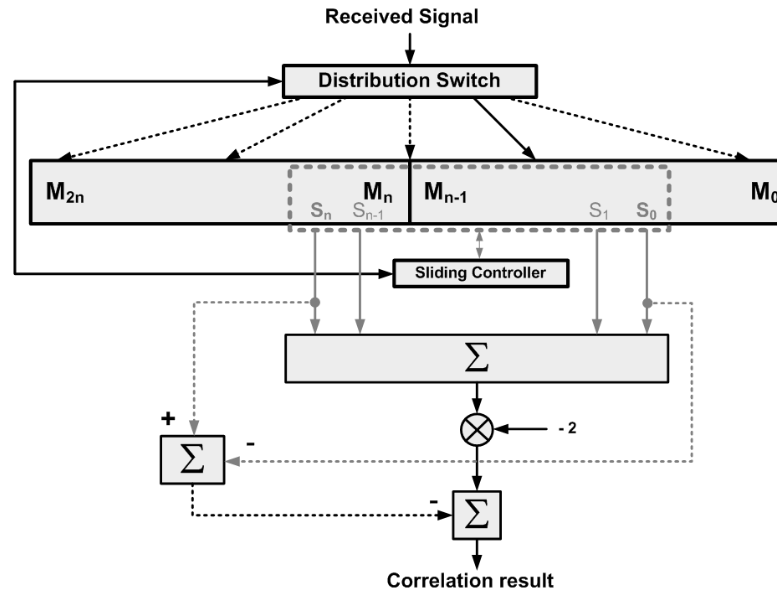

Figure 6-6 Modified differential digital matched filter (MDDMF) architecture

3) *Modified segment processing digital matched filter:*

The MSPDMF has been implemented for *K*=3 in a similar fashion to the SPDMF in Section 6.4.1. Figure 6-7 illustrates the MSPDMF with *K*=2, which reduces the diagram's complexity and improves clarity. The resulting correlation time is 36.6ms, representing a reduction of 36% compared to the SPDMF.

6.4.3 Tier 3: Engine Optimisation

Despite the use of the various optimisation techniques discussed and applied earlier and the novel correlation architectures introduced, the engine utilisation did not

increase to such high values as expected. The continuous read and write operations from and to the memory is the main reason for this. The engine has a limitation of using up to 4 simultaneous read or write operation per step, each being 32-bit. The processor is also a 32-bit-based as mentioned in previous chapters. Hence not all of the 32-bits in each variable, register and memory location were fully utilised in previous implementations.

The utilisation could be improved through applying vector operations that are supported within each of the architecture's cells. In addition, by using logic and shift operations, various information bits can be packed together and unpacked. This would add extra bitwise functions and operations such as logic, shift and compare to the design. In this research work, this optimisation process is termed bitwise optimisation. One of the optimisation techniques applied is use the XNOR to carry out multiplication operations instead of multipliers cells.

Furthermore, calculations have shown that a hard-coded technique will be beneficial in the bitwise approach. If the C/A code generated is stored in the memory through continuous write operations, then another readout operation of the

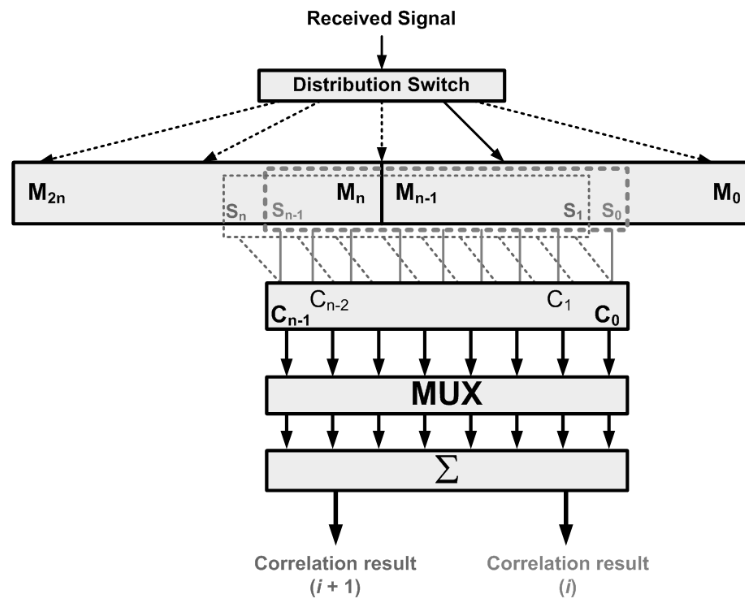

Figure 6-7. Modified segment processing digital matched filter (MSPDMF) architecture for K=2

1023 codes needs to take place afterwards to pack the code bits together. This will introduce enormous overheads. The hard-coded C/A code replica is completely packed offline, which will clearly enhance performance. Applying this approach would result in great reductions to the size of the memory locations required by the MCDMF correlator, and hence the numbers of read/write operations will be dramatically reduced, which will enhance the correlation speed.

In order to further reduce the correlation time, a new technique has been implemented which will maximise the utilisation of processor cells. This is achieved by maximising the usage of local register cells (REG) which hold intermediate and temporary data. In order to facilitate this, the C/A packed codes located in the memory will be read once and moved to the local registers at the start of the correlation program. This will reduce the total number of steps required and hence will speed up the correlation process.

By applying all of the above mentioned optimisation techniques, the MCDMF;s 1023 correlations were processed in only 0.3 ms which is a great improvement to a level far below the 1 ms constraint. From the analysis of the resulting assembly code and other intermediate files, further modifications were still possible, however. These include the need to increase the number of specific cell types. Hence, the number of adders has increased from 64 to 95, logic cells from 13 to 33, shift operations from 20 to 64 and constant cells (REG and CONST) from 32 to 81. The latest modifications have resulted in the 1023 correlations taking place in only 100µs which represents a 5-fold improvement. Details of these results and the associated performance levels are presented in section 6.5.

6.5 Performance Analysis

In this section, comparisons are presented of all of the architectures implemented. To give fair comparisons, the same input signals and C/A codes were used with all engines and architectures, and the architectures have been separated into two classes: those without packed data; and those with packed data, in which bitwise optimisation was applied.

Table 6-2 Comparison of results for digital matched filter correlator architectures (without packed data)

	CDMF	M-CDMF	DDMF	M-DDMF	SPDMF	M-SPDMF
Correlation time (ms)	29.05	14.80	13.63	11.77	57.67	36.61
Data memory access energy (μJ)	27.22	13.63	13.59	2.75	19.84	5.75
Program memory access energy (μJ)	208.50	191.21	5.62	8.62	28.84	19.78
Cells dynamic energy (μJ)	32.14	16.37	15.08	13.02	63.82	40.51
Dynamic energy*	306.45	245.09	53.13	33.22	145.43	98.74
Step count	1,097,079	573,308	732,778	300,332	2,180,029	991,648
Memory usage (Bytes)	3,163	3,203	2,076	3,433	5,425	5,445

* Interconnections energy is not included

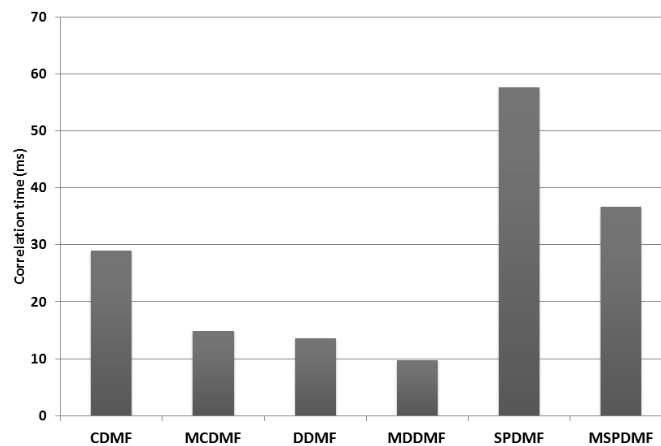

Figure 6-8 Comparison of resulting Correlation times (ms) for the matching filter architectures

Table 6-2 presents the data obtained from the various designed and simulated correlation architectures. Figure 6-8 presents a chart summarizing the correlation times obtained for various correlator algorithms and architectures presented in this work. It is clear that the novel ‘Modified’ architectures has accomplished better correlation processing times than the others. From Figure 6-8, it is clear that the MDDMF provides the fastest correlation, even though it did not reach the target of 1 ms. The dynamic energy consumption of the cells which is a major factor in the processor’s overall energy consumption, is illustrated in Figure 6-9. The MSPDMF achieved the lowest energy consumption for data memory access, while dynamic energy consumption of the DDMF cells was the lowest.

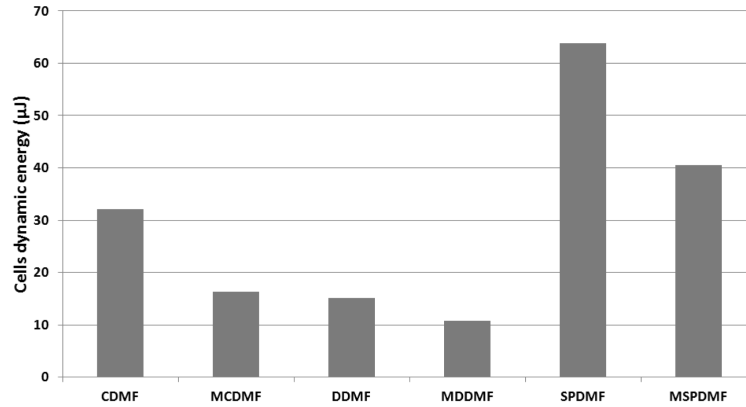

Figure 6-9 Comparison of cell dynamic energy (μJ) for various matched filter architectures

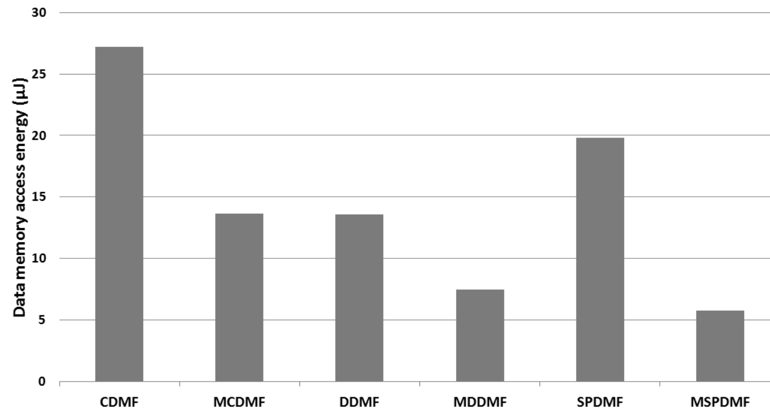

Figure 6-10 Comparison of data memory access energy (μJ) for various matched filter architectures

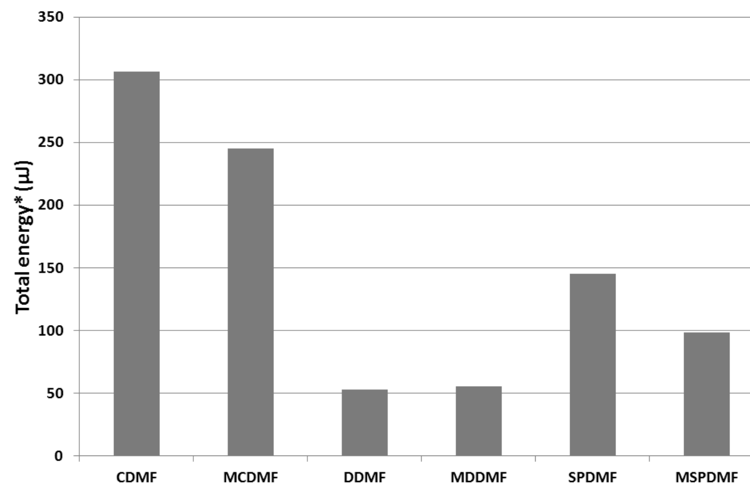

Figure 6-11 Comparison of total energy* (μJ) for various matched filter architectures
interconnections energy is not included

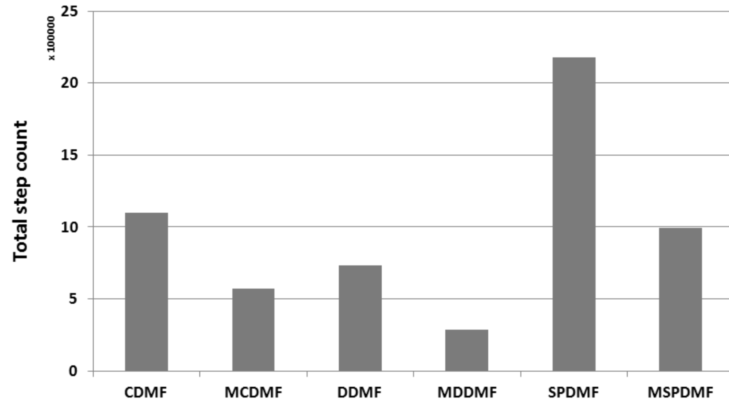

Figure 6-12 Comparison of total step count for various matched filter architectures

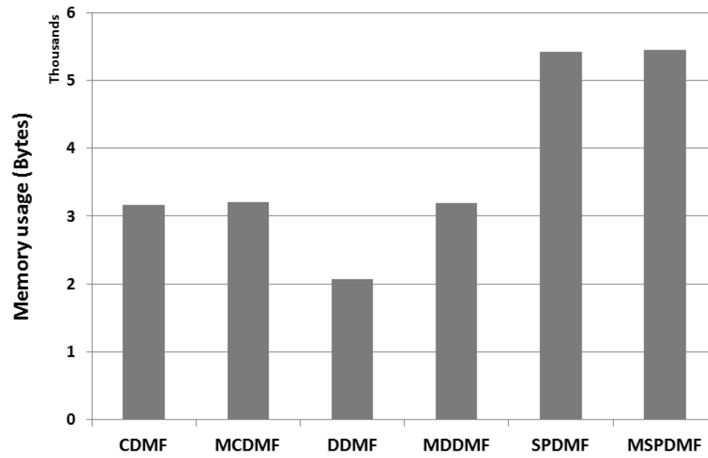

Figure 6-13 Comparison of memory usage (bytes) for various matched filter architectures

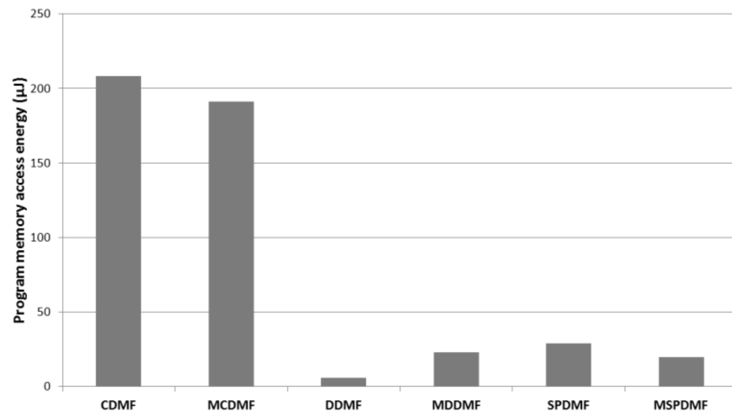

Figure 6-14 Comparison of program memory access energy (µJ) for various matched filter architectures

Table 6-3 Comparison of results for digital matched filter correlator architectures (with packed data)

	CDMF	M-CDMF	DDMF	M-DDMF	SPDMF	M-SPDMF
Correlation time (μs)	62.8	62.31	69.47	67.77	76.14	73.23
Data memory access energy (nJ)	6.40	4.91	9.93	5.34	13.47	5.78
Program memory access energy (nJ)	2.79	1.81	5.91	2.51	9.03	3.21
Cells dynamic energy (μJ)	1.09	0.57	2.95	0.29	4.81	0.67
Dynamic energy* (μJ)	1.21	0.65	3.25	0.71	5.29	0.77
Step count	1087	1074	1121	1096	1155	1118
memory usage (Bytes)	568	444	876	696	1184	948

* Interconnections energy is not included

Furthermore, the MCDMF has been implemented with a data packing approach (bitwise optimisation). The engine was modified by changing the types and numbers of embedded cells. The optimised architecture has accomplished the correlation process in only 62 μs. This is a considerable improvment over the 11 ms reached earlier and presented in Table 6-2. This is due to the increased number of cells introduced, accompanied by achieving the maximum optimisation of the processor. This has been accomplished by embedding the correlation process in a single 'step'. The complete process includes the initial configuration of the cells and their connections, which is then followed by the correlation step that will be rerun through a loop by changing the parameters every jump. The full results are

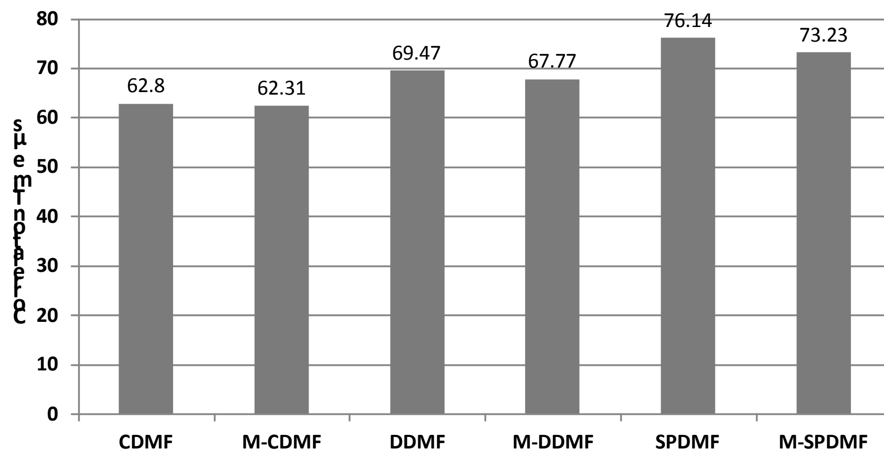

Figure 6-15 Comparison of correlation time (μs) with packed data results for matching filter architectures

listed in Table 6-3.

This eliminates the need for further architectural reconfiguration during the correlation process. In the dynamic reconfigurable architecture, this is the maximum optimisation that could be achieved. Figure 6-15 illustrates the correlation times obtained for all the correlation architectures after data packing optimisation/technique. It is clear that the MCDMF obtained the best results of 62.31µs, the fastest correlation execution time. This is mainly due to the fact that,

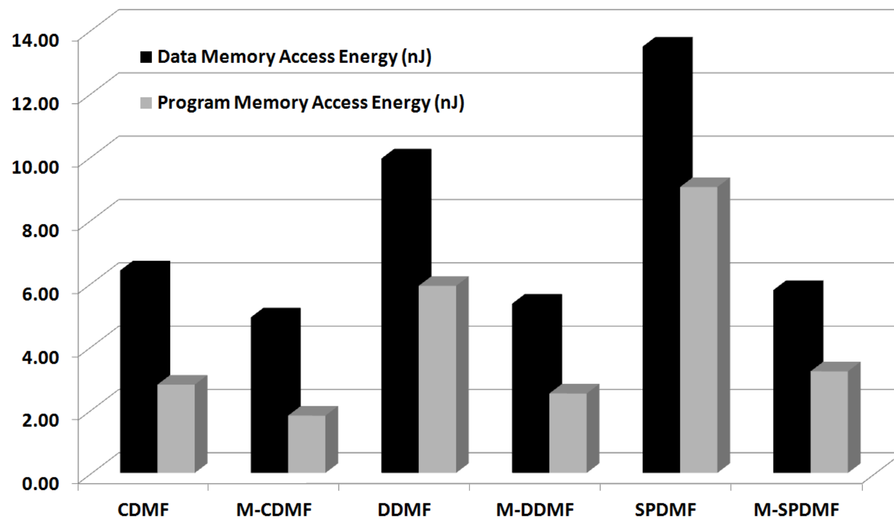

Figure 6-16 Comparison of data and Program memory access energy (µJ) for various matched filter architectures with packed data optimisation

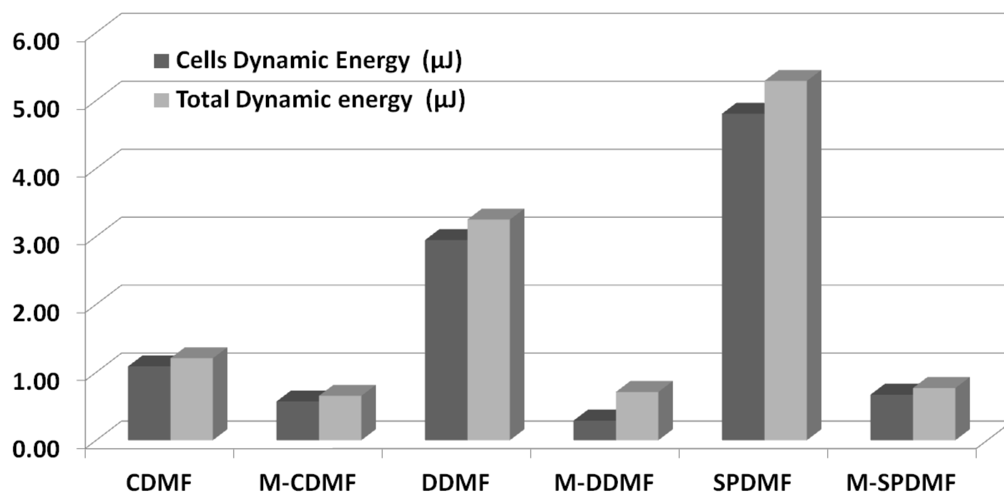

Figure 6-17 Comparison of cells and total dynamic energy (µJ) for various matched filter architectures with packed data optimisation

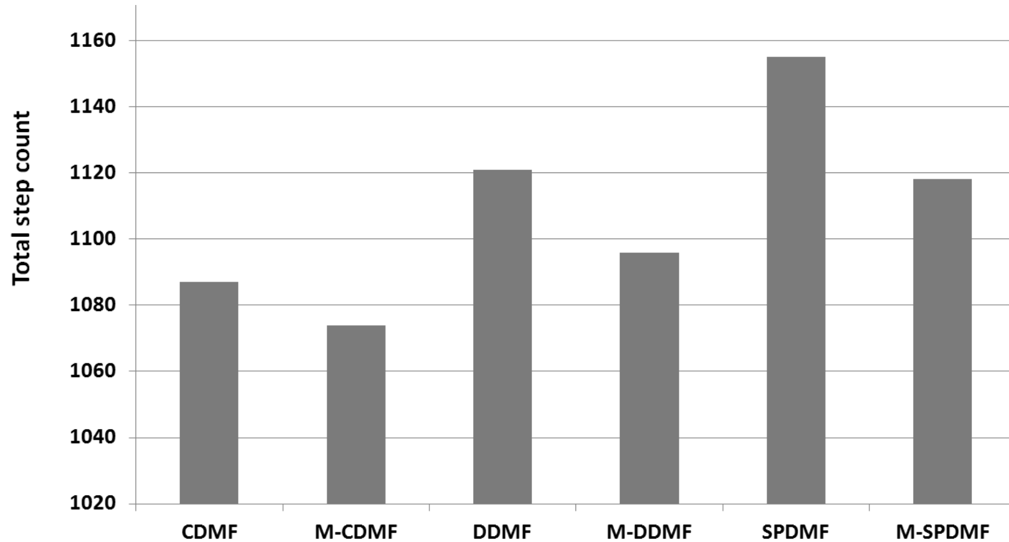

Figure 6-18 Comparison of total step count for various matched filter architectures with packed data optimisation

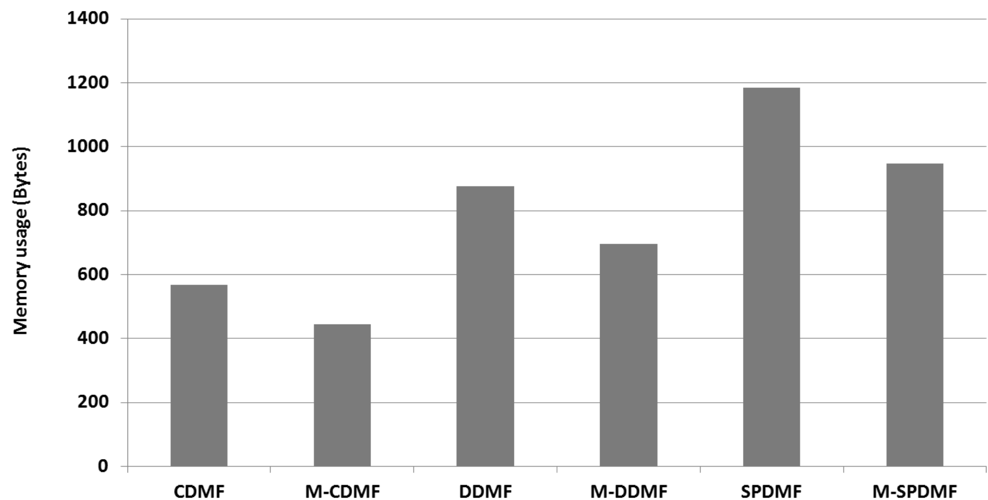

Figure 6-19 Comparison of memory usage (bytes) for various matched filter architectures with packed data optimisation

at this stage, when the whole algorithm is capsulated in a single configuration, all other architectures have overheads that are supposed to optimise the correlation. Moreover, since the data is packed, any processing of it will result in overheads, as reflected in the superiority of the M-CDMF over the other architectures.

Since the correlation time has been massively reduced, all the associated improvements follow suit and, in particular, energy consumption. Figure 6-16 illustrates the memory access energy for data and program, while Figure 6-17

illustrates the full dynamic energy consumption of all of the architectures. Finally, the step count and memory usage are illustrated in Figure 6-18 and Figure 6-19 respectively.

6.6 Conclusion

In this work, a novel correlation engine has been presented which is based on a dynamically programmable platform targeting the computationally intensive correlation function used in GPS-based positioning. Various optimisation techniques have been exploited in order to achieve the best performance on the platform. In addition, the modified correlation architectures MCDMF, MDDMF and MSPDMF have been introduced. They have demonstrated efficiency in terms of correlation time and energy consumption. Furthermore, the bitwise optimisation technique has been applied to digital matched filters, which demonstrate the maximum utilisation of the architecture leading to high correlation speed of 62 μs for 1023 phase search correlations. Comparisons of the achieved results and associated architectural configurations have been presented. This work represents a promising step towards high speed, ultra-low-energy GPS receivers.

Chapter 7

GPS MULTI-CHANNEL CORRELATION USING THE DYNAMICALLY RECONFIGURABLE PLATFORM

7.1 Introduction

Modern GPS correlators or correlation processors which are based on fixed architectures are capable of handling, on average, a 12-channel parallel correlation. In Chapter 6 a novel GPS correlation engine has been introduced. Its structure is based on a dynamically reconfigurable platform, and it has been concluded that the engine is capable of executing a complete GPS single channel correlation in 62 μs.

This chapter focuses on studying the capability of the designed engine for handling multi-channel correlations, and whether or not there will be a need to modify the engine design.

This chapter is organised into seven sections. Section 7.2 discusses the GPS correlator engine. This following by a detailed analysis of Engine 2. Multi-channel GPS correlation is introduced in section 7.4, and Section 7.5 discusses the results which are following by a conclusion section.

7.2 GPS Engine Correlator Capability Review

The focus of the previous chapter was to discuss the novel implementation of a GPS correlator processor based on a dynamic reconfiguration platform. Various optimisation

techniques have been implemented in order to conclude with the possibility of implementing an architecture which can achieve a correlation time of 62 μs for a 1023 samples in a GPS channel. Even though such a high-speed correlation has been achieved, applying this for a single channel in practical implementation is not the whole story, because the engine still has to wait for 1023 samples or a complete 1 ms in order to fully receive all of the GPS samples for the dedicated channel. This means that the engine is running for only 62 μs out of the 1 ms available time for a complete frame. This represents just 6.2% of the time while the other 93.8% of the time is spent in sleeping or idle mode. In other words, in order to benefit from such a novel high speed correlation engine, multiple GPS channels in parallel or multi-correlations must be handled. In the remainder of this chapter, the engine which runs for a single correlation, which is the outcome of the previous chapter, is hereafter named 'Engine 1'.

Table 7-1 Industrial correlator processors and associated correlation channels

GPS correlation Engine	Number of correlation channels	Ref
Zarlink GP1020	6 CH correlator	[94]
Zarlink GP2021	12 CH correlator	[86]
Zarlink GP4020	12 CH correlator	[95]
Atmel ATR0620	16 CH correlator	[96]
ST STA2062	32 CH correlator	[97]
Atheros AR1511	8 CH correlator	[100]
ST STA8058	16 CH correlator	[101]
LOCOSYS LS20030	Up to 66 correlator	[102]

7.3 GPS correlation 'Engine 2'

Table 7-1 lists the main existing industrial correlator processors and their associated numbers of channels. It is clear that a 12-channel correlator is the average acceptable standard in modern industrial dedicated GPS correlation processors. The research work in this chapter focuses on designing a correlator engine to process 12 or more GPS channels in parallel. This requires a modification to 'Engine 1' in order to adjust it so as to be capable of dealing with multiple channels in parallel. It is clear that 'Engine 1' is capable of executing the correlation process, but only for a single channel. In order to

accomplish precise processing for 12 channels within the limit of 1ms, extra cells, functionalities and optimisations are required. ‘Engine 2’ is based on updating ‘Engine 1’ with additional routines, mainly logic functions; hence, the necessity to increase the number of logic cells in the engine. In order to maintain the sample processing stream, and to overcome the overheads imposed due to the extra cells needed and to maintain the processing speed of 62μs, the amount of samples processed has been reduced. A reduction of almost 20% of the amount of data processed per run has been implemented. The number of samples has been optimised to be between 816 to 852 samples instead of 1023 per run. This is the number of samples that will be processed in a 62 μs time frame. However the design is for 12 complete 1023 samples for 12 different GPS channels to be completed in 1ms.

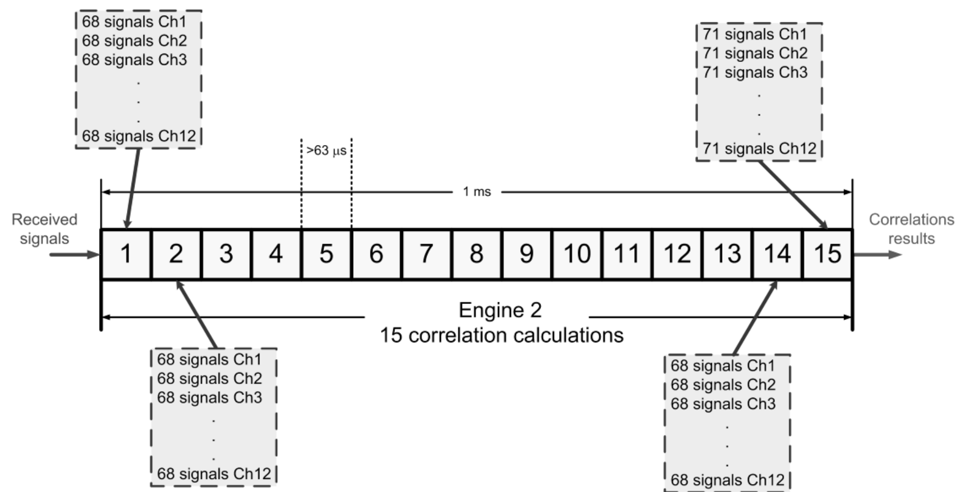

Figure 7-1 ‘Engine 2’ is capable of providing 12 GPS correlation channels

Figure 7-1 demonstrates the data allocation in the ‘Engine 2’ design for processing the 12 correlation channels in parallel. The design is based on sampling and processing the first 68 samples (signals) of each of the 12 channels. The samples have been divided into 15 segments (1 to 15): 68 samples per channel for the first fourteen and 71 samples per channel for the 15th segment. This allows the 1023 samples/channel or the complete frame to be completed. After trying different designs, this arrangement has proven to be the most optimised in terms of processing time, energy consumption and engine size,

where the latter refers to the number of cells involved in the design of the engine. Engine 2 has been simulated successfully, and it has been proven that the novel dynamic reconfigurable GPS correlation engine is capable of efficiently processing 12 correlation channels.

7.4 Multi-Channel Correlation Solution

Having achieved 12 channel correlations to match the industrial norm for 12 parallel correlations is a good step forward. However, it is not enough. Achieving the 12-channel correlation on a dynamically reconfigurable engine is a novel contribution of this research, an important part of which is the results and discussion that follow. It was desired to design the engine for more than the 12 channels in order to exploit the dynamic reconfiguration platform to the limit and to see how far it can go in terms of number of the parallel correlations that could be implemented and the associated overheads imposed.

This will allow a clear understanding of the platform and its dynamics and the balance between various design factors of area and energy when there is a clear constraint on processing time.

In most industrial correlators at present, the maximum available capacity is for 32 parallel channels, although some have gone beyound 50. In order to exceed this, ‘Engine 3’ was designed. This is capable of processing 72 channels in parallel, and has been achieved by the integration of six 12 channel correlators in parallel, as illustrated in the dataflow and time processing diagram shown in Figure 7-2. This has been achieved through the integration of 6 ‘Engine 2’ cores resulting in 6x12 = 72 parallel correlations. Based on the same concept ‘Engine 4 was then designed by the integration of two ‘Engine 3’ cores resulting in 2 x 6x12 = 144 parallel correlations. The processing time and dataflow diagram for ‘Engine 4’ are presented in Figure 7-3. Similarly ‘Engine 5’ was designed based on the integration of two ‘Engine 4’ cores, as demonstrated in Figure 7-4, resulting in the capacity to handle 288 parallel GPS channel correlations.

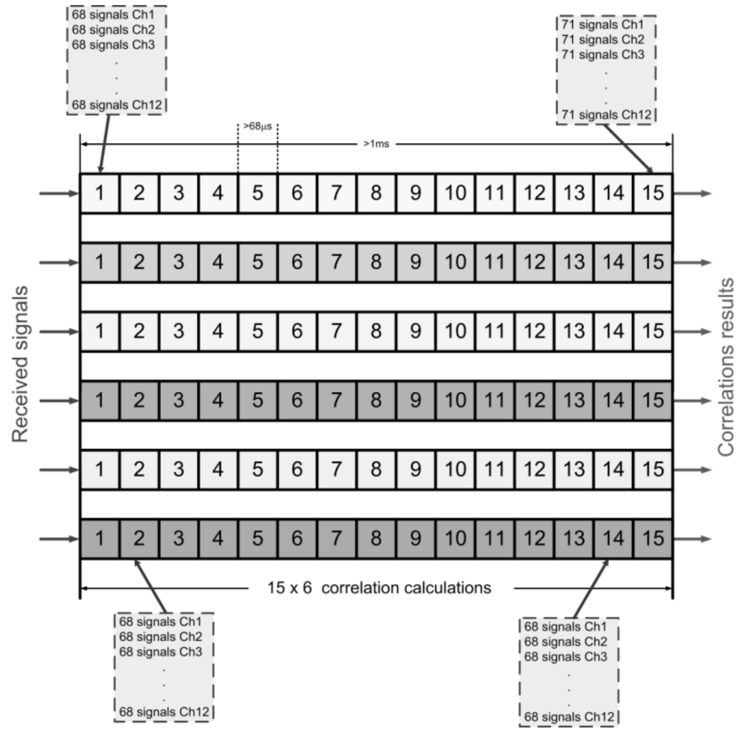

Figure 7-2 'Engine 3' is capable of providing 72 GPS correlation channels

Reaching such enormous number of correlations, two questions arise. Firstly, is it useful? And secondly what are the overheads involved? The answer to the first question is given here, while will the second is postponed until the following section which discusses the overheads involved.

On average there are likely to be 8 to 12 satellites available on the horizon at any one time to receive data from. The main focus in this research is the possibility to achieve the shortest Time to First Fix or the quickest achievement of a location. This is of interest from a research point of view as well as industrial or commercial motivations. It is well known that the minimum number of satellites required to get a location fix is basically four; however, in order to speed up the search process, and due to the fact that there is a possibility of having up to 12 satellites on the horizon of GPS receivers could be upgraded to handle many more channel correlations. The more correlations available on the receiver, the faster it can achieve an earlier time-to-first-fix, as the extra

correlators allow a speedier search of the satellites and quick correlations of the signal in the search domain.

7.5 Analysisof Results and Discussion

As discussed in chapter 6, the designed GPS Engines which are based on dynamically reconfigurable platform and constructed of various cell types and numbers.

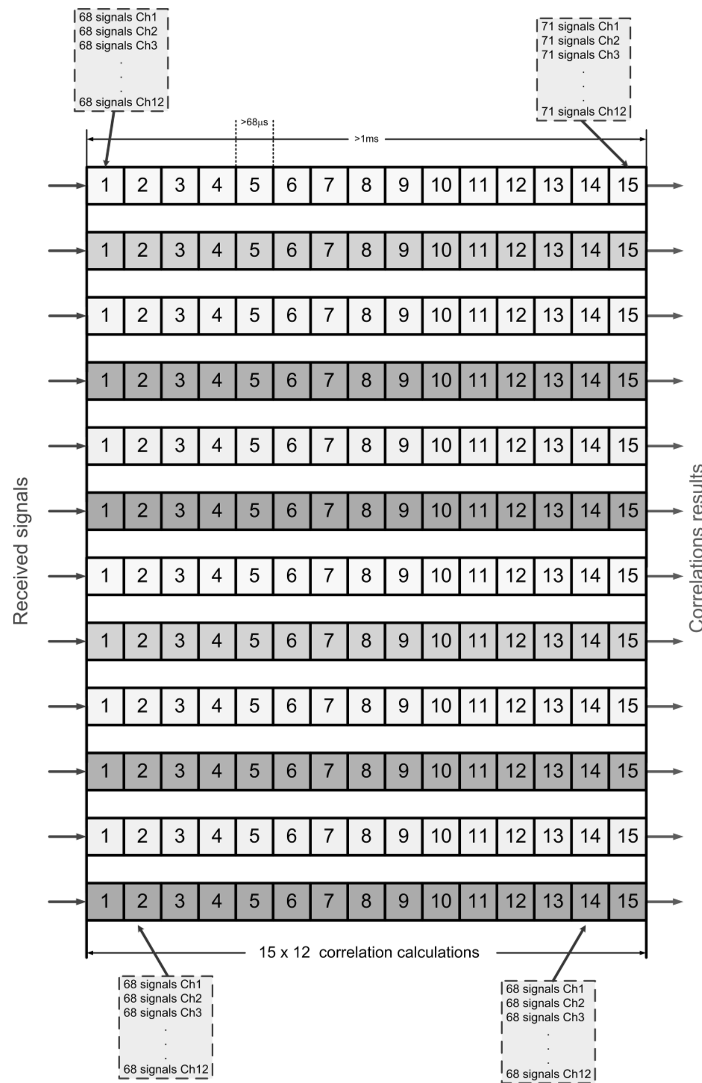

Figure 7-3 'Engine 4' is capable of providing 144 GPS correlation channels

Figure 7-5 presents results of the single correlation processing time on the various engines. It would be anticipated that there should not be any different, and that it should remain below 63 μs as is the case for "Engine 1". However, the obtained results demonstrate that processing time increased with the increase in the number of the embedded engines. With details analysis it appear to be obvious that due to the increase of the number of cells, the associated delay in signals, processing and step time has to increase. "Engine 5" is by far the most complex engine, hence the longest processing time per correlation of 69.47 μs. Similarly, the various comparison parameters show the same trend. Figure 7-6 demonstrates clearly the memory usage trend among the various engines, while Figure 7-7 represents the total dynamic energy for the various engines. That is why the data memory access energy followed the same upward trend in consistency with the other parameters as presented in Figure 7-8.

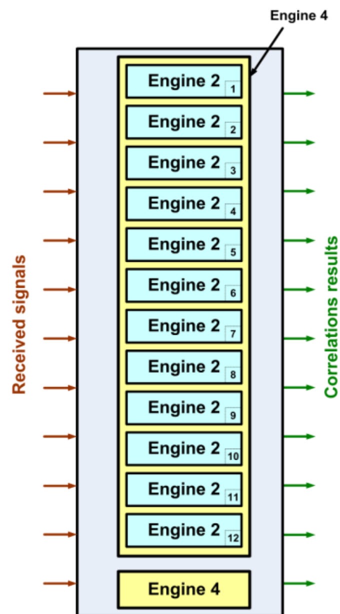

Figure 7-4 'Engine 5' core based on two 'Engine 4' cores or twenty four 'Engine 2' cores for providing 288 GPS correlations

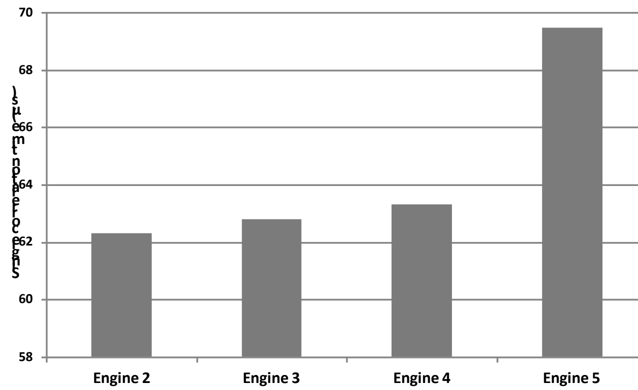

Figure 7-5 Variation in the single correlation results between the different multi-correlation engines

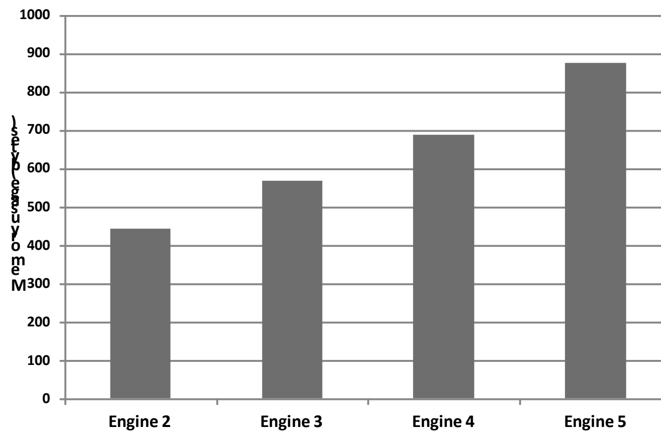

Figure 7-6 Memory usage for the different multi-correlation engines

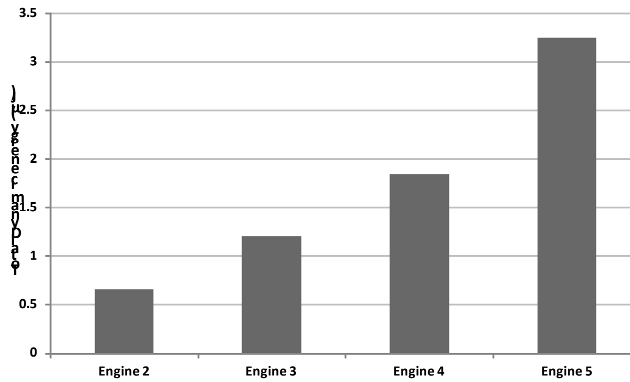

Figure 7-7 Total dynamic energy for the different multi-correlation engines

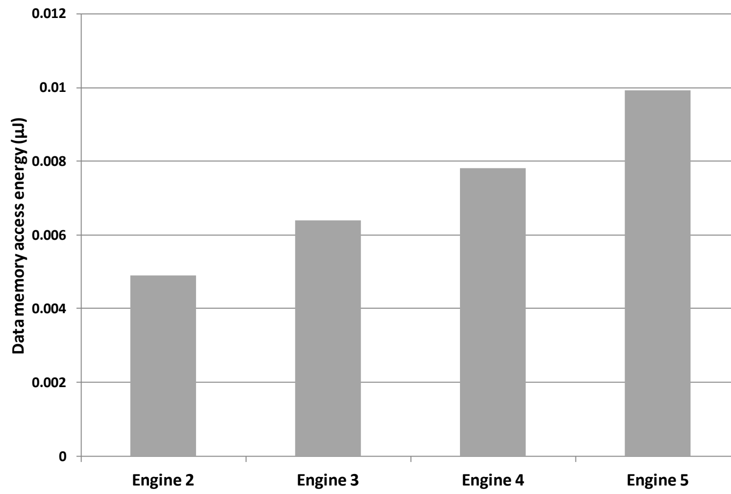

Figure 7-8 Data memory access energy for the different multi-correlation engines

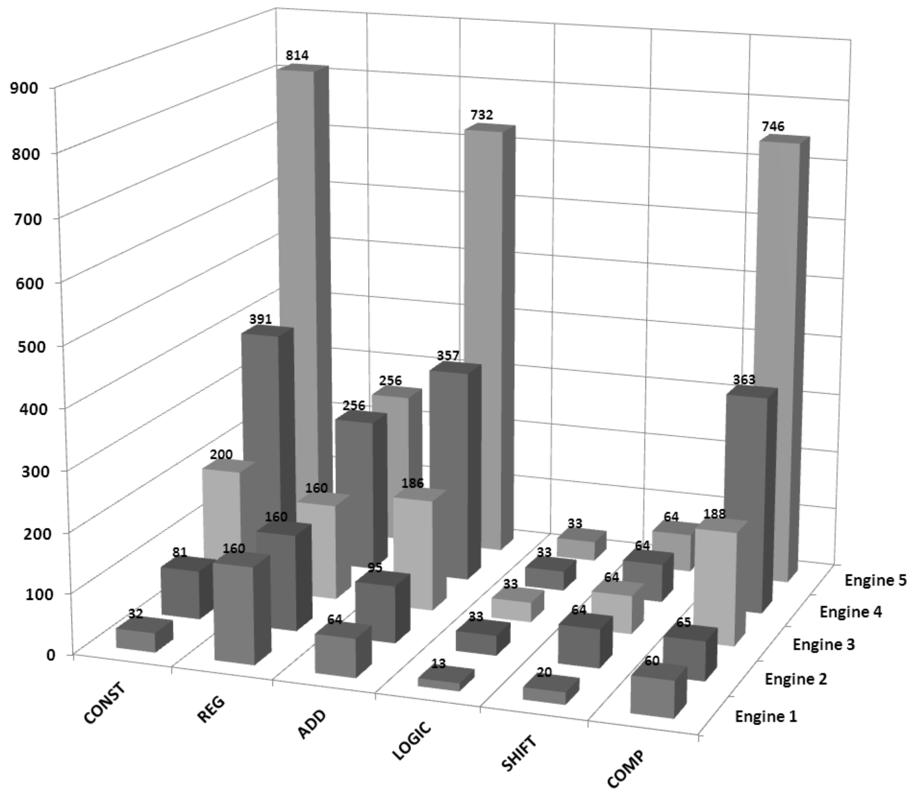

Figure 7-9 Number of core cells used in each engine

Table 7-2 Summary of key cell numbers used in each engine

	CONST	REG	ADD	LOGIC	SHIFT	COMP
Engine 1	32	160	64	13	20	60
Engine 2	81	160	95	33	64	65
Engine 3	200	160	186	33	64	188
Engine 4	391	256	357	33	64	363
Engine 5	814	256	732	33	64	746

Table 7-2 shows the key cell types and their numbers, and Figure 7-9 indicates the number of core cells associated with each engine.

From Table 7-2, cells can be divided into two categories in respect of their usage within the engines. Category 1 includes cells necessary to facilitate the concatenation of the 12 parallel correlations, i.e. from ‘Engine 1’ to ‘Engine 2’. Those cells are ‘Logic’ and ‘Shift’ cells, and from ‘Engine 2’ upwards there is no need to increase their numbers. This is as expected, since those cells are responsible for the necessary work in handling and manipulating data in the various correlation steps inside the engines for the multiple channels.

Category 2 cells represent the core parts of the engines or the correlation process. Those are mainly, ‘Add’, ‘Comp’ and ‘Const’ cells.

The overheads associated with the increased number of correlation channels will be reflected in the extra numbers of cells used which will be physically integrated into the GPS correlation processor. In order to analyse this relationship, a normalised chart is given in Figure 7-10. Normalisation is necessary here due to the vast variations in the number of cells of different types, used as shown in Table 7-2.

In Figure 7-10 the numbers of the two different categories of cell clearly demonstrate two different trends. In addition, the dashed line shows that a logarithmic relationship is almost consistent for the ‘Add’, ‘Comp’ and ‘Const’ cells. However, Figure 7-11 represents the normalised number of correlations associated with each engine, and that the correlation curve on the chart shows that the relationship in fact is almost linear with the increase in the number of ‘Engine 2’ integrated into Engines 3, 4 and 5.

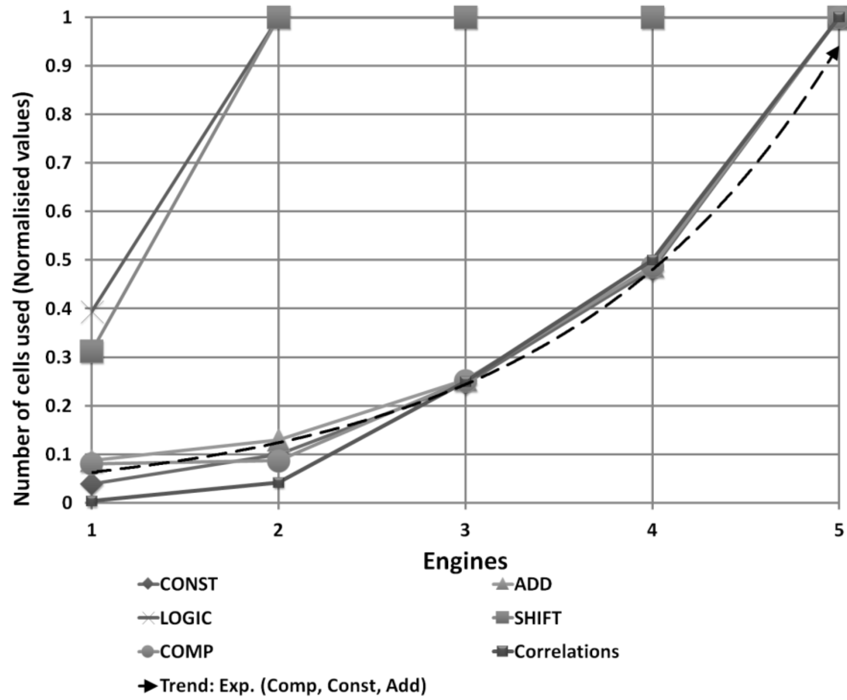

Figure 7-10 Normalised cell numbers per correlation engine

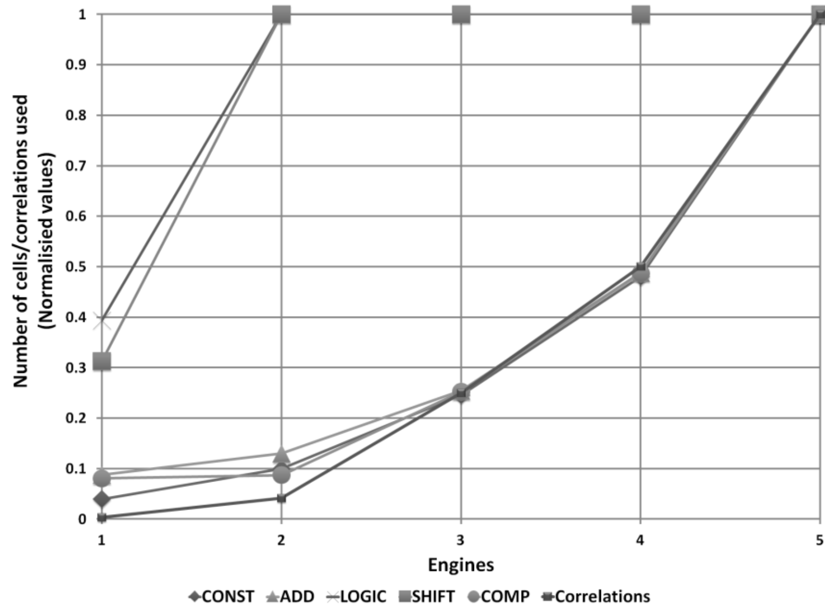

Figure 7-11 Normalised number of cells used per engine and the associated normalised number of correlations

Table 7-3 Summary of the number of correlations and embedded ‘Engine 2’s’ in each engine

	Correlations	Number of embedded ‘Engine 2’
Engine 1	1	n/a
Engine 2	12	1
Engine 3	72	6
Engine 4	144	12
Engine 5	288	24

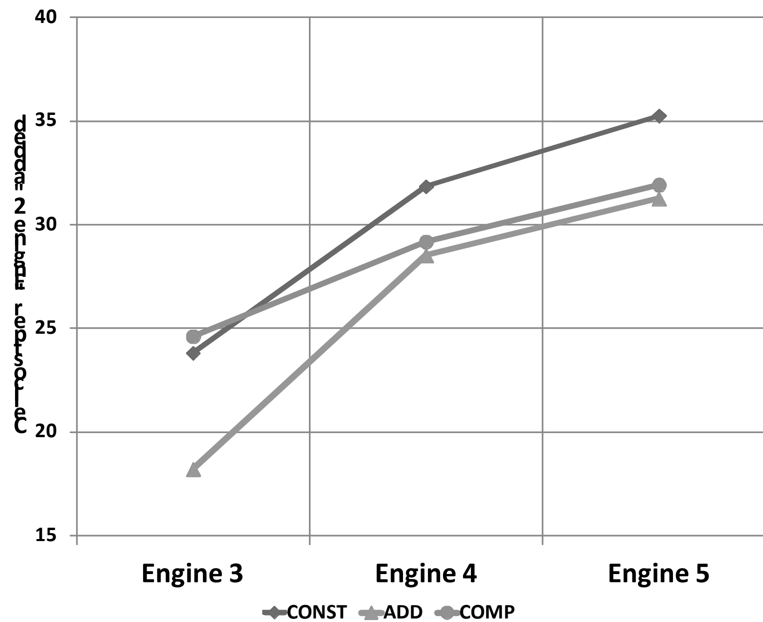

Figure 7-12 Cost of adding “Engine 2” to the various engines

In order to estimate the resulting overheads, the focus is clearly on the category 2 cells, ‘Add’, ‘Comp’ and ‘Const’. A key parameter assisting in the calculations of overhead is the number of parallel correlations and, most importantly, the number of embedded ‘Engine 2’ in engines 3 to 5. This is summarised in Table 7-3. As ‘Engine 2’ is the basic core for the other engines, Figure 7-12 represents the calculated cell cost per added ‘Engine 2’. The cost is represented by the extra number of cells added, which has been calculated using equation (7-1).

$$\begin{aligned}&Cell\ x\ cost\ per\ added\ \text{“}Engine\ 2\text{”}\\&= \frac{(Cell\ x\ total\ number\ for\ Engine\ M - Cell\ x\ numbers\ for\ \text{"}Engine\ 2\text{"})}{(Embedded\ number\ of\ \text{"}Engine\ 2\text{"}\ in\ Engine\ M) - 1}\end{aligned} \quad (7\text{-}1)$$

where x is cell type and M is from 3 to 5.

By implementing this equation (7-1) and using the data from Table 7-3, it is obvious that the cost escalates for category 2 cells, as illustrated in Figure 7-12. As the number of integrated correlations increase, the engine complexity increases and therefore the implementation area will increase due to the increase in integrated cell numbers. There is a clear trade-off between the number of correlations required and the area and power consumption which are proportional to the increase in cell number.

7.6 Conclusions

It is concluded that, for practical realisations of a multi-correlation engine, ‘Engine 2’ and ‘Engine 3’ provide the optimal solutions where 12 and 72 correlations respectively are calculated. Moreover, a compromise could be achieved by having an engine with three embedded ‘Engine 2’ instead of six, which is the case in ‘Engine 3A’. This new engine would provide 36 parallel correlations. Designing and simulating ‘Engine 3A’ which supports 36 parallel correlations resulted in an engine with the following cells: Comp 111, Const 126, Add 128, Reg 154, Logic 33 and Shift 64. Moreover, the engine is based on three embedded ‘Engine 2’ correlators. This represents a step forward in the area of dynamically reconfigurable architectures and correlation systems.

Chapter 8

SUMMARY AND FUTURE WORK

8.1 Intoduction

The aim of this thesis has been to investigate an efficient reconfigurable architecture for telecommunication systems in general and baseband in particular. The key blocks investigated for reconfigurable performance evaluations are the convolution coder, Interleaver, Reed Solomon encoder and decoder, and GPS correlator.
The main aim was to introduce a reconfigurable architecture capable of handling intensive processing tasks whilst using the lowest possible power consumption.

This chapter is organised into four sections. The first section summarises the contents of the thesis and identifies the contributions made. The second section draws conclusions from the work presented in this thesis. Concluding remarks are provided in the third section, and the final section outlines areas for future investigation.

8.2 Summary of Thesis and Contribution

This thesis has investigated the possibility of realising various communication systems (baseband in particular) on a dynamically reconfigurable architecture.

Chapter 2 discussed the various reconfigurable architectures, with a focus on coarse-grain reconfigurable architecture. In order to design a reconfigurable architecture to suit the challenges of communication systems, it has to have many crucial characteristics. Its primary feature should be low power consumption. In order to sustain low power consumption, the use of heterogeneous PEs is the best approach. PEs can be tailored specifically to the system's needs, resulting in the highest utilisation which then means lower power consumption and smaller area.

It appears from the above that the most promising architectures for telecommunications systems are the MOVE and RICA. These are very different architectures; however, they are alike in that their PEs are heterogeneous, they are standalone systems which do not need external processor for control and are C-language programmable.

However, the RICA appears to be superior in power consumption, since the processor has been built with low power being its central principles and is dynamically reconfigurable, while the MOVE designers only began to address power savings at a later stage.

Chapter 3 introduced a novel reconfigurable architecture that provides a multi-rate punctured convolution coder. This architecture can be used in wired and wireless communication systems, and it incorporates both convolution and puncturing.

The convolution-punctured multi-rate architecture has achieved a superior throughput of 100 Mbps for all the required rates. Although the main architecture is the core that provides the concatenated convolution-punctured code, the reconfigurable input and output interface designs were added to broaden the usability of this reconfigurable fabric. The main advantage of this architecture is that a single clock cycle is enough to provide the parallel convolution-punctured code for its parallel inputs, which can be used to maximise the throughput of the whole transmitter system.

Chapter 4 introduced a novel reconfigurable interleaver. The target application was the WiMAX standard with its sophisticated block size systems. The Interleaver has

been researched and designed into a reconfigurable fabric architecture and with a dynamically reconfigurable instruction cell-based architecture (RICA). The interleaver's throughput as a reconfigurable fabric satisfies the standard requirement, while on RICA the throughput as well as the dynamic power consumption were superior to the fabric realisation and other ASIC realisations. These results are a good step forward towards a fully reconfigurable baseband telecommunications system. Moreover, the results are a promising step towards integrating the whole WIMAX on a dynamically reconfigurable (RICA) architecture.

Chapter 5 introduced a novel Reed Solomon encoder architecture with parallel parity output. A novel high speed and low power 32-bit Galois Field (GF) multiplier cell was embedded within the novel low-power processor for programmable Reed Solomon coding, and its design, optimisation and implementation have been introduced. The real-time programmable RS encoder and decoder processor supports several communication standards such as WiMAX and DVB-H. A number of approaches and optimisation techniques have been implemented in order to enhance the performance of the processor. The processor achieves high throughput and provides significant improvements in performance and energy consumption.

The novel GF multiplier cell leads to a reduction in memory access energy of 72.4%, which in turn improves the processor performance. Different design approaches and optimisation techniques have been applied in order to enhance the processor throughput and reduce its energy consumption. The throughputs achieved are up to 200 Mbps and 92 Mbps for the encoder and decoder respectively. The associated dynamic energy consumption is in the range of 0.34 to 1.17μJ, which represents a design suitable for present and future mobile devices.

Chapter 6 introduced a novel engine based on a dynamically programmable platform targeting the computationally intensive correlation function used in GPS-based positioning. Various optimisation techniques have been exploited in order to

achieve the best performance. In addition, modified MCDMF, MDDMF and MSPDMF correlation architectures have been introduced, which demonstrate efficiency in terms of correlation time and energy consumption. Furthermore, the bitwise optimisation technique has been applied to digital matched filters that demonstrate the maximum utilisation of the architecture, leading to a high correlation speed of 62 µs for 1023 phase search correlations. Comparisons of the results achieved and related architectural configurations have been presented. This work is a promising step towards high-speed, ultra-low-energy-GPS receivers.

Chapter 7 presented a novel optimised multi correlation processor. It is concluded that for the practical realisation of the multi-correlation engine, the ‘Engine 2’ and ‘Engine 3’ designs provide the optimum solutions, where 12 and 72 correlations respectively can be calculated. Moreover, a compromise could be achieved by having an engine with three embedded ‘Engine 2’ instead of six, which is the case in ‘Engine 3A’. This new engine would provide 36 parallel correlations. Designing and simulating ‘Engine 3A’ which supports 36 parallel correlations resulted in an engine with the following numbers of cells: Comp 111, Const 126, Add 128, Reg 154, Logic 33 and Shift 64. Moreover, the engine is based on three embedded ‘Engine 2’ correlators. This work represents a step forward in the area of dynamically reconfigurable architectures and correlation systems.

The main achievement of this work is the development of a multi-correlator module for the GPS system. This is in addition to the RS codec reconfigurable processor. Moreover, the GF Mul reconfigurable cell, reconfigurable convolution and interleaver which have been developed are important for reconfigurable telecommunication systems.

All of the above give a clear picture of how to realise the full potential of a dynamically reconfigurable architecture targeting full telecommunications baseband system.

8.3 FUTURE WORK

This work can be combined with existing research on reconfigurable digital signal processing blocks in order to produce an overall reconfigurable baseband transceiver.

A full transceiver of WiMAX or WiFi would be the logical step forward for defining the best reconfigurable architecture suitable for telecommunications systems. Further algorithms and processor-specific optimisations are expected to be necessary in conducting further research in this area. Figure 8-1 demonstrate's a conceptual design of how completely reconfigurable baseband telecommunication architecture could be. In this diagram, the architecture is reconfigurable between 802.11 and 802.16 or WiFi and WiMAX systems; this is inspired by the Espacenet project mentioned in earlier chapters. A development of a reconfigurable mapper, pulse shaping, randomiser and modulation would be necessary for future work to achieve fully reconfigurable baseband system.

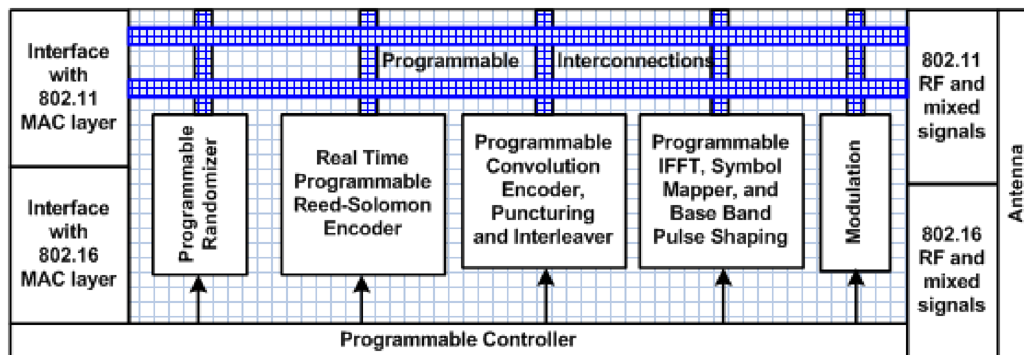

Figure 8-1 Fully reconfigurable baseband architecture

This would be the first stage in a larger project, and would need to be followed by further stages in order to create a universal reconfigurable architecture for telecommunications, which would be capable if carrying out any communication

protocol needed. This will allow future universal, adaptable mobile devices where the system will automatically reconfigure the architecture necessary for the communication protocol needed in particular locations or situations. This will have an enormous influence on device size and performance. Moreover, this will greatly increase battery life and reduce the production costs of devices.

The first step to proceed forward with this research work for the GPS receiver is the design and fabrication of an Engine 2 processor. In addition, further work should follow with either of Engines three, three-A or four. This will provide rich data for more in-depth analysis of processor performance, and its benefits and overheads.
Clearly, this work will not only affect the progress of research into dynamically reconfigurable architectures, but will influence the implementation of navigation processors. This is due to the anticipation of better performance and lower power consumption to be achieved with the new processors. Furthermore, the development of a complete GPS receiver would be a logical research step to follow; hence, this will lead to challenges in the integration and data handling capability of the new processor.

REFERENCES

[1] G. Estrin, B. Bussel, R. Turn, and J. Bibb, Parallel processing in a restructurable computer system, *IEEE Transactions on Electronic Computers*, vol.EC-12, no.6, pp.747-755, Dec 1963.

[2] W. Carter, K Duong, R H Freeman, H Hsieh, J Y Ja, J E Mahoney, L T Ngo, er al., A user programmable reconfigurable gate array, *CICC Proceedings*, May 1986, pp. 233-235, 1986.

[3] FPGA - Field Programmable Gate Array, available: http://www.fpgacentral.com/pld-types/fpga-field-programmable-gate-array, accessed on 23/3/2013.

[4] The Industry's Breakthrough 7 Series FPGA Families, available at http://www.xilinx.com/products/silicon-devices/fpga/index.htm, accessed on 23/3/2013.

[5] R.D. Wittig and P. Chow, OneChip: an FPGA processor with reconfigurable logic, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp.126-135, 17-19 Apr 1996.

[6] R. Razdan, M.D. Smith, A high-performance microarchitecture with hardware-programmable functional units, *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp.172-180, 30 Nov-2 Dec 1994.

[7] S. Hauck, T.W. Fry, M.M. Hosler, J.P. Kao, The Chimaera reconfigurable functional unit, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.12, no.2, pp.206-217, Feb. 2004.

[8] T. Miyamori and K. Olukotun, REMARC: Reconfigurable Multimedia Array Coprocessor, *IEICE Transactions on Information and Systems*, vol. E82-D, pp. 389-397, 1998.

[9] C. Ebeling, and D.C. Cronquist, and P. Franklin, RaPiD - Reconfigurable pipelined datapath, Lecture Notes in Computer Science: Field-Programmable Logic Smart Applications, New Paradigms and Compilers, Vol. 1142, pp. 126-135, 1996

[10] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, B. Hutchings, A reconfigurable arithmetic array for multimedia applications, *Proceedings of ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pp. 135-143, 1999.

[11] R.P.S. Sidhu, S. Wadhwa, A. Mei, V.K. Prasanna, A Self-Reconfigurable Gate Array Architecture, *Proceedings of The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, pp. 106-120, 2000.

[12] R. David, D. Chillet, S. Pillement, O. Sentieys, DART: a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints, *Proceedings Parallel and Distributed Processing International Symposium*, pp. 8, 15-19 April 2001.

[13] F. Barat, M. Jayapala, T. Vander, et. al., Low Power Coarse-Grained Reconfigurable Instruction Set Processor, *In Field Programmable Logic and Application*, Vol. 2778, pp. 230-239, 2003.

[14] G. Sassatelli, G. Cambon, J. Galy, L. Torres, A dynamically reconfigurable architecture for embedded systems, *12th International Workshop on Rapid System Prototyping*, pp.32-37, 2001.

[15] E. Mirsky, A. DeHon, MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157-166, 17-19 Apr 1996.

[16] L.J.K. Durbeck and N.J. Macias, The Cell Matrix: an architecture for nanocomputing, *Nanotechnology Journal*, Vol. 12, Issue 3, pp. 217-230, 2001.

[17] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, J.M. Rabaey, A 1 V heterogeneous reconfigurable processor IC for baseband wireless applications, *IEEE International Solid-State Circuits Conference*, pp.68-69, 9-9 Feb. 2000.

[18] J.R. Hauser, J. Wawrzynek, Garp: a MIPS processor with a reconfigurable coprocessor, *The 5th Annual IEEE Symposium Proceedings on Field-Programmable Custom Computing Machines*, pp.12-21, 16-18 Apr 1997.

[19] J. Becker, T. Pionteck, C. Habermann, M. Glesner, Design and implementation of a coarse-grained dynamically reconfigurable hardware

architecture, *IEEE Computer Society Workshop on VLSI*, pp.41-46, May 2001.

[20] D.C. Chen and J.M. Rabaey, PADDI: Programmable Arithmetic Devices for Digital Signal Processing, *IEEE VLSI Signal Processing*, vol. IV, pp. 240-249, IEEE Press, Nov. 1990.

[21] D.C. Chen and J.M. Rabaey, A Reconfigurable Multiprocessor IC for Rapid Prototyping of Real Time Data Paths, *IEEE Journal of Solid State Circuits*, Vol. 27, No. 12, pp. 1895-1992.

[22] P. Hilfinger, A high-level language and silicon compiler for digital signal processing, *IEEE Custom Integrated Circuits Conferences proceedings*, pp. 240-243, May 1985.

[23] J. A. Hennessy, D. L. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann Publishers, 1990.

[24] D. Kesler, S. Dautovic, R. Struharik, Design and verification of dynamically reconfigurable architecture, *IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics (SISY)*, pp.413-418, 20-22 Sept. 2012.

[25] H. Singh, Lee Ming-Hau, Lu Guangming, F.J. Kurdahi, N. Bagherzadeh, E.M. Chaves Filho, MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications, *IEEE Transactions on Computers*, vol.49, no.5, pp.465-481, May 2000.

[26] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, R. Laufer, PipeRench: a coprocessor for streaming multimedia acceleration, *Proceedings of the 26th International Symposium on Computer Architecture*, pp.28-39, 1999.

[27] R. Hartenstein, M. Herz, T. Hoffmann, U. Nageldinger, On Reconfigurable Co-Processing Units, *Proceedings of Reconfigurable Architectures Workshop (RAW98), held in conjunction with 12th International Parallel Processing Symposium (IPPS-98) and 9th Symposium on Parallel and Distributed Processing (SPDP-98)*, Orlando, Florida, USA, March 30, 1998.

[28] R.W. Hartenstein, M. Herz, T. Hoffmann, U. Nageldinger, Using the KressArray for reconfigurable computing, *Proceeding of SPIE, Configurable Computing: Technology and Applications*, vol. 3526, Boston, USA, 1998.

[29] R.W. Hartenstein, R. Kress, A datapath synthesis system for the reconfigurable datapath architecture, *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scale* , pp.479-484, 29 Aug-1 Sep 1995.

[30] H. Corporaal, Design of transport triggered architectures, *Proceedings of Fourth Great Lakes Symposium on Design Automation of High Performance VLSI Systems*, GLSV '94, pp.130-135, Mar 1994.

[31] J. Heikkinen, J. Sertamo, T. Rautiainen, J.Takala, Design of transport triggered architecture processor for discrete cosine transform, *15th Annual IEEE International ASIC/SOC Conference*, pp.87-91, 25-28 Sept. 2002.

[32] P. Hamalainen,; J. Heikkinen, M. Hannikainen, T.D. Hamalainen, Design of transport triggered architecture processors for wireless encryption, *8th Euromicro Conference on Digital System Design Proceedings*, pp.144-152, 30 Aug.-3 Sept. 2005.

[33] J. Heikkinen, J. Takala, A. Cilio, and H. Corporaal, On Efficiency of Transport Triggered Architectures in DSP Applications, *Advances in Systems Engineering, Signal Processing and Communications*, pp. 25-29, WSES Press, New York, NY, USA, 2002.

[34] H. Yifan, S. Dongrui, B. Mesman, H. Corporaal, MOVE-Pro: A low power and high code density TTA architecture, *International Conference on Embedded Computer Systems (SAMOS)*, pp.294-301, 18-21 July 2011.

[35] S. Khawam, I. Nousias, M. Milward, Yi Ying, M. Muir, T. Arslan, The Reconfigurable Instruction Cell Array, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol 16, pp 75-85, Jan. 2008.

[36] A. Major, Yi Ying, I.Nousias, M. Milward, S. Khawam, T. Arslan, H.264 Decoder Implementation on a Dynamically Reconfigurable Instruction Cell Based Architecture, *IEEE International SOC Conference*, pp.49-52, 24-27 Sept. 2006.

[37] Z. Wang, A.T. Erdogan, T. Arslan, “A SDR Platform for Mobile Wi-Fi/3G UMTS System on a Dynamic Reconfigurable Architecture*, 2009 European Signal Processing Conference (EUSIPCO-2009)*, August 24-28, 2009.

[38] I. Nousias, S. Khawam, M. Milward, M. Muir, T. Arslan, A Multi-objective GA based Physical Placement Algorithm for Heterogeneous Dynamically Reconfigurable Arrays, *17th International Conference on Field*

Programmable Logic and Applications (FPL 2007), pp. 497-500, Amsterdam, Netherlands, 27-29 August 2007.

[39] Z. Wang, T. Arslan, A.T. Erdogan, Implementation of Hardware Encryption Engine for Wireless Communication on a Reconfigurable Instruction Cell Architecture, *4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008)*, pp.148-152, 23-25 Jan. 2008.

[40] T. Hirao, Kim Dahoo, I. Hida, T. Asai, M. Motomura, A restricted dynamically reconfigurable architecture for low power processors, *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp.1-7, Dec. 2013.

[41] O. Atak, A. Atalar, BilRC: An Execution Triggered Coarse Grained Reconfigurable Architecture, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.21, no.7, pp.1285-1298, July 2013.

[42] R.W. Brodersen, J.M. Rabaey, Evolution of Microsystem Design, *Proceedings of the 15th European Solid-State Circuits Conference ESSCIRC '89*, pp. 208-217, 20-22 Sept. 1989.

[43] R. Hartenstein, Coarse grain reconfigurable architectures, *In Proceedings of the 2001 Asia and South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 564-570, 2001.

[44] IEEE Std 802.16-2004, IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for fixed broadband wireless access systems.

[45] Digital Modulation in Communications Systems - An Introduction, Application Note 1298, http://cp.literature.agilent.com/litweb/pdf/5965-7160E.pdf, accessed on 15/09/2014.

[46] ESPACENET. Available: http://www.e-spacenet.net/, accessed on 23/3/2013.

[47] T. Arslan, N. Haridas,; E. Yang, A.T. Erdogan, N. Barton, A.J. Walton, J.S. Thompson, A. Stoica, T. Vladimirova, K.D. McDonald-Maier, W.G.J. Howells, ESPACENET: A Framework of Evolvable and Reconfigurable Sensor Networks for Aerospace–Based Monitoring and Diagnostics, *First NASA/ESA Conference on Adaptive hardware and systems*, pp. 323–329. 15-18 June 2006.

[48] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*, Wiley, 2002.

[49] B. Tang, Parallel punctured convolutional encoder, European patent no. EP 1176727 A2, 2002.

[50] Xilinx LogiCORE IP: Convolutional Encoder v9.0, Xilinx, Inc., San Jose, CA, April 2014, Available: http://www.xilinx.com/support/documentation/ip_documentation/convolution/v9_0/pg026_convolution.pdf.

[51] A.P. Chandrakasan, R.W. Brodersen, Minimising power consumption in digital CMOS circuits, *Proceedings of the IEEE*, Vol. 83, Issue 4, pp. 498-523, April 1995.

[52] T. Matsumoto, F. Adachi, BER analysis of convolution coded QDPSK, in digital mobile radio, *IEEE Transactions on Vehicular Technology*, Vol. 40, No. 2, MAY 1991.

[53] M.J. Meeuwsen, O. Sattari, B.M. Baas, A Full-Rate Software Implementation of an IEEE 802.11a Compliant Digital Baseband Transmitter, *In Proceedings IEEE Workshop on Signal Processing Systems*, 2004, October 2004.

[54] K. Chang and G.E. Sobelman, FPGA-Based Design of a Pulsed-OFDM System, Proceedings, *IEEE Asia Pacific Conference on Circuits and Systems*, pp. 1130-1133, 2006.

[55] K. Chang, G.E. Sobelman, E. Saberinia and A.H. Tewfik, Transmitter Architecture for Pulsed OFDM, *IEEE Asia Pacific Conference Proceedings on Circuits and Systems*, pp. 693-696, 2004.

[56] B. Soreng, S. Kumar, Efficient implementation of Convolution Encoder and Viterbi Decoder, *2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*, pp. 1270-1273, 20-21 March 2013.

[57] Datasheet, The CS3310 Programmable Convolution Encoder, Amphion Semiconductor Ltd., Available: http://www.digchip.com/datasheets/download_datasheet.php?id=240961&part-number=CS3310. Retrieved: 21-06-2014.

[58] E. Tell and D. Liu, A Hardware Architecture for a Multi Mode Block Interleaver, *Proceedings of the International Conference on Circuits and Systems for Communications (ICCSC)*, Moscow, Russia, June 2004.

[59] C. Berrou, S. Evans and G. Battail, Turbo block codes, *Proceedings of Seminar on Turbo Coding*, Lund, Sweden, pp.1-7, Aug. 1996.

[60] O.Y. Takeshita,; Costello, D.J., Jr., New deterministic Interleaver designs for turbo codes, *IEEE Transactions on Information Theory*, vol.46, no.6, pp.1988-2006, Sep 2000.

[61] A. Troya, K. Maharatna, M. Krstic, E. Grass, U. Jagdhold, R. Kraemer, Low-Power VLSI Implementation of the Inner Receiver for OFDM-Based WLAN Systems, *IEEE Transactions on Circuits and Systems*, vol.55, no.2, pp.672-686, March 2008.

[62] R. Machauer, A. Wiesler, and F. Jondral, Comparison of UTRA-FDD and CDMA200 with intra- and intercell interface, *Proceedings IEEE 6th International Symposium on Spread Spectrum Techniques and Applications (ISSSTA '00)*, vol. 2, pp.652–656, NJ, USA, September 2000.

[63] J. Glosser, J. Moreno,M.Mudsill, et al., Trends in compilable DSP architecture, *Proceedings of Workshop on Signal Processing Systems (SiPS 2000)*, October 2000, USA, pp. 181–199.

[64] Y.S. Kavian, A.Falahati, A. Khayatzadeh, M. Naderi, High Speed Reed-Solomon Decoder with Pipeline Architecture, *Wireless and Optical Communications Networks, 2005, WOCN 2005, Second IFIP International Conference*, Mar 2005, pp.415-419.

[65] I. Reed and G. Solomon, Polynomial codes over certain Finite Fields, *Journal of the Society for Industrial and Applied Mathematics*, Vol. 8, No. 2, pp. 300-304, June 1960.

[66] S.B. Wicker and V.K. Bhargava, *Reed-Solomon Codes and Their Applications*, IEEE Press, September 1999.

[67] R. Riemann and K. Winstein, Improving 802.11 Range with Forward Error Correction, MIT Computer Science and Artificial Intelligence Laboratory Technical Report, Feb 2005.

[68] J.L. Massey, Deep Space Communications and Coding: A Match Made in Heaven, in Advanced Methods for Satellite and Deep Space Communications, J. Hagenauer (ed.), Lecture Notes in Control and Information Sciences, Volume 182, Berlin: Springer-Verlag, 1992.

[69] M.S. Schulte, M.J. Iancu, D. Iancu, A. Glossner, J. Instruction set extensions for Reed-Solomon encoding and decoding, *16th IEEE International Conference on Application-Specific Systems, Architecture*, pp. 364- 369, July 2005.

[70] D. Taipale, I.E. Scheiwe and T.M. Redheendran, Reed-Solomon Decoding on the StarCore Processor, Tech. Rep. AN1841/D, Motorola Semiconductors, Denver, Colombia, USA, May 2000.

[71] W.J. Gross, F.R. Kschischang, P.G. Gulak, An FPGA Interpolation Processor for Soft-Decision Reed-Solomon Decoding, *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pp. 310-311, 2004.

[72] J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, M. Ware, Trends in compilable DSP architecture, *Proceeding Workshop on Signal Processing Systems (SiPS 2000)*, USA, pp. 181-199, October 2000.

[73] Implementation guidelines for DVB handheld services (draft TR 102 377 V1.3.1), DVB Document A092, July 2005.

[74] A.O. El-Rayis, Xin Zhao, T. Arslan, A.T. Erdogan, Dynamically programmable Reed Solomon processor with embedded Galois Field multiplier, *International Conference on FPT 2008*, pp.269-272, 8-10 Dec. 2008.

[75] J. Kim, T. Kim, Memory Access Optimisation Through Combined Code Scheduling, Memory Allocation, and Array Binging in Embedded System Design, *Proceeding of 42nd Design Automation Conference 2005*, pp.105-110, Jun 2005.

[76] Weisstein, http://mathworld.wolfram.com/HornersRule.html accessed on 8th Sep 2014.

[77] M.H. Jing, T.K. Truong, Y.H. Chen and Y.C. Luo, The Design of RS Decoder with a Small Core for Portable Communication, *Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems 2004*, Vol. 2, pp.1069-1072, Dec 2004.

[78] B. Skalr, *Digital Communications: Fundamentals and Applications*, Second Edition, Prentice-Hall, 2001, ISBN 0-13-084788-7.

[79] Y.S. Kavian, A. Falahati, A. Khayatzadeh, M. Naderi, High Speed Reed-Solomon Decoder with Pipeline Architecture, *Second IFIP International Conference on Wireless and Optical Communications Networks (WOCN) 2005*, pp.415-419, Mar 2005.

[80] S.S. Lee and M.K. Song, An Efficient Recursive Cell Architecture of Modified Euclid's Algorithm for Decoding Reed-Solomon Codes, *IEEE Transactions on Consumer Electronics*, Vol. 48, Issue 4, pp. 845-849, Nov 2002.

[81] A.C. Dam, M.G.J. Lammertink, K.C. Rovers, J. Slagman, et al, Hardware/Software Co-design Applied to Reed-Solomon Decoding for the DMB Standard, *9th EURPMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pp.447-455, 2006.

[82] W.J. Gross, F. R. Kschischang, P.G. Gulak, An FPGA Interpolation Processor for Soft-Decision Reed-Solomon Decoding, *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pp. 310-311, 2004.

[83] J.B.Y Tsui, Fundamentals of Global Positioning System Receivers A Software Approach, ISBN 0-471-38154-31, Wiley 2000.

[84] L. Winternitz, M. Moreau, G.J. Boegner, Navigator GPS Receiver for Fast Acquisition and Weak Signal Space Applications, ION GNSS, Long Beach, CA, September 21-24, 2004.

[85] R. Gold, Optimal binary sequences for spread spectrum multiplexing, *IEEE Transactions on Information Theory*, vol. 13, pp. 619-621, October 1967.

[86] Zarlink Semiconductor, GP2021 GPS 12-Channel Correlator, datasheet, August 2005.

[87] M. Lieu and T. Chiueh, A low-power digital matched filter for direct-sequence spread-spectrum signal acquisition, *IEEE Journal of Solid-State Circuits*, vol. 36, no. 6, pp.933-943, June 2001.

[88] W.C. Lin, K.C. Liu, and C.K. Wang, Differential matched filter architecture for spread spectrum communication system, *Electronics Letters*, Volume 32, Issue 17, pp. 1539 – 1540, Aug. 1996.

[89] X. Guo, J. Chen, Y. Qiu A new architecture of matched-filter employing coefficient recode technique for spread spectrum communication systems, *5th International Conference on ASIC*, Vol. 2, pp. 800-803, Oct. 2003.

[90] X. Guan, J. Chen, A new algorithm of digital matched filter with a segment processing method, *6th International Conference on ASIC ASICON*, vol.1, pp. 240-243, Oct. 2005.

[91] T. Arslan, M. Millward, S. Khawam, I. Nousias, Y. Ying, Reconfigurable Instruction Cell Array , Patent EP1877927, 2008.

[92] A. El-Rayis, T. Arslan, A.T. Erdogan, Addressing Future Space Challenges using Reconfigurable Instruction Cell Based Architectures, *NASA/ESA Conference on Adaptive Hardware and Systems*, pp.199-203, 22-25 June 2008.

[93] S.A. White, Applications of distributed arithmetic to digital signal processing: A tutorial review, *IEEE ASSP Magazine*, Vo1 6, No.3, pp.4-19, July 1989.

[94] Zarlink GP1020 Available: http://ulp.zarlink.com/zarlink/gp1020-datasheet-jan1997.pdf, accessed on 15-09-2014.

[95] Zarlink GP4020, Available: http://pdf.datasheetcatalog.com/datasheet/zarlinksemiconductor/zarlink_GP4020_MAY_02.pdf, accessed on 15-09-2014.

[96] ATMEL ATR0620, Available: http://pdf.datasheetcatalog.com/datasheet/atmel/doc4574.pdf, accessed on 15-09-2014.

[97] STA8058, TESEO™. high performance GPS multichip module (MCM) http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/CD00174947.pdf, accessed on 14-09-2014.

[98] Datasheet of GPS smart antenna module, LS20030~3 https://cdn.sparkfun.com/datasheets/GPS/LS20030~3_datasheet_v1.3.pdf accessed on 14-09-2014.

[99] ST STA2062, Available: http://www.st.com/web/en/resource/technical/document/data_brief/CD00172700.pdf, accessed on 15-09-2014.

[100] Atheros AR1511, Available: https://wikidevi.com/files/Atheros/specsheets/AR1511.pdf, accessed on 15-09-2014.

[101] ST STA8058 High performance GPS multichip module, Sep 2013, Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/CD00174947.pdf , accessed on 15-09-2014.

[102] LOCOSYS, Datasheet of GPS smart antenna module LS20030, https://cdn.sparkfun.com/datasheets/GPS/LS20030~3_datasheet_v1.3.pdf, accessed on 15-09-2014.

APPENDIX A: MATLAB MODELS

A.1 Interleaver Model:

```
% %aa=
hex_2_dec('3A5EE7AE499E6F1C6FC128BCBDAB57CDBCCDE3A792CA92C24DBC8D7832FBBFDF23ED8A941627A
565CF7D167A45B809CC');
% %bb= dec_2_bin(aa);
%                                    m                                    =
('898407646983235129903369408620228549208237089767538283785779873984348065286651305271483786817872270
2216228486253004');
% %m is 115 digit decimal
%                                                                         %m
=('00111010010111101110011110101110010010011001111001101111000111000110111111000001001010001011110010
1111011010101101010111110011011011110011001101111000111010011110010010110010101001001011000010010011
0110111100100011010111100000110010111110111011111111011111001000111110110110001010100101000001011000
1001111010010101100101110011110111110100010110011110100100010110111000000001001110011100');
% %m is 384 binary digit

Ncbps=384;
%Ncbps = 115;

% QPSK, rate 3/4


% Ncpc: number of coded bits per subcarrier
%          PICKUP ONLY ONE VALUE:
%1 - BPSK
%Ncpc = 1;
%2 - QPSK
Ncpc = 2;
%4 - 16QAM
%Ncpc = 4;
%6 - 64QAM
%Ncpc = 6;


% %********************Interleaver for OFDM WIMAX*****************
%
% %  k= 0: (Ncbps-1);
%
% %first permutation:
% % m(k) = (Ncbps/12)* kmod12 + floor(k/12)      k=0, ....., Ncbps-1
% for k = 1:(Ncbps-1)
%     kk = k-1;
%     ff = mod(kk,12);
%   %  k = 0:(Ncbps-1);
%     m(k) = (Ncbps*ff/12) + floor(kk/12);
% end
%
% %Second permutation:
% % jk = s.floor(mk /s) + (mk + Ncbps - FLOOR(12*mk/Ncpbs))mod(S)
% %                           k=0, ....., Ncbps-1
% % s = ceil(Ncpc/2);
%
% s = ceil (Ncpc/2);
%
% for k = 1:(Ncbps)
%     %kk = k-1;
%     x = m(k)+ Ncbps - floor(12* m(k)/Ncbps);
```

```
%     mm = mod(x, s);
%     j(k)= s * floor( m(k)/s) + mm;
% end
%
% d_o = j;
%
% xxx = dec_2_hex(j);

%************************De-Interleaver OFDM WIMAX ******************

%                                   data_h                                   =
('77FA4F174E3EE670E8CD3F7690C42CDBF9B7FB436CF19ABDED0A1CD81BEC9B3015BADA31F550497D56EDB4
88CC72FC5C');
% data_d = hex_2_dec(data_h);


%first permutation:
%m(j) = s * floor (j/s) + (j+floor(12 * j/Ncbps))mod(s)
%j=0,1 ....., Ncbps-1

s = ceil (Ncpc/2);

for j = 1:(Ncbps)
   jj = j-1;
   jk = jj + floor(12 * jj /Ncbps);
   jf = mod(jk,s);
  %  k = 0:(Ncbps-1);
   m1(j) = (s * floor(jj/s)) + jf;
end

%Second permutation:
%k(j) = 12 * m(j) - (Ncbps - 1) * floor(12 * m(j)/Ncbps);

for j = 1:(Ncbps)
   k1(j) = 12 * m1(j) - (Ncbps - 1) * floor(12 * m1(j)/Ncbps);
end
```

A.2 16-QAM Interleaver Model:

```
clear
%clc

%aa=
hex_2_dec('3A5EE7AE499E6F1C6FC128BCBDAB57CDBCCDE3A792CA92C24DBC8D7832FBBFDF23ED8A941627A
565CF7D167A45B809CC');
%

% EEC6A1CB7E04736CBC6195D3    :
zzz                                   =
('111011101100011010100001110010110111111000000100011100110110110010111100011000011001010111010011');
% B7C4EF0E4C76CFDC7069B3CE
% DBE0E5B7B54E887DA4AE3130
% EEC6A1CB7E04736CBC6195D3B7C4EF0E4C76CFDC7069B3CEDBE0E5B7B54E887DA4AE3130


%bb= dec_2_bin(aa);
%m                                   =
('898407646983235129903369408620228549208237089767538283785779873984348065286651305271483786817872270
2216228486253004');
%m is 115 digit decimal
%m
=('0011101001011110111001111010111001001001100111100110111100011100011011111100000100101000101111001011
1110110101011010101111100110110111100110011011110001110100111100100101100101010010010110000100100110
0110111100100011010111100000110010111110111011111110111110010001111101101100010101001010000010110001
0011110100101011001011100111101111101000101100111101001000101101110000000100111001100');
%m is 384 binary digit
```

```
%Ncbps = 384;   % QPSK 16 subchannel
%Ncbps = 192;   % BPSK 16 subchannel
%Ncbps = 768;   % 16-QAM 16 subchannel
Ncbps = 96;   %  16-QAM 2 subchannel
%!!!!!Ncbps = 115;

% 16-QAM, rate 3/4 2 subchannel


% Ncpc: number of coded bits per subcarrier
%           PICKUP ONLY ONE VALUE:
%1 - BPSK
%Ncpc = 1;
%2 - QPSK
%Ncpc = 2;
%4 - 16QAM
Ncpc = 4;
%6 - 64QAM
%Ncpc = 6;

%for z = 14:40
z = 12;
display(z);
%  k= 0: (Ncbps-1);

%first permutation:
% m(k) = (Ncbps/12)* kmod12 + floor(k/12)      k=0, ....., Ncbps-1

for k = 0:(Ncbps-1)
    %kk = k-1;
    %ff = mod(kk,z);
  %  k = 0:(Ncbps-1);
    m(k+1) = (Ncbps/z)*mod(k,z) + floor(k/z);
end

%display (m)

%Second permutation:
% jk = s.floor(mk /s) + (mk + Ncbps - FLOOR(12*mk/Ncpbs))mod(S)
%                              k=0, ....., Ncbps-1
% s = ceil(Ncpc/2);

s = ceil (Ncpc/2);

for k = 0:(Ncbps-1)
    %kk = k-1;
%    display(k)
   % x = m(k)+ Ncbps - floor(z* m(k)/Ncbps);
   % mm = mod((m(k)+ Ncbps - floor(z* m(k)/Ncbps)), s);
    j(k+1)= s * floor(m(k+1)/s) + mod((m(k+1)+ Ncbps - floor(z* (m(k+1)/Ncbps))), s);
%    display(j(k))
end

%display(j)
d_o = j;
j = j+1;

for i = 1:Ncbps
    def= j(i);
    gom(i) = zzz(def);
end

%gom;
sprintf('The input is :');
in = bin_2_hex(zzz);
display(in)
sprintf('\n The ouput is :');
out = bin_2_hex(gom);
```

```
display(out)
%end



%xxx = dec_2_hex(j);
```

A.3 QPSK Interleaver Model:

```
%aa=
hex_2_dec('3A5EE7AE499E6F1C6FC128BCBDAB57CDBCCDE3A792CA92C24DBC8D7832FBBFDF23ED8A941627A
565CF7D167A45B809CC');
%bb= dec_2_bin(aa);
%m                                                                                      =
('89840764698323512990336940862022854920823708976753828378577987398434806528665130527148378681787227O
2216228486253004');
%m is 115 digit decimal
%m
=('0011101001011110111001111010111001001001100111100110111100011100011011111100000100101000101111001O
1111011010101101010111110011011011110011001101111000111010011110010010110010101001001011000010010011
0110111100100011010111100000110010111110111011111111011111001000111110110110001010100101000001011000
1001111010010101100101110011110111110100010110011110100100010110111000000001001110011OO');
%m is 384 binary digit

Ncbps=384;
%!!!!!Ncbps = 115;

% QPSK, rate 3/4


% Ncpc: number of coded bits per subcarrier
%          PICKUP ONLY ONE VALUE:
%1 - BPSK
%Ncpc = 1;
%2 - QPSK
Ncpc = 2;
%4 - 16QAM
%Ncpc = 4;
%6 - 64QAM
%Ncpc = 6;


x = 32;

% reading data input file
%A = fread(fid, 2)


%  k= 0: (Ncbps-1);

%first permutation:
% m(k) = (Ncbps/12)* kmod12 + floor(k/12)      k=0, ....., Ncbps-1
for k = 1:(Ncbps)
   %kk= k-1;
  %  ff = mod((k-1),12);
  %  k = 0:(Ncbps-1);
   m(k) = (Ncbps/x)* mod((k-1),x) + floor((k-1)/x);
end
%display (m)
%Second permutation:
% jk = s.floor(mk /s) + (mk + Ncbps - FLOOR(12*mk/Ncpbs))mod(S)
%                              k=0, ....., Ncbps-1
% s = ceil(Ncpc/2);
```

```
s = ceil (Ncpc/2);

for k = 1:(Ncbps)
    %kk = k-1;
%    display(k)
 %   x = m(k)+ Ncbps - floor(12* m(k)/Ncbps);
 %   mm = mod(x, s);
    j(k)= s * floor(m(k)/s) + mod((m(k)+ Ncbps - floor(x* m(k)/Ncbps)), s);
   % display(j(k))
end

display(j)
d_o = j;

%xxx = dec_2_hex(j);
```

APPENDIX B: VERILOG DESIGNS

B.1 Interleaver Design:

```
//--------------------------------------------------------------------------------------------------
//
// Title       : Reconfigurable De-Interleaver
// Design      : Reconfig_deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//--------------------------------------------------------------------------------------------------
//
// Description : Top Level
//
//--------------------------------------------------------------------------------------------------

`timescale 1ns / 1ps
`define  size  1152
`define inter_conf_bits 4
`define conf_bits 5



module top_reconfig_deinterleaver (clk,data_in,en,rst,config_bits,data_out,ready_leaver) ;

// ---- User defined diagram parameters --- //

parameter size=`size;

parameter inter_conf_bits = `inter_conf_bits;

parameter conf_bits = `conf_bits;


// ------------ Port declarations --------- //
input clk;
wire clk;
input data_in;
wire data_in;
input en;
wire en;
input rst;
wire rst;
input [conf_bits-1:0] config_bits;
wire [conf_bits-1:0] config_bits;
output data_out;
wire data_out;
output ready_leaver;
wire ready_leaver;

// ----------- Signal declarations -------- //
wire en_rg1;
wire en_sr2;
wire rg1_ready;
wire rg2_ready;
wire sr1_ready;
wire [inter_conf_bits-1:0] rg1_config;
wire [size-1:0] rg1_to_rg2;
wire [conf_bits-1:0] rg2_config;
wire [size-1:0] rg2_to_sr2;
wire [inter_conf_bits-1:0] sr1_config;
wire [size-1:0] sr1_to_rg1;
wire [inter_conf_bits-1:0] sr2_config;

// -------- Component instantiations -------//


reconfig_controller U_reconfig_controller
(
	.config_bits(config_bits[conf_bits-1:0]),
	.en(en),
	.en_rg1(en_rg1),
	.rg1_config(rg1_config[inter_conf_bits-1:0]),
	.rg2_config(rg2_config[conf_bits-1:0]),
	.rst(rst),
	.sr1_config(sr1_config[inter_conf_bits-1:0]),
	.sr2_config(sr2_config[inter_conf_bits-1:0])
```

```
);


RG1 U_RG1
(
        .clk(clk),
        .rg1_config(rg1_config[inter_conf_bits-1:0]),
        .rg1_ready(rg1_ready),
        .rg1_to_rg2(rg1_to_rg2[size-1:0]),
        .rst(rst),
        .sr1_ready(sr1_ready),
        .sr1_to_rg1(sr1_to_rg1[size-1:0])
);

RG2 U_RG2
(
        .clk(clk),
        .en_sr2(en_sr2),
        .rg1_ready(rg1_ready),
        .rg1_to_rg2(rg1_to_rg2[size-1:0]),
        .rg2_config(rg2_config[conf_bits-1:0]),
        .rg2_ready(rg2_ready),
        .rg2_to_sr2(rg2_to_sr2[size-1:0]),
        .rst(rst)
);

shift_register_in U_shift_register_in
(
        .clk(clk),
        .data_in(data_in),
        .en_rg1(en_rg1),
        .rst(rst),
        .sr1_config(sr1_config[inter_conf_bits-1:0]),
        .sr1_ready(sr1_ready),
        .sr1_to_rg1(sr1_to_rg1[size-1:0])
);



SRout U_SRout
(
        .clk(clk),
        .data_out(data_out),
        .en_sr2(en_sr2),
        .ready_leaver(ready_leaver),
        .rg2_ready(rg2_ready),
        .rg2_to_sr2(rg2_to_sr2[size-1:0]),
        .rst(rst),
        .sr2_config(sr2_config[inter_conf_bits-1:0])
);

endmodule

//-----------------------------------------------------------------------------------------------------
//
// Title       : SRout
// Design      : Reconfig_deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//-----------------------------------------------------------------------------------------------------
//
// File        : d:\My_Designs\Deinterleaver\Reconfig_deinterleaver\src\SRout.v
// Generated   : Thu Nov 23 12:56:41 2006
// From        : interface description file
// By          : Itf2Vhdl ver. 1.20
//
//-----------------------------------------------------------------------------------------------------
//
// Description :
//
//-----------------------------------------------------------------------------------------------------
// History    :
//   Version        Date          Changes
//     0.01       23 Nov 2006     Initial
//     0.11       27 Nov 2006     adding a soft reset "11111"
//
//
//-----------------------------------------------------------------------------------------------------
`timescale 1ps / 1ps

//              Code        Block Size (bits)
//      1       0001        12
//      2       0010        24
//      3       0011        48
//      4       0100        72
//      5       0101        96
//      6       0110        144
//      7       0111        192             new
//      8       1000        288
//      9       1001        384
//      14      1110        576
//      0       0000        768
```

```
//      12              1100             1152
//      15      1111       soft_reset

module SRout ( data_out ,rst ,clk ,ready_leaver, en_sr2 ,rg2_to_sr2 ,sr2_config ,rg2_ready );

parameter size = 1152;   //max size
parameter inter_conf_bits = 4;   //internal  coniguration bits

input rst ;                //reset
wire rst ;
input clk ;                //main clock
wire clk ;
input [size-1:0] rg2_to_sr2 ;//incoming parallel data after shuffling
wire [size-1:0] rg2_to_sr2 ;
input [inter_conf_bits-1:0] sr2_config ;     //config word coming from the controller
wire [inter_conf_bits-1:0] sr2_config ;
input rg2_ready ;            //incoming data is valid to copy
wire rg2_ready ;
input en_sr2;               //switching ON NOTICE: data will start to go out, put
wire en_sr2;                //not valid without ready Interleaver Activated

output data_out ;
reg data_out ;
output ready_leaver ;
reg ready_leaver ;

reg [size:0] temp_reg;   //total = size+1
reg [10:0] count_in ;

always @ (posedge clk)//or negedge rst_n
begin
        if ((rst == 1'b1)||(sr2_config == 4'b1111))
                begin
                        temp_reg      = 'b0;
                        data_out      = 'b0;
                        ready_leaver  = 'b0;
                        count_in = 11'b000_0000_0001;     //now count-up

                end
        else
          begin
    if (en_sr2 == 1'b1)  // to begin added initial o/p of RG2
      begin
           ready_leaver  = 1'b1;
           if (sr2_config == 4'b0001)    //Block Size: 12 [BPSK 1 subchannel]   #######
             begin
                        //temp_reg[size/96:0] =  {temp_reg[(size/96)-1:0],data_in};
                        data_out = temp_reg[size/96];//MSB is the 1st output bit
               if ((count_in != 11'b000_0000_1100) && (rg2_ready == 1'b0))
                 begin
                        count_in = count_in + 1;
                                temp_reg[size/96:0] = {temp_reg[(size/96)-1:0], temp_reg[size/96]};
                 end
                        else
                                begin
                                        temp_reg[(size/96)-1:0] = rg2_to_sr2[(size/96)-1:0];
                                        count_in    = 11'b000_0000_0001;
                   //data_out = tmp[384];                                                  //MSB is the 1st output bit
                                        //sr1_ready   = 1'b1;
                                end
             end
           else if (sr2_config == 4'b1001)//Block Size:384 [QPSK 16 or 16QAM 8 -subch] #######
                 begin
                        data_out = temp_reg[size/3];  //MSB is the 1st output bit
                                if ((count_in != 11'b001_1000_0000) && (rg2_ready == 1'b0))
                   begin
                        count_in = count_in + 1;
                                  temp_reg[size/3:0] = {temp_reg[(size/3)-1:0], temp_reg[size/3]};       //
                   end
                                else
                                        begin
                                                temp_reg[(size/3)-1:0] = rg2_to_sr2[(size/3)-1:0];
                                                count_in    = 11'b000_0000_0001;
                                        end
                 end

           else if (sr2_config == 4'b0000)//Block Size:768 [16-QAM 16 subchannel] #######
                 begin
                        data_out = temp_reg[size/1.5];  //MSB is the 1st output bit
                                if ((count_in != 11'b011_0000_0000) && (rg2_ready == 1'b0))
                   begin
                        count_in = count_in + 1;
                                  temp_reg[size/1.5:0] = {temp_reg[(size/1.5)-1:0], temp_reg[size/1.5]};                    //
                   end
                                else
                                        begin
                                                temp_reg[(size/1.5)-1:0] = rg2_to_sr2[(size/1.5)-1:0];
                                                count_in    = 11'b000_0000_0001;
                                        end
                 end
           else if (sr2_config == 4'b1100)//Block Size:1152 [64-QAM 16subchannel] #######
                 begin
                        data_out = temp_reg[size];  //MSB is the 1st output bit
```

```
                                        if ((count_in != 11'b000_0000_0001)&& (rg2_ready == 1'b0))
                        begin
                                count_in = count_in + 1;
                                        temp_reg[size:0] = {temp_reg[size-1:0], temp_reg[size]};            //
                        end
                                        else
                                                begin
                                                        temp_reg[size-1:0] = rg2_to_sr2[size-1:0];
                                                        count_in    = 11'b000_0000_0001;
                                                end
                    end
            end
        else    ready_leaver  = 1'b0;


    end
end

endmodule



//--------------------------------------------------------------------------------------------------
//
// Title       : TOP_tb
// Design      : Deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//-------------------------------------------------------------------------------------------------
//
// Description : testbench for reconfigurable de-Interleaver
//
//-------------------------------------------------------------------------------------------------
`timescale 1 ns / 1 ps
////////////////////////////////////////////////////////////////////////
//////          External Configuration bits                ///
////////////////////////////////////////////////////////////////////////
//              Configuration Word          Block Size      Modulation Type No. of Subchannels
//              Value           Configuration Word          Block Size      Modulation Type         No. of Subchannels
//1               1               00001                     12                      BPSK            1
//2               2               00010                     24                      BPSK            2
//3               18              10010                     24                      QPSK            1
//4               3               00011                     48                      BPSK            4
//5               11              01011                     48                      QPSK            2
//6               19              10011                     48                      16-QAM          1
//7               4               00100                     72                      64-QAM          1
//8               5               00101                     96                      BPSK            8
//9               13              01101                     96                      QPSK            4       new
//10            21              10101           96                      16-QAM          2
//11            6               00110           144                     64-QAM          2
//12            7               00111           192                     BPSK            16
//13            15              01111           192                     QPSK            8
//14            23              10111           192                     16-QAM          4
//15            8               01000           288                     64-QAM          4
//16            9               01001           384                     QPSK            16
//17            25              11001           384                     16-QAM          8
//18            14              01110           576                     64-QAM          8
//19            18                10000                     768                 16-QAM          16
//20            12                01100             1152                64-QAM          16
//21    31      11111           soft_reset              soft_reset

`define size  1152
`define inter_conf_bits 4
`define conf_bits 5

module top_tb ();


parameter period = 30;      // Set clock period to 100ns
parameter delay  =  50;
parameter width =   16;
parameter tap    =  73;
parameter length  = 1000;       //16 short 16000 long
parameter size    =`size;
parameter inter_conf_bits  = `inter_conf_bits;
parameter conf_bits        = `conf_bits;



   reg              inputdata[0:767]; //767
   reg              out_expected[0:767];  //767
//   reg          y_expected[199:0];//20 inputs
   reg          clk;
   reg          rst, data_in;//, clk;    ,
   reg          en;        // enable
   reg          [4:0] config_bits;
   wire             data_out;
   wire             ready_leaver;

   integer          i ,j ,f1;
```

```
    initial
      fork
        clk <= 0;
        forever #(period/2) clk = !clk;        // creates clock of period "period"
      join


   top_reconfig_deinterleaver deleaver_top(
        .clk(clk),
        .data_in(data_in),
        .en(en),
        .rst(rst),
        .config_bits(config_bits),
        .data_out(data_out),
        .ready_leaver(ready_leaver)) ;


   initial
      begin
//                           enable <= 1'b1;
                          //  cnt = 0;
                            i = 0; j = 0;
          //                enable <=1;
      #0                     rst = 1'b1;
                    #1                      en = 1'b0;
      #(delay)                rst = 1'b1;
      #(period)               rst = 1'b0;
      #1                     config_bits = 5'b01001;//          384             QPSK          16
      #(period)               en = 1'b1;

//     #(period)              enable = 1;
                    assign data_in = inputdata[i];                                    //by me

      #(period*tap*length)


                            $finish;
     end // initial begin

/*
initial
      begin
        $dumpfile("simulation.rtl.vcd");
        $dumpvars;
      end
*/

//   always @(posedge clk)
    initial
      begin
 //        assign enable =1'b1;
////        $readmemh("deinterleaverqpsk16.dat", inputdata);
////        $readmemh("deinterleaverqpsk16_out.dat", out_expected);
//         $readmemh("outputy.dat", y_expected);
//         f1 = $fopen("deinterleaverqpsk16_real_o.dat");
//         f2 = $fopen("routput.y.dat");
////              $display("Deinterleaver        dataNo.          Time   Result Expt  PASS/FAIL");

      end

    always @(posedge clk)

      begin
                 //    $display("clock");

        if ((rst == 1'b0) && (en == 1'b1))
                                                      begin
            //j <=  j + 1; //$display("clock ****");
            if ((i != 'd800) && (ready_leaver == 1'b1))
               begin
                i <= i+1;   //1

                 //  $display("reset and enable");
               if ( data_out == out_expected[i]) //&& (data_out [0]== out_expected[i+1]))
                  begin
                    $fwrite(f1, "%b   %b   PASS\n", data_out, out_expected[i]);//, out_expected[i+1]);
                    $display("Ahmed      %d : %d      %b      %b    PASS   OK\n", i, $time, data_out, out_expected[i]);//, out_expected[i+1]);
                  end
               else
                  begin
                    $fwrite(f1, "%b   %b   FAIL\n", data_out, out_expected[i]);//, out_expected[i+1]);
                    $display("failure    %d : %d      %b      %b   FAIL   X x\n", i, $time, data_out, out_expected[i]);//, out_expected[i+1]);
                  end
      //          if ( y == y_expected[i])
      //             begin
      //               $fwrite(f2, "%b   %b   PASS\n", y, y_expected[i]);
      //               $display("Ahmed Y     %d : %d      %b      %b   PASS   OK ", i, $time, y, y_expected[i]);
      //             end
      //           else
      //             begin
      //               $fwrite(f2, "%b   %b   FAIL\n", y, y_expected[i]);
      //               $display("failed Y    %d : %d      %b      %b   FAIL   X y", i, $time, y, y_expected[i]);
             end
```

```
            end
        end


//assign datain = inputdata[i];
//end
//endmodule // stimulus





endmodule


//-----------------------------------------------------------------------------------------------------
//
// Title       : reconfig_controller
// Design      : Reconfig_deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//-----------------------------------------------------------------------------------------------------
// History    :
//   Version         Date         Changes
//    0.01        24 Nov 2006     Initial
//    0.11        27 Nov 2006     adding a code which will begin used as a soft reset "11111"
//-----------------------------------------------------------------------------------------------------


///////////////////////////////////////////////////////////////////////////////
//////           External Configuration bits                    ///
///////////////////////////////////////////////////////////////////////////////
//            Configuration Word          Block Size     Modulation Type No. of Subchannels
//            Value          Configuration Word          Block Size     Modulation Type          No. of Subchannels
//1            1              00001                      12                    BPSK               1
//2            2              00010                      24                    BPSK               2
//3            18             10010                      24                    QPSK               1
//4            3              00011                      48                    BPSK               4
//5            11             01011                      48                    QPSK               2
//6            19             10011                      48                    16-QAM             1
//7            4              00100                      72                    64-QAM             1
//8            5              00101                      96                    BPSK               8
//9            13             01101                      96                    QPSK               4      new
//10           21           10101         96              16-QAM          2
//11           6            00110         144             64-QAM          2
//12           7            00111         192             BPSK            16
//13           15           01111         192             QPSK            8
//14           23           10111         192             16-QAM          4
//15           8            01000         288             64-QAM          4
//16           9            01001         384             QPSK            16
//17           25           11001         384             16-QAM          8
//18           14           01110         576             64-QAM          8
//19           18             10000                      768                   16-QAM             16
//20           12             01100           1152        64-QAM          16
//21    31     11111          soft_reset          soft_reset

//    1              00001        12       BPSK          1
//    2              00010        24           BPSK          2
//    3              00011        24           QPSK          1
//    4              00100        48           BPSK          4
//    5              00101        48           QPSK          2    old
//    6              00110        48           16-QAM                 1
//    7              00111        72           64-QAM                 1
//    8              01000        96           BPSK          8
//    9              01001        96           QPSK          4
//    10     01010        96      16-QAM            2
//    11     01011        144              64-QAM                 2
//    12     01100        192              BPSK          16
//    13     01101        192              QPSK          8
//    14     01110        192              16-QAM                 4
//    15     01111        288              64-QAM                 4
//    16     10000        384              QPSK          16
//    17     10001        384              16-QAM                 8
//    18     10010        576              64-QAM                 8
//    19     10011        768              16-QAM                 16
//    20     10100        1152      64-QAM           16

/////////////////////////////////////////////////////
///      Internal Config Table           ///
/////////////////////////////////////////////////////

//                Code           Block Size (bits)
//      1         0001           12
//      2         0010           24
//      3         0011           48
//      4         0100           72
//      5         0101           96
//      6         0110           144
//      7         0111           192              new
//      8         1000           288
//      9         1001           384
```

```
//      14              1110            576
//      0               0000            768
//      12              1100            1152
//      15      1111    soft_reset


//                  Code            Block Size (bits)
//      1     0001                 12
//      2     0010                 24
//      3     0011                 48
//      4     0100                 72
//      5     0101                 96                old
//      6     0110                 144
//      7     0111                 192
//      8     1000                 288
//      9     1001                 384
//      10    1010                 576
//      11    1011                 768
//      12    1100                 1152



`timescale 1ps / 1ps
`define  size  1152
`define inter_conf_bits 4
`define conf_bits 5


module reconfig_controller ( rg2_config ,rst ,sr1_config ,en ,en_rg1 ,sr2_config ,rg1_config ,config_bits );

parameter conf_bits = `conf_bits;   // external coniguration bits
parameter inter_conf_bits = `inter_conf_bits;   //internal  coniguration bits

input rst ;
wire rst ;
input en ;
wire en ;
input [conf_bits-1:0] config_bits ;
wire [conf_bits-1:0] config_bits ;

output [conf_bits-1:0] rg2_config ;
reg [conf_bits-1:0] rg2_config ;
output [inter_conf_bits-1:0] sr1_config ;
reg [inter_conf_bits-1:0] sr1_config ;
output [inter_conf_bits-1:0] sr2_config ;
reg [inter_conf_bits-1:0] sr2_config ;
output [inter_conf_bits-1:0] rg1_config ;
reg [inter_conf_bits-1:0] rg1_config ;
output en_rg1 ;
reg en_rg1 ;

always @ ( rst or en or config_bits)    //
                    if (rst == 1'b1)  //reset
                                    begin
                                                    en_rg1     <=  'b0;

        sr1_config <=  'b0;
                                                    rg1_config <=  'b0;
        sr2_config <=  'b0;
        rg2_config <=  'b0;
                                    end
   else
      begin
      if (en == 1'b1)  //enable shift register => enable for Interleaver
         en_rg1 = 1'b1;
      if (config_bits==5'b0_0001) //00001            12          BPSK            1
         begin
            sr1_config =  config_bits[inter_conf_bits-1:0];   //0001
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
            sr2_config =  config_bits[inter_conf_bits-1:0];    //

            rg2_config =  config_bits;
         end
      else if (config_bits==5'b0_0010)// 00010            24          BPSK            2
         begin
            sr1_config =  config_bits[inter_conf_bits-1:0];   //0010
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
            sr2_config =  config_bits[inter_conf_bits-1:0];    //

            rg2_config =  config_bits;
         end
      else if (config_bits==5'b1_0010)//10010            24          QPSK            1
         begin
            sr1_config =  config_bits[inter_conf_bits-1:0];   //0010
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
            sr2_config =  config_bits[inter_conf_bits-1:0];    //

            rg2_config =  config_bits;
         end
      else if (config_bits==5'b0_0011)//00011  0011            48                          BPSK                4
         begin
            sr1_config =  config_bits[inter_conf_bits-1:0];   //0010
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
```

```
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_1011)//01011    0011          48                              QPSK                  2
     begin
        sr1_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};   //0011
                                                    rg1_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};    //
        sr2_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b1_0011)//10011    0011          48                              16-QAM               1
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0010
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_0011)//00100   0100          72                              64-QAM               1
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0100
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_0101)//00101    0101          96                              BPSK                  8
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0101
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_1101)//01101   0101          96                              QPSK                  4
     begin
        sr1_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};   //0101
                                                    rg1_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};    //
        sr2_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b1_0101)//10101    0101          96                              16-QAM               2
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0101
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_0110)//00110    0110          144                             64-QAM               2
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0110
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
   else if (config_bits==5'b0_0111)//00111    0111         192                             BPSK                  16
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0111
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_1111)//01111     0111         192                             QPSK                  8
     begin
        sr1_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};   //0111
                                                    rg1_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};    //
        sr2_config =  {! config_bits[inter_conf_bits-1], config_bits[inter_conf_bits-2:0]};    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b1_0111)//10111     0111         192                             16-QAM               4
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //0111
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b1_0101)//01000   1000          288                             64-QAM               4
     begin
        sr1_config =  config_bits[inter_conf_bits-1:0];   //1000
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
        sr2_config =  config_bits[inter_conf_bits-1:0];    //

        rg2_config =  config_bits;
     end
  else if (config_bits==5'b0_1001)//01001    1001          384                             QPSK                  16
```

```
        begin
           sr1_config =  config_bits[inter_conf_bits-1:0];   //1001
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
           sr2_config =  config_bits[inter_conf_bits-1:0];    //

           rg2_config =  config_bits;
        end
      else if (config_bits==5'b1_1001)//11001  1001           384                  16-QAM          8
        begin
           sr1_config =  config_bits[inter_conf_bits-1:0];   //1001
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
           sr2_config =  config_bits[inter_conf_bits-1:0];    //

           rg2_config =  config_bits;
        end
     else if (config_bits==5'b0_1110)//01110   1110           576                  64-QAM          8
        begin
           sr1_config =  config_bits[inter_conf_bits-1:0];   //1110
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
           sr2_config =  config_bits[inter_conf_bits-1:0];    //

           rg2_config =  config_bits;
        end
     else if (config_bits==5'b1_0000)//    0000               768                  16-QAM          16
        begin
           sr1_config =  config_bits[inter_conf_bits-1:0];   //0000
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
           sr2_config =  config_bits[inter_conf_bits-1:0];    //

           rg2_config =  config_bits;
        end
      else if (config_bits==5'b0_1100)//01100                 1152                 64-QAM          16
        begin
           sr1_config =  config_bits[inter_conf_bits-1:0];   //1100
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
           sr2_config =  config_bits[inter_conf_bits-1:0];    //

           rg2_config =  config_bits;
        end
      else if (config_bits==5'b1_1111)//11111            -----     soft_Reset    -----
        begin
           sr1_config =  config_bits[inter_conf_bits-1:0];   //1100
                                                    rg1_config =  config_bits[inter_conf_bits-1:0];    //
           sr2_config =  config_bits[inter_conf_bits-1:0];    //

           rg2_config =  config_bits;
        end


      end
endmodule


//-------------------------------------------------------------------------------------------------
//
// Title       : RG1
// Design      : Reconfig_deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//-------------------------------------------------------------------------------------------------
//
// Description :    Asynchron Latch
//
//-------------------------------------------------------------------------------------------------

//                    Code           Block Size (bits)
//      1             0001           12
//      2             0010           24
//      3             0011           48
//      4             0100           72
//      5             0101           96
//      6             0110           144
//      7             0111           192             new
//      8             1000           288
//      9             1001           384
//      14            1110           576
//      0             0000           768
//      12            1100           1152
//      15      1111       soft_reset


//                  Code             Block Size (bits)
//       1     0001               12
//       2     0010               24
//       3     0011               48
//       4     0100               72
//       5     0101               96
//       6     0110               144
//       7     0111               192
//       8     1000               288
//       9     1001               384
//       10     1010              576
```

```
//      11    1011                768
//      12    1100                1152


`timescale 1ps / 1ps

`define  size  1152
`define inter_conf_bits 4


module RG1 ( rst ,sr1_to_rg1 ,sr1_ready ,clk ,rg1_config ,rg1_to_rg2 ,rg1_ready );

parameter size = `size;   //max size
parameter inter_conf_bits = `inter_conf_bits;   //internal  coniguration bits

input rst ;                  //reset
wire rst ;
input [size-1:0] sr1_to_rg1 ;  //input parallel block
wire [size-1:0] sr1_to_rg1 ;
input sr1_ready ;             //valid input data indicator
wire sr1_ready ;
input clk ;                  //main clock
wire clk ;
input [inter_conf_bits-1:0] rg1_config ;       //configuration word to define block size used
wire [inter_conf_bits-1:0] rg1_config ;

output [size-1:0] rg1_to_rg2 ; //buffer or latch data to begin used for the next RG2
reg [size-1:0] rg1_to_rg2 ;
output rg1_ready;             //latched data is ready
reg rg1_ready;

//}} End of automatically maintained section
always @ ( rst or sr1_ready or rg1_config)    //sr1_to_rg1 or
                    if ((rst == 1'b1) ||(rg1_config == 4'b1111)) //reset
                                        begin
                                                        rg1_to_rg2 <=  96'h0;
                                                        rg1_ready <=  1'b0;
                                        end
   else if (sr1_ready == 1'b1)
                                        begin
                     if (rg1_config == 4'b0001)    //Block Size: 12 [BPSK 1 subchannel]   #######
              begin
                rg1_to_rg2[(size/96)-1:0] <=  sr1_to_rg1[(size/96)-1:0];
                                        rg1_ready <= 1'b1;
              end
          else if (rg1_config == 4'b1001)//Block Size:384 [QPSK 16 or 16QAM 8 -subch] #######
              begin
                rg1_to_rg2[(size/3)-1:0] <=  sr1_to_rg1[(size/3)-1:0];
                                        rg1_ready <= 1'b1;
              end
          else if (rg1_config == 4'b0000)//Block Size:768 [16-QAM 16 subchannel] #######
              begin
                rg1_to_rg2[(size/1.5)-1:0] <=  sr1_to_rg1[(size/1.5)-1:0];
                                        rg1_ready <= 1'b1;
              end
          else if (rg1_config == 4'b1100)//Block Size:1152 [64-QAM 16subchannel] #######
                                                                                begin
                rg1_to_rg2[(size)-1:0] <=  sr1_to_rg1[(size)-1:0];
                                        rg1_ready <= 1'b1;
              end
          end
                    else  rg1_ready <=  1'b0;
endmodule

//------------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------------
//
// Title       : SRin
// Design      : Reconfig_deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//------------------------------------------------------------------------------------------------
//
// Description :   Input Shift Register
//                counter have been changed for up count for HW reduction reason
//------------------------------------------------------------------------------------------------
`timescale 1ns / 1ps

//                      Code             Block Size (bits)
//      1               0001             12
//      2               0010             24
//      3               0011             48
//      4               0100             72
//      5               0101             96
//      6               0110             144
//      7               0111             192                new
//      8               1000             288
//      9               1001             384
//      14              1110             576
//      0               0000             768
//      12              1100             1152
```

```
//      15      1111      soft_reset

//                  Code              Block Size (bits)
//      1     0001                12
//      2     0010                24
//      3     0011                48
//      4     0100                72
//      5     0101                96         O L D
//      6     0110                144
//      7     0111                192
//      8     1000                288
//      9     1001                384
//      10    1010                576
//      11    1011                768
//      12    1100                1152

`define  size  1152
`define inter_conf_bits 4

module shift_register_in ( data_in ,sr1_to_rg1 ,rst ,clk ,en_rg1 ,sr1_config ,sr1_ready );

parameter size = `size;   //max size
parameter inter_conf_bits = `inter_conf_bits;   //internal  coniguration bits


input data_in ;    //input stream
wire data_in ;
input rst ;       //reset
wire rst ;
input clk ;       //main clock
wire clk ;
input en_rg1 ;    //enable from the controller
wire en_rg1 ;
input [inter_conf_bits-1:0] sr1_config ;  // configuration bits to define the block size ONLY!!
wire [inter_conf_bits-1:0] sr1_config ;

output [size-1:0] sr1_to_rg1 ;    // parallel data block to begin sent to reg1
reg [size-1:0] sr1_to_rg1 ;

output sr1_ready ;              // indicator for output data avaliability
reg sr1_ready;

reg [size:0] temp_reg;   //total = size+1
reg [10:0] count_in ;   //internal counter

always @ (posedge clk)//or posedge rst
begin
            if ((rst == 1'b1) || (sr1_config == 4'b1111))   //1111 is soft reset
                        begin
                                    temp_reg   = 'b0;
                                    sr1_to_rg1 = 'b0;
                                    sr1_ready  = 'b0;
                                    count_in = 11'b000_0000_0001;     //now count-up

                        end
            else
             begin
     if (en_rg1 == 1'b1)
       begin
             if (sr1_config == 4'b0001)    //Block Size: 12 [BPSK 1 subchannel]   #######
            begin
                        temp_reg[size/96:0] =  {temp_reg[(size/96)-1:0],data_in};
                        if (count_in != 11'b000_0000_1100)
                begin
                            count_in = count_in + 1;
                  sr1_ready  = 1'b0;
                end
                        else
                                    begin
                                                sr1_to_rg1      = temp_reg[(size/96)-1:0];
                                                count_in    = 11'b000_0000_0001;
                                                sr1_ready  = 1'b1;
                                    end
             end

           else if (sr1_config == 4'b1001)//Block Size:384 [QPSK 16 or 16QAM 8 -subch] #######
                begin
                                    temp_reg[size/3:0] =  {temp_reg[((size/3)-1):0],data_in};
                                    if (count_in != 11'b001_1000_0000)
                      begin
                            count_in = count_in + 1;
                        sr1_ready  = 1'b0;
                      end
                                    else
                                                begin
                                                            sr1_to_rg1      = temp_reg[((size/3)-1):0];
                                                            count_in    = 11'b000_0000_0001;
                                                            sr1_ready  = 1'b1;
                                                end
                end

           else if (sr1_config == 4'b0000)//Block Size:768 [16-QAM 16 subchannel] #######
```

```
                    begin
                                        temp_reg[size/1.5] =  {temp_reg[(size/1.5)-1:0],data_in};
                                        if (count_in != 11'b011_0000_0000)
                        begin
                                count_in = count_in + 1;
                          sr1_ready  = 1'b0;
                        end
                                        else
                                                    begin
                                                                sr1_to_rg1      = temp_reg[(size/1.5)-1:0];
                                                                count_in    = 11'b000_0000_0001;
                                                                sr1_ready  = 1'b1;
                                                    end
                    end
              else if (sr1_config == 4'b1100)//Block Size:1152 [64-QAM 16subchannel] #######
                    begin
                                        temp_reg =  {temp_reg[size-1:0],data_in};
                                        if (count_in != 11'b000_0000_0001)
                        begin
                                count_in = count_in + 1;
                          sr1_ready  = 1'b0;
                        end
                                        else
                                                    begin
                                                                sr1_to_rg1      = temp_reg[size-1:0];
                                                                count_in    = 11'b100_1000_0000;
                                                                sr1_ready  = 1'b1;
                                                    end
                    end
         end
        else    sr1_ready  = 1'b0;
   end
end

endmodule


//-----------------------------------------------------------------------------------------------
//
// Title       : SRin
// Design      : Reconfig_deinterleaver
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//-----------------------------------------------------------------------------------------------
//
// Description :   Input Shift Register
//               counter have been changed for up count for HW reduction reason
//-----------------------------------------------------------------------------------------------
`timescale 1ns / 1ps

//                  Code            Block Size (bits)
//      1           0001            12
//      2           0010            24
//      3           0011            48
//      4           0100            72
//      5           0101            96
//      6           0110            144
//      7           0111            192             new
//      8           1000            288
//      9           1001            384
//      14          1110            576
//      0           0000            768
//      12          1100            1152
//      15      1111        soft_reset


//                  Code                Block Size (bits)
//      1     0001                  12
//      2     0010                  24
//      3     0011                  48
//      4     0100                  72
//      5     0101                  96          O L D
//      6     0110                  144
//      7     0111                  192
//      8     1000                  288
//      9     1001                  384
//      10    1010                  576
//      11    1011                  768
//      12    1100                  1152

`define  size  1152
`define inter_conf_bits 4

module SRin ( data_in ,sr1_to_rg1 ,rst ,clk ,en_rg1 ,sr1_config ,sr1_ready );

parameter size = `size;   //max size
parameter inter_conf_bits = `inter_conf_bits;   //internal  coniguration bits


input data_in ;    //input stream
wire data_in ;
input rst ;       //reset
```

```
wire rst ;
input clk ;       //main clock
wire clk ;
input en_rg1 ;    //enable from the controller
wire en_rg1 ;
input [inter_conf_bits-1:0] sr1_config ;  // configuration bits to define the block size ONLY!!
wire [inter_conf_bits-1:0] sr1_config ;

output [size-1:0] sr1_to_rg1 ;    // parallel data block to begin sent to reg1
reg [size-1:0] sr1_to_rg1 ;

output sr1_ready ;              // indicator for output data avaliability
reg sr1_ready;

reg [size:0] temp_reg;   //total = size+1
reg [10:0] count_in ;   //internal counter

always @ (posedge clk)//or posedge rst
begin
        if ((rst == 1'b1) || (sr1_config == 4'b1111))   //1111 is soft reset
                begin
                        temp_reg   = 'b0;
                        sr1_to_rg1 = 'b0;
                        sr1_ready  = 'b0;
                        count_in = 11'b000_0000_0001;     //now count-up

                end
        else
         begin
    if (en_rg1 == 1'b1)
      begin
         if (sr1_config == 4'b0001)    //Block Size: 12 [BPSK 1 subchannel]   #######
        begin
                temp_reg[size/96:0] =  {temp_reg[(size/96)-1:0],data_in};
                if (count_in != 11'b000_0000_1100)
           begin
                   count_in = count_in + 1;
              sr1_ready  = 1'b0;
           end
                else
                        begin
                                sr1_to_rg1      = temp_reg[(size/96)-1:0];
                                count_in    = 11'b000_0000_0001;
                                sr1_ready  = 1'b1;
                        end
         end

       else if (sr1_config == 4'b1001)//Block Size:384 [QPSK 16 or 16QAM 8 -subch] #######
           begin
                        temp_reg[size/3:0] =  {temp_reg[((size/3)-1):0],data_in};
                        if (count_in != 11'b001_1000_0000)
              begin
                   count_in = count_in + 1;
                 sr1_ready  = 1'b0;
              end
                        else
                                begin
                                        sr1_to_rg1      = temp_reg[((size/3)-1):0];
                                        count_in    = 11'b000_0000_0001;
                                        sr1_ready  = 1'b1;
                                end
           end

       else if (sr1_config == 4'b0000)//Block Size:768 [16-QAM 16 subchannel] #######
           begin
                        temp_reg[size/1.5] =  {temp_reg[(size/1.5)-1:0],data_in};
                        if (count_in != 11'b011_0000_0000)
              begin
                   count_in = count_in + 1;
                 sr1_ready  = 1'b0;
              end
                        else
                                begin
                                        sr1_to_rg1      = temp_reg[(size/1.5)-1:0];
                                        count_in    = 11'b000_0000_0001;
                                        sr1_ready  = 1'b1;
                                end
           end
      else if (sr1_config == 4'b1100)//Block Size:1152 [64-QAM 16subchannel] #######
           begin
                        temp_reg =  {temp_reg[size-1:0],data_in};
                        if (count_in != 11'b000_0000_0001)
              begin
                   count_in = count_in + 1;
                 sr1_ready  = 1'b0;
              end
                        else
                                begin
                                        sr1_to_rg1      = temp_reg[size-1:0];
                                        count_in    = 11'b100_1000_0000;
                                        sr1_ready  = 1'b1;
                                end
           end
```

```
        end
        else    sr1_ready  = 1'b0;
    end
end

endmodule
```

B.2 GF Reconfigurable Multiplier Design

```
//-----------------------------------------------------------------------------------------------------
//
// Title       : GFmul_core
// Design      : GFpmul
// Author      : Ahmed El-Rayis
// Company     : The University of Edinburgh
//
//-----------------------------------------------------------------------------------------------------
//
// File        : d:\My_Designs\ReedSolomon\GFpmul\src\GFmul_core.v
//
//-----------------------------------------------------------------------------------------------------
//
// Description : GF mul with programmable polynomial for gf(2^8)
//
//-----------------------------------------------------------------------------------------------------
`timescale 1ps / 1ps

/*
module GFmul_core ( a ,b ,c , p);

input [7:0] a ;
wire [7:0] a ;
input [7:0] b ;
wire [7:0] b ;
input [7:0] p;
wire [7:0] p;

output [7:0] c ;
reg [7:0] c ;

reg [14:0] d;

reg [7:0] g8 ;
reg [7:0] g9 ;
reg [7:0] g10 ;
reg [7:0] g11 ;
reg [7:0] g12 ;
reg [7:0] g13 ;
reg [7:0] g14 ;

always @ (a or b or p)// g8, g9, g10, g11, g12, g13, g14
   begin
     //  "d" will begin used as the calculation part (multiplication)
      d[0] = a[0]&b[0];
      d[1] = a[0]&b[1] ^ a[1]&b[0];
      d[2] = a[0]&b[2] ^ a[1]&b[1] ^ a[2]&b[0];
      d[3] = a[0]&b[3] ^ a[1]&b[2] ^ a[2]&b[1] ^ a[3]&b[0];
      d[4] = a[0]&b[4] ^ a[1]&b[3] ^ a[2]&b[2] ^ a[3]&b[1] ^ a[4]&b[0];
      d[5] = a[0]&b[5] ^ a[1]&b[4] ^ a[2]&b[3] ^ a[3]&b[2] ^ a[4]&b[1] ^ a[5]&b[0];
      d[6] = a[0]&b[6] ^ a[1]&b[5] ^ a[2]&b[4] ^ a[3]&b[3] ^ a[4]&b[2] ^ a[5]&b[1] ^ a[6]&b[0];
      d[7] = a[0]&b[7] ^ a[1]&b[6] ^ a[2]&b[5] ^ a[3]&b[4] ^ a[4]&b[3] ^ a[5]&b[2] ^ a[6]&b[1] ^ a[7]&b[0];
      d[8] =            a[1]&b[7] ^ a[2]&b[6] ^ a[3]&b[5] ^ a[4]&b[4] ^ a[5]&b[3] ^ a[6]&b[2] ^ a[7]&b[1];
      d[9] =                        a[2]&b[7] ^ a[3]&b[6] ^ a[4]&b[5] ^ a[5]&b[4] ^ a[6]&b[3] ^ a[7]&b[2];
      d[10]=                                    a[3]&b[7] ^ a[4]&b[6] ^ a[5]&b[5] ^ a[6]&b[4] ^ a[7]&b[3];
      d[11]=                                                a[4]&b[7] ^ a[5]&b[6] ^ a[6]&b[5] ^ a[7]&b[4];
      d[12]=                                                            a[5]&b[7] ^ a[6]&b[6] ^ a[7]&b[5];
      d[13]=                                                                        a[6]&b[7] ^ a[7]&b[6];
      d[14]=                                                                                    a[7]&b[7];

      // this part is used to calculate the galois field generated by
      // provided primitive polynomials p

          g8 = p;
      g9[0] = g8[7]&p[0];
      g9[1] = g8[7]&p[1] ^ g8[0];
      g9[2] = g8[7]&p[2] ^ g8[1];
      g9[3] = g8[7]&p[3] ^ g8[2];
      g9[4] = g8[7]&p[4] ^ g8[3];
      g9[5] = g8[7]&p[5] ^ g8[4];
      g9[6] = g8[7]&p[6] ^ g8[5];
      g9[7] = g8[7]&p[7] ^ g8[6];

      g10[0] = g9[7]&p[0];
      g10[1] = g9[7]&p[1] ^ g9[0];
      g10[2] = g9[7]&p[2] ^ g9[1];
      g10[3] = g9[7]&p[3] ^ g9[2];
      g10[4] = g9[7]&p[4] ^ g9[3];
```

```
      g10[5] = g9[7]&p[5] ^ g9[4];
      g10[6] = g9[7]&p[6] ^ g9[5];
      g10[7] = g9[7]&p[7] ^ g9[6];

      g11[0] = g10[7]&p[0];
      g11[1] = g10[7]&p[1] ^ g10[0];
      g11[2] = g10[7]&p[2] ^ g10[1];
      g11[3] = g10[7]&p[3] ^ g10[2];
      g11[4] = g10[7]&p[4] ^ g10[3];
      g11[5] = g10[7]&p[5] ^ g10[4];
      g11[6] = g10[7]&p[6] ^ g10[5];
      g11[7] = g10[7]&p[7] ^ g10[6];

      g12[0] = g11[7]&p[0];
      g12[1] = g11[7]&p[1] ^ g11[0];
      g12[2] = g11[7]&p[2] ^ g11[1];
      g12[3] = g11[7]&p[3] ^ g11[2];
      g12[4] = g11[7]&p[4] ^ g11[3];
      g12[5] = g11[7]&p[5] ^ g11[4];
      g12[6] = g11[7]&p[6] ^ g11[5];
      g12[7] = g11[7]&p[7] ^ g11[6];

      g13[0] = g12[7]&p[0];
      g13[1] = g12[7]&p[1] ^ g12[0];
      g13[2] = g12[7]&p[2] ^ g12[1];
      g13[3] = g12[7]&p[3] ^ g12[2];
      g13[4] = g12[7]&p[4] ^ g12[3];
      g13[5] = g12[7]&p[5] ^ g12[4];
      g13[6] = g12[7]&p[6] ^ g12[5];
      g13[7] = g12[7]&p[7] ^ g12[6];

      g14[0] = g13[7]&p[0];
      g14[1] = g13[7]&p[1] ^ g13[0];
      g14[2] = g13[7]&p[2] ^ g13[1];
      g14[3] = g13[7]&p[3] ^ g13[2];
      g14[4] = g13[7]&p[4] ^ g13[3];
      g14[5] = g13[7]&p[5] ^ g13[4];
      g14[6] = g13[7]&p[6] ^ g13[5];
      g14[7] = g13[7]&p[7] ^ g13[6];



      // this is the programable part based on the primitive polynomial

      c[0] = d[0] ^ d[8]&g8[0] ^ d[9]&g9[0] ^ d[10]&g10[0] ^ d[11]&g11[0] ^ d[12]&g12[0] ^ d[13]&g13[0] ^ d[14]&g14[0];
      c[1] = d[1] ^ d[8]&g8[1] ^ d[9]&g9[1] ^ d[10]&g10[1] ^ d[11]&g11[1] ^ d[12]&g12[1] ^ d[13]&g13[1] ^ d[14]&g14[1];
      c[2] = d[2] ^ d[8]&g8[2] ^ d[9]&g9[2] ^ d[10]&g10[2] ^ d[11]&g11[2] ^ d[12]&g12[2] ^ d[13]&g13[2] ^ d[14]&g14[2];
      c[3] = d[3] ^ d[8]&g8[3] ^ d[9]&g9[3] ^ d[10]&g10[3] ^ d[11]&g11[3] ^ d[12]&g12[3] ^ d[13]&g13[3] ^ d[14]&g14[3];
      c[4] = d[4] ^ d[8]&g8[4] ^ d[9]&g9[4] ^ d[10]&g10[4] ^ d[11]&g11[4] ^ d[12]&g12[4] ^ d[13]&g13[4] ^ d[14]&g14[4];
      c[5] = d[5] ^ d[8]&g8[5] ^ d[9]&g9[5] ^ d[10]&g10[5] ^ d[11]&g11[5] ^ d[12]&g12[5] ^ d[13]&g13[5] ^ d[14]&g14[5];
      c[6] = d[6] ^ d[8]&g8[6] ^ d[9]&g9[6] ^ d[10]&g10[6] ^ d[11]&g11[6] ^ d[12]&g12[6] ^ d[13]&g13[6] ^ d[14]&g14[6];
      c[7] = d[7] ^ d[8]&g8[7] ^ d[9]&g9[7] ^ d[10]&g10[7] ^ d[11]&g11[7] ^ d[12]&g12[7] ^ d[13]&g13[7] ^ d[14]&g14[7];



   end
endmodule

*/

/*
**************** T E S T    B E N C H ******************
*/
module test();
reg [7:0] a,b;
reg [7:0] p;
wire [7:0] c;

GFmul_core toplevel  (.a(a),.b(b),.p(p),.c(c)); //gfmult (.a(a), .b(b), .c(c))

initial
begin
                $set_toggle_region(toplevel);
                                $toggle_start;

                #10
                p = 29;
   a = 99;
                b = 9;

                #10
   $display("******************************************************************** ");
   $display("**************** T E S T    B E N C H ***************** ");
   $display("******************************************************************** ");
   $display("a used hex = %x, decimal = %d, binary = %b", a, a, a);
   $display("b used hex = %x, decimal = %d, binary = %b", b, b, b);
   $display(" ");
                $display("Polynomial used hex = %x, decimal = %d, binary = %b", p, p, p);
   $display("The calculated results => 92 decimal or 5c hex ");
   $display("Results: hex = %x, decimal = %d ", c, c); // [31:24], c[23:16], c[15:8], c[7:0]);
   $display("mult. 1 : %b or decimal : %d", c[7:0], c[7:0]);
```

```
//$display("mult. 2 : %b or decimal : %d", c[15:8], c[15:8]);
//$display("mult. 3 : %b or decimal : %d", c[23:16], c[23:16]);
//$display("mult. 4 : %b or decimal : %d", c[31:24], c[31:24]);
$display("*********************************************************************** ");
        #20
 p = 29;
 a = 255;
        b = 238;
        #15
$display("*********************************************************************** ");
$display("a used hex = %x, decimal = %d  binary = %b", a, a, a);
$display("b used hex = %x, decimal = %d  binary = %b", b, b, b);
$display(" ");
        $display("Polynomial used hex = %x, decimal = %d, binary = %b", p, p, p);
$display("The calculated results => decimal: 86 or hex: 56  ");
$display("Results: hex = %x, decimal = %d ", c, c);//[31:24], c[23:16], c[15:8], c[7:0]);
$display("mult. 1 : %b or decimal : %d", c, c);
//$display("mult. 2 : %b or decimal : %d", c[15:8], c[15:8]);
//$display("mult. 3 : %b or decimal : %d", c[23:16], c[23:16]);
//$display("mult. 4 : %b or decimal : %d", c[31:24], c[31:24]);
$display("*********************************************************************** ");

 p = 29;
 a = 255;
        b = 212;
        #10
$display("*********************************************************************** ");
$display("a used hex = %x, decimal = %d  binary = %b", a, a, a);
$display("b used hex = %x, decimal = %d  binary = %b", b, b, b);
$display(" ");
        $display("Polynomial used hex = %x, decimal = %d, binary = %b", p, p, p);
$display("The calculated results => decimal: 195 or hex: C3  ");
$display("Results: hex = %x, decimal = %d ", c, c);//[31:24], c[23:16], c[15:8], c[7:0]);
$display("mult. 1 : %b or decimal : %d", c, c);
//$display("mult. 2 : %b or decimal : %d", c[15:8], c[15:8]);
//$display("mult. 3 : %b or decimal : %d", c[23:16], c[23:16]);
//$display("mult. 4 : %b or decimal : %d", c[31:24], c[31:24]);
$display("*********************************************************************** ");


 p = 29;
 a = 0;
        b = 212;
        #10
$display("*********************************************************************** ");
$display("a used hex = %x, decimal = %d  binary = %b", a, a, a);
$display("b used hex = %x, decimal = %d  binary = %b", b, b, b);
$display(" ");
        $display("Polynomial used hex = %x, decimal = %d, binary = %b", p, p, p);
$display("The calculated results => decimal: 0 or hex: 00  ");
$display("Results: hex = %x, decimal = %d ", c, c);//[31:24], c[23:16], c[15:8], c[7:0]);
$display("mult. 1 : %b or decimal : %d", c, c);
//$display("mult. 2 : %b or decimal : %d", c[15:8], c[15:8]);
//$display("mult. 3 : %b or decimal : %d", c[23:16], c[23:16]);
//$display("mult. 4 : %b or decimal : %d", c[31:24], c[31:24]);
$display("*********************************************************************** ");


#100
    $toggle_stop;
            $toggle_report("gfmul_saif.out", 1.0e-9, "test.toplevel");


        $finish;
end
endmodule
```

B.3 RS ENCONDER DESIGN

```
//-----------------------------------------------------------------------------------------------
//
// Title      : rs encoder
// Design     : GFpmul
// Author     : Ahmed El-Rayis
// Company    : The University of Edinburgh
//
//-----------------------------------------------------------------------------------------------
//
// File       : d:\My_Designs\ReedSolomon\GFpmul\src\GFmul_core.v
```

```
//
//-----------------------------------------------------------------------------------------------
//
// Description : RS Encoder
//              [GF mul with programmable polynomial for gf(2^8)]
//
//-----------------------------------------------------------------------------------------------
`timescale 1ps / 1ps

        /*

module rs_encode(datain, enable, p, q0, q1, q2, q3, q4, q5, q6, q7,
            q8, q9, q10, q11, q12, q13, q14, q15, rst, clkin, gin0, gin1, gin2, gin3, gin4, gin5, gin6, gin7, gin8,
gin9, gin10, gin11, gin12, gin13, gin14, gin15);

//input:
input clkin;
input enable;
input rst;
input [7:0] datain;
input [7:0] gin0, gin1, gin2, gin3, gin4, gin5, gin6, gin7, gin8, gin9, gin10,
gin11, gin12, gin13, gin14, gin15;
input [7:0] p; // primitive polynomial

//////////////////
wire [7:0] m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, m14, m15;
wire [7:0] m2;
wire [7:0] m1;
wire [7:0] m0;
wire [7:0] z0;
wire [7:0] z1;
wire [7:0] z2;
wire [7:0] z3, z4, z5, z6, z7, z8, z9, z10, z11, z12, z13, z14, z15;
wire [7:0] bb, fback;
wire clk;

//output:
output [7:0] q0;
output [7:0] q1;
output [7:0] q2;
output [7:0] q3;
output [7:0] q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15;

assign clk = clkin & enable;

//////////////////////////////////////////////////////////////

FF b0(z0, q0, rst, clk);
FF b1(z1, q1, rst, clk);
FF b2(z2, q2, rst, clk);
FF b3(z3, q3, rst, clk);

FF b4(z4, q4, rst, clk);
FF b5(z5, q5, rst, clk);
FF b6(z6, q6, rst, clk);
FF b7(z7, q7, rst, clk);

FF b8(z8, q8, rst, clk);
FF b9(z9, q9, rst, clk);
FF b10(z10, q10, rst, clk);
FF b11(z11, q11, rst, clk);

FF b12(z12, q12, rst, clk);
FF b13(z13, q13, rst, clk);
FF b14(z14, q14, rst, clk);
FF b15(z15, q15, rst, clk);

assign bb = 8'b00000000;
assign z0 = m0; //GFADD a0(bb, m0, z0);
GFADD a1(q0, m1, z1);
```

```
GFADD a2(q1, m2, z2);
GFADD a3(q2, m3, z3);
GFADD a4(q3, m4, z4);
GFADD a5(q4, m5, z5);
GFADD a6(q5, m6, z6);
GFADD a7(q6, m7, z7);
GFADD a8(q7, m8, z8);
GFADD a9(q8, m9, z9);
GFADD a10(q9, m10, z10);
GFADD a11(q10, m11, z11);
GFADD a12(q11, m12, z12);
GFADD a13(q12, m13, z13);
GFADD a14(q13, m14, z14);
GFADD a15(q14, m15, z15);


assign fback = q15 ^ datain;

GFmul_core u0 (fback, gin0, m0, p);
GFmul_core u1 (fback, gin1, m1, p);
GFmul_core u2 (fback, gin2, m2, p);
GFmul_core u3 (fback, gin3, m3, p);

GFmul_core u4 (fback, gin4, m4, p);
GFmul_core u5 (fback, gin5, m5, p);
GFmul_core u6 (fback, gin6, m6, p);
GFmul_core u7 (fback, gin7, m7, p);

GFmul_core u8 (fback, gin8, m8, p);
GFmul_core u9 (fback, gin9, m9, p);
GFmul_core u10(fback, gin10, m10, p);
GFmul_core u11(fback, gin11, m11, p);

GFmul_core u12(fback, gin12, m12, p);
GFmul_core u13(fback, gin13, m13, p);
GFmul_core u14(fback, gin14, m14, p);
GFmul_core u15(fback, gin15, m15, p);

endmodule


module GFADD(in1, in2, out);
  input [7:0] in1;
  input [7:0] in2;
  output [7:0] out;
  assign out = in1^in2;
endmodule


module FF(d, q, rst, clk);
  input [7:0] d;
  input  clk;
  output  [7:0] q;
  reg [7:0] out;
  input rst;
  always @(posedge clk or rst)
     if(rst) out <= 8'b00000000; else
     begin
      out <= #1 d;
     end
     assign q = out;
endmodule


////////////////////////programmable mul//////////////////////////////////

module GFmul_core ( a ,b ,c , p);

input [7:0] a ;
```

```
wire [7:0] a ;
input [7:0] b ;
wire [7:0] b ;
input [7:0] p;
wire [7:0] p;

output [7:0] c ;
reg [7:0] c ;

reg [14:0] d;

reg [7:0] g8 ;
reg [7:0] g9 ;
reg [7:0] g10 ;
reg [7:0] g11 ;
reg [7:0] g12 ;
reg [7:0] g13 ;
reg [7:0] g14 ;

always @ (a or b or p)// g8, g9, g10, g11, g12, g13, g14
  begin
    //  "d" will begin used as the calculation part (multiplication)
     d[0] = a[0]&b[0];
     d[1] = a[0]&b[1] ^ a[1]&b[0];
     d[2] = a[0]&b[2] ^ a[1]&b[1] ^ a[2]&b[0];
     d[3] = a[0]&b[3] ^ a[1]&b[2] ^ a[2]&b[1] ^ a[3]&b[0];
     d[4] = a[0]&b[4] ^ a[1]&b[3] ^ a[2]&b[2] ^ a[3]&b[1] ^ a[4]&b[0];
     d[5] = a[0]&b[5] ^ a[1]&b[4] ^ a[2]&b[3] ^ a[3]&b[2] ^ a[4]&b[1] ^ a[5]&b[0];
     d[6] = a[0]&b[6] ^ a[1]&b[5] ^ a[2]&b[4] ^ a[3]&b[3] ^ a[4]&b[2] ^ a[5]&b[1] ^ a[6]&b[0];
     d[7] = a[0]&b[7] ^ a[1]&b[6] ^ a[2]&b[5] ^ a[3]&b[4] ^ a[4]&b[3] ^ a[5]&b[2] ^ a[6]&b[1] ^ a[7]&b[0];
     d[8] =            a[1]&b[7] ^ a[2]&b[6] ^ a[3]&b[5] ^ a[4]&b[4] ^ a[5]&b[3] ^ a[6]&b[2] ^ a[7]&b[1];
     d[9] =                        a[2]&b[7] ^ a[3]&b[6] ^ a[4]&b[5] ^ a[5]&b[4] ^ a[6]&b[3] ^ a[7]&b[2];
     d[10]=                                    a[3]&b[7] ^ a[4]&b[6] ^ a[5]&b[5] ^ a[6]&b[4] ^ a[7]&b[3];
     d[11]=                                                a[4]&b[7] ^ a[5]&b[6] ^ a[6]&b[5] ^ a[7]&b[4];
     d[12]=                                                            a[5]&b[7] ^ a[6]&b[6] ^ a[7]&b[5];
     d[13]=                                                                        a[6]&b[7] ^ a[7]&b[6];
     d[14]=                                                                                    a[7]&b[7];

     // this part is used to calculate the galois field generated by
     // provided primitive polynomials p

          g8 = p;
     g9[0] = g8[7]&p[0];
     g9[1] = g8[7]&p[1] ^ g8[0];
     g9[2] = g8[7]&p[2] ^ g8[1];
     g9[3] = g8[7]&p[3] ^ g8[2];
     g9[4] = g8[7]&p[4] ^ g8[3];
     g9[5] = g8[7]&p[5] ^ g8[4];
     g9[6] = g8[7]&p[6] ^ g8[5];
     g9[7] = g8[7]&p[7] ^ g8[6];

     g10[0] = g9[7]&p[0];
     g10[1] = g9[7]&p[1] ^ g9[0];
     g10[2] = g9[7]&p[2] ^ g9[1];
     g10[3] = g9[7]&p[3] ^ g9[2];
     g10[4] = g9[7]&p[4] ^ g9[3];
     g10[5] = g9[7]&p[5] ^ g9[4];
     g10[6] = g9[7]&p[6] ^ g9[5];
     g10[7] = g9[7]&p[7] ^ g9[6];

     g11[0] = g10[7]&p[0];
     g11[1] = g10[7]&p[1] ^ g10[0];
     g11[2] = g10[7]&p[2] ^ g10[1];
     g11[3] = g10[7]&p[3] ^ g10[2];
     g11[4] = g10[7]&p[4] ^ g10[3];
     g11[5] = g10[7]&p[5] ^ g10[4];
     g11[6] = g10[7]&p[6] ^ g10[5];
     g11[7] = g10[7]&p[7] ^ g10[6];

     g12[0] = g11[7]&p[0];
```

```
      g12[1] = g11[7]&p[1] ^ g11[0];
      g12[2] = g11[7]&p[2] ^ g11[1];
      g12[3] = g11[7]&p[3] ^ g11[2];
      g12[4] = g11[7]&p[4] ^ g11[3];
      g12[5] = g11[7]&p[5] ^ g11[4];
      g12[6] = g11[7]&p[6] ^ g11[5];
      g12[7] = g11[7]&p[7] ^ g11[6];

      g13[0] = g12[7]&p[0];
      g13[1] = g12[7]&p[1] ^ g12[0];
      g13[2] = g12[7]&p[2] ^ g12[1];
      g13[3] = g12[7]&p[3] ^ g12[2];
      g13[4] = g12[7]&p[4] ^ g12[3];
      g13[5] = g12[7]&p[5] ^ g12[4];
      g13[6] = g12[7]&p[6] ^ g12[5];
      g13[7] = g12[7]&p[7] ^ g12[6];

      g14[0] = g13[7]&p[0];
      g14[1] = g13[7]&p[1] ^ g13[0];
      g14[2] = g13[7]&p[2] ^ g13[1];
      g14[3] = g13[7]&p[3] ^ g13[2];
      g14[4] = g13[7]&p[4] ^ g13[3];
      g14[5] = g13[7]&p[5] ^ g13[4];
      g14[6] = g13[7]&p[6] ^ g13[5];
      g14[7] = g13[7]&p[7] ^ g13[6];

      // this is the programable part based on the primitive polynomial

      c[0] = d[0] ^ d[8]&g8[0] ^ d[9]&g9[0] ^ d[10]&g10[0] ^ d[11]&g11[0] ^ d[12]&g12[0] ^ d[13]&g13[0] ^
d[14]&g14[0];
      c[1] = d[1] ^ d[8]&g8[1] ^ d[9]&g9[1] ^ d[10]&g10[1] ^ d[11]&g11[1] ^ d[12]&g12[1] ^ d[13]&g13[1] ^
d[14]&g14[1];
      c[2] = d[2] ^ d[8]&g8[2] ^ d[9]&g9[2] ^ d[10]&g10[2] ^ d[11]&g11[2] ^ d[12]&g12[2] ^ d[13]&g13[2] ^
d[14]&g14[2];
      c[3] = d[3] ^ d[8]&g8[3] ^ d[9]&g9[3] ^ d[10]&g10[3] ^ d[11]&g11[3] ^ d[12]&g12[3] ^ d[13]&g13[3] ^
d[14]&g14[3];
      c[4] = d[4] ^ d[8]&g8[4] ^ d[9]&g9[4] ^ d[10]&g10[4] ^ d[11]&g11[4] ^ d[12]&g12[4] ^ d[13]&g13[4] ^
d[14]&g14[4];
      c[5] = d[5] ^ d[8]&g8[5] ^ d[9]&g9[5] ^ d[10]&g10[5] ^ d[11]&g11[5] ^ d[12]&g12[5] ^ d[13]&g13[5] ^
d[14]&g14[5];
      c[6] = d[6] ^ d[8]&g8[6] ^ d[9]&g9[6] ^ d[10]&g10[6] ^ d[11]&g11[6] ^ d[12]&g12[6] ^ d[13]&g13[6] ^
d[14]&g14[6];
      c[7] = d[7] ^ d[8]&g8[7] ^ d[9]&g9[7] ^ d[10]&g10[7] ^ d[11]&g11[7] ^ d[12]&g12[7] ^ d[13]&g13[7] ^
d[14]&g14[7];



   end
endmodule



  */

/*
**************** T E S T   B E N C H ******************
*/


module test();


reg     [7:0] inputdata [0:238]; //767
reg     [7:0] out_expected[0:15];  //767
reg     clk;
```

```
reg      rst;

wire [7:0] q0;
wire [7:0] q1;
wire [7:0] q2;
wire [7:0] q3;
wire [7:0] q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15;


parameter period = 2;     // Set clock period 10nsto 100MHz  2ns  500MHz
parameter delay  =  100;
parameter width =   16;
parameter tap    =  73;
parameter length  = 1000;      //16 short 16000 long


reg clkin;
reg enable;
//reg rst;
reg [7:0] datain;
reg [7:0] p; // primitive polynomial
reg [7:0] data_out [0:15];
reg [7:0] gin0, gin1, gin2, gin3, gin4, gin5, gin6, gin7, gin8,
gin9, gin10, gin11, gin12, gin13, gin14, gin15;

integer  i ,j ,f1;



rs_encode toplevel ( .datain(datain),
                .enable(enable),
                .p(p),
                .q0(q0), .q1(q1), .q2(q2), .q3(q3), .q4(q4), .q5(q5), .q6(q6), .q7(q7),
                .q8(q8), .q9(q9), .q10(q10), .q11(q11), .q12(q12), .q13(q13), .q14(q14), .q15(q15),
                .rst(rst),
                .clkin(clkin), .gin0(gin0), .gin1(gin1), .gin2(gin2), .gin3(gin3), .gin4(gin4), .gin5(gin5),
                .gin6(gin6), .gin7(gin7), .gin8(gin8), .gin9(gin9), .gin10(gin10), .gin11(gin11), .gin12(gin12),
                .gin13(gin13), .gin14(gin14), .gin15(gin15));


  initial
    fork
      clkin <= 0;
      forever #(period/2) clkin = !clkin;       // creates clock of period "period"
    join
     //   always @(posedge clk)
  initial
    begin
      $readmemb("rs_encoder_input_239.dat", inputdata); //deinterleaver_BPSK1_12ch.dat
      $readmemb("rs_encoder_output_16_calc.dat", out_expected);
      f1 = $fopen("rs_encoder_output_16_real.dat");
     $display(" ***************** R S  E n c o d e r with G F M U L - S i n g l e *********");
   end

initial
begin
     $set_toggle_region(toplevel);
               $toggle_start;

             #10
             p = 29;
      gin0 = 'd79 ;
      gin1 = 'd44 ;
      gin2 = 'd81 ;
      gin3 = 'd100;
      gin4 = 'd49 ;
      gin5 = 'd183;
      gin6 = 'd56 ;
      gin7 = 'd17 ;
      gin8 = 'd232;
```

```
      gin9 = 'd187;
      gin10= 'd126;
      gin11= 'd104;
      gin12= 'd31 ;
      gin13= 'd103;
      gin14= 'd52 ;
      gin15= 'd118;
              i = 'b0;
              j = 'b0;

        //datain = 99;
     enable = 0;


              ///////////////////////////////adjust =>

     #10
     #10                    rst = 1'b0; i = 0; j = 0;
               #50                    enable = 1'b1;
     #(delay)                 rst = 1'b1;
     #(period)                rst = 1'b0;
     #10                    // config_bits = 5'b00101;//12//0_1001;//                        384 QPSK 16
     #(period)               // en = 1'b1;             //0_0001           12  BPSK  1
    // assign datain = {inputdata[j]};  //:j+1];                                   //by me

                          //assign data_in = inputdata[
               #600
//     #(period*tap*length)
//               #(period) enable=1'b0;
               #(period) rst=1'b1;
//               #(period)
              $toggle_stop;
                  $toggle_report("rsenc_saif.out", 1.0e-9, "test.toplevel");

      $finish;

    end // initial begin

/*
initial
     begin
       $dumpfile("simulation.rtl.vcd");
       $dumpvars;
     end
*/



   always @(posedge clkin)
     begin
       if ((rst == 1'b0) && (enable == 1'b1))
                                     begin
            if (j <239)
               begin
                  datain = {inputdata[j]};
                  $display("inputdata[%d] = %d",j, inputdata[j]);
               end
            if ((j == 239))
               begin
             //  i <= i+1;   //for output data

            $display("****************************************************************** ");
            $display("**************** T E S T    B E N C H ***************** ");
            $display("****************************************************************** ");
              data_out [15] = q15;
              data_out [14] = q14;
              data_out [13] = q13;
              data_out [12] = q12;
              data_out [11] = q11;
              data_out [10] = q10;
```

```
          data_out [ 9] = q9;
          data_out [ 8] = q8;
          data_out [ 7] = q7;
          data_out [ 6] = q6;
          data_out [ 5] = q5;
          data_out [ 4] = q4;
          data_out [ 3] = q3;
          data_out [ 2] = q2;
          data_out [ 1] = q1;
          data_out [ 0] = q0;
        end
        if ((i < 'd16) && (j >= 'd239))
          begin
            if ((data_out[i] == out_expected[i]))
              begin
                $fwrite(f1, "%b   %b  PASS\n", data_out[i], out_expected[i]);
                $display("Ahmed       %d-%d  :  %d        O:  %b        Exp:  %b      PASS     OK\n",  i,j,  $time,  data_out[i],
out_expected[i]);
              end
            else
              begin
                $fwrite(f1, "%b   %b  FAIL\n", data_out[i], out_expected[i]);
                $display("failure      %d-%d  :  %d        O:  %b        Exp:  %b      FAIL     X  x\n",  i,j,  $time,  data_out[i],
out_expected[i]);
              end
            i <= i+1;   //for output data
          end
       j <=  j + 1; //for input data

          end

 end

endmodule
```