Durham University

Durham E-Theses

Just-in-time Hardware generation for abstracted reconfigurable computing

Grocutt, Thomas Christopher

How to cite:

Grocutt, Thomas Christopher (2005) *Just-in-time Hardware generation for abstracted reconfigurable computing*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/2704/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Just-In-Time Hardware Generation For Abstracted Reconfigurable Computing

Thomas Christopher Grocutt

The copyright of this thesis rests with the author or the university to which it was submitted. No quotation from it, or information derived from it may be published without the prior written consent of the author or university, and any information derived from it should be acknowledged.

A Thesis presented for the degree of
Doctor of Philosophy

Durham University

Centre for Electronic Systems

School of Engineering

University of Durham

England

2005

11 OCT 2006

Declaration

The work in this thesis is based on research carried out in the Centre for Electronic Systems, School of Engineering, University of Durham, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless otherwise referenced to the contrary in the text.

Copyright © 2005 by Thomas Christopher Grocutt.

The copyright of this thesis rests with the author. No quotation from it should be published in any format, including electronic and the Internet, without the author's prior written consent. All information derived from this thesis must be acknowledged appropriately.

Acknowledgements

I would like to thank my project supervisor, Simon Johnson, for his time, help and encouragement; the staff in the engineering department for their continued help and the AIG group in the department of physics for allowing me to use their Cray XD1. In particular, I would like to thank my father for his patience whilst proof reading this thesis. I would also like to thank the open source programming team that created LaTeX and thus saving me from Word.

Abstract

This thesis addresses the use of reconfigurable hardware in computing platforms, in order to harness the performance benefits of dedicated hardware whilst maintaining the flexibility associated with software. Although the reconfigurable computing concept is not new, the low level nature of the supporting tools normally used, together with the consequent limited level of abstraction and resultant lack of backwards compatibility, has prevented the widespread adoption of this technology. In addition, bandwidth and architectural limitations, have seriously constrained the potential improvements in performance. A review of existing approaches and tools flows is conducted to highlight the current problems being faced in this field.

The objective of the work presented in this thesis is to introduce a radically new approach to reconfigurable computing tool flows. The runtime based tool flow introduces complete abstraction between the application developer and the underlying hardware. This new technique eliminates the ease of use and backwards compatibility issues that have plagued the reconfigurable computing concept, and could pave the way for viable mainstream reconfigurable computing platforms. An easy to use, cycle accurate behavioural modelling system is also presented, which was used extensively during the early exploration of new concepts and architectures. Some performance improvements produced by the new reconfigurable computing tool flow, when applied to both a MIPS based embedded platform, and the Cray XD1, are also presented. These results are then analyzed and the hardware and software factors

affecting the performance increases that were obtained are discussed, together with potential techniques that could be used to further increase the performance of the system.

Lastly a heterogenous computing concept is proposed, in which, a computer system, containing multiple types of computational resource is envisaged, each having their own strengths and weaknesses (e.g. DSPs, CPUs, FPGAs). A revolutionary new method of fully exploiting the potential of such a system, whilst maintaining scalability, backwards compatibility, and ease of use is also presented.

Contents

Acknowledgements iii

Abstract iv

List of Figures xvi

List of Tables xix

List of Listings xx

Glossary 1

1 Introduction 4

1.1 Conventional Computing Architectures 4

1.1.1 Instruction Set Expansion 4

1.1.2 Dedicated Hardware 5

1.2 Reconfigurable Computing 6

1.2.1 Hardware Implementations 7

1.2.1.1 Direct Processor Integration 7

1.2.1.2 External RC Area 9

1.2.2 Tools Flows 11

1.2.2.1 Separate SW And HW Tool Flows 11

1.2.2.2 Common High Level Language 12

1.2.2.3 Low Level Tool Flows 15

1.2.3 Scheduling 16

1.2.4 Floating Point Maths 17

1.3 Research Conducted 18

1.4 Warp Processing 20

1.4.1 Hardware Platforms 20

1.4.2 Tool Flow 21

1.4.3 Performance Improvements 22

1.5 Summary 23

2 Behavioural Simulation **25**

2.1 Background 25

2.2 CPUSim 27

2.2.1 Concurrency 27

2.2.2 Flexibility 30

2.2.3 Instantiation and Connection 32

2.2.4 Visualisation and Statistics 34

2.3 Summary 35

3 EPIC Simulation **37**

3.1 CPU Architecture 37

3.1.1 Additional RC Hardware Blocks 38

3.1.1.1 Code Profiler 39

3.1.1.2 Reconfigurable Execution Unit 39

3.2 RC Conversion Algorithms 40

3.2.1 Instruction Combination 40

3.2.2 Loop Conversion 43

3.2.2.1 Data Flow Pipeline 44

3.3 Performance evaluation 46

3.3.1 Test Algorithms 46

3.3.1.1 Copy algorithm 47

3.3.1.2 Half Brightness Algorithm 47

3.3.1.3 Mandelbrot Algorithm 47

3.3.2 Loop Dependencies 47

3.4 Results 48

3.4.1 Copy algorithm 48

3.4.2 Half Brightness Algorithm 49

3.4.3 Mandelbrot Algorithm 51

3.4.4 HW Pipeline Implementation 54

3.4.5 Summary 57

4 Loop Conversion **58**

4.1 Abstract Instruction Model 60

4.1.1 Supporting Additional ISAs 61

4.2 Targeting The Hardware Pipeline 62

4.3 Loop Identification 63

4.4 Instruction Linearization 64

4.4.1 MUX Insertion 65

4.4.2 Instruction Guarding 66

4.5 Optimization 66

4.5.1 Hardware Dependency Removal 67

4.5.2 Stack Removal 67

4.5.3 Iteration Dependency Removal 68

4.5.4 Instruction Removal 69

4.5.5 Tree Re-balancing 70

4.6 Pipeline Generation 71

4.6.1 Operation Scheduling 71

4.6.1.1 Pointer Aliasing 73

4.6.2 Data Forwarder Addition 74

4.6.3 Register Remapping 74

4.7 Target Implementation 75

4.7.1 Pipeline configuration generation 75

4.7.2 Program modification 76

5 MIPS Test Platform **78**

5.1 Platform Details 78

5.1.1 RC Area Integration 79

5.1.2 Hardware 80

5.1.2.1 CPU/RC Area Interface 84

5.2 Software 86

5.2.1 Console Software 86

5.2.2 Data Transfer Software 87

5.3 Test Algorithms 87

5.3.1 PRBS Generator 88

5.3.2 FFT 91

5.3.3 Low Pass Filter 91

5.3.4 Normalization 92

5.3.5 Block Search 92

5.3.6 Mandelbrot 94

5.3.7 Half Brightness 94

5.3.8 Factorial and Series Sum 95

5.3.9 Copy 96

5.3.10 Sort 96

5.4 Performance Scalability 97

5.4.1 Bandwidth 97

5.4.2 Parallelism 98

5.4.3 Algorithm Complexity 100

5.4.3.1 Increased section size 101

5.4.3.2 Increased complexity 102

5.5 Platform Evaluation 103

5.5.1 Abstraction 103

5.5.2 Automatic conversion 103

5.5.3 Low conversion time 104

5.5.4 Large performance increase 104

5.6 Summary 104

6 Cray XD1 Platform **106**

6.1 Cray XD1 Overview 106

6.2 Platform Details 109

6.2.1 Execution 109

6.2.2 Tool flow 110

6.2.3 Memory Access 111

6.3 Performance Evaluation 113

6.3.1 PRBS Generator 114

6.3.2 Half Brightness 116

6.3.3 Low Pass Filter 117

6.3.4 Normalization 117

6.3.5 Copy 119

6.3.6 Series Sum 120

6.4 MIPS RC Platform Comparison 120

6.5 Summary 122

6.5.1 Cray XD1 Platform Limitations 122

6.5.2 Clock Speeds 123

6.5.3 Performance Improvements 124

7 Optimisations Of The Reconfigurable Computing System **125**

7.1 Hardware Conversion Tools 125

7.1.1 Loop Extraction 125

7.1.2 Floating Point Operations 127

7.1.3 Optimization 127

7.1.4 Scheduling 128

7.1.4.1 Operation Variants 128

7.1.4.2 Local Feedback 129

7.1.4.3 DMA Operation Scheduling 130

7.1.4.4 FPGA Tool Integration 130

7.1.5 Hardware Software Integration 131

7.2 Platforms 132

7.2.1 MIPS 132

7.2.2 XD1 133

7.2.3 Benchmark Algorithms 134

7.3 An Ideal Reconfigurable Computing Platform 135

7.3.1 Processor Integration 135

7.3.2 Homogeneous RC Area 137

7.3.2.1 Partial Reconfigurability 138

7.3.2.2 Homogeneous Structure 138

7.3.2.3 Configuration Controller 139

7.3.2.4 Specialized Hardware 139

7.3.2.5 Design For Place And Route 140

7.3.2.6 Clock Domains 140

7.3.3 Memory Sub-System 141

7.3.4 Code Profiling 142

7.3.5 Hardware Scheduling 143

7.4 Heterogeneous Computing 144

7.5 Future Research 147

7.6 Summary 148

8 Conclusion **150**

Bibliography **156**

A MIPS Test Algorithms Source Code **172**

A.1 PRBS Generator (Standard) 172

A.2 PRBS Generator (Unrolled) 173

A.3 FFT 175

A.4 Low Pass Filter 178

A.5 Normalization 180

A.6 Block Search (Planar) 182

A.7 Block Search (Packed) 185

A.8 Mandelbrot 188

A.9 Half Brightness 189

A.10 Factorial and Series Sum 190

A.11 Copy 190

A.12 Sort 191

List of Figures

1.1 Separate SW and HW tool chains 11

1.2 Common high level language 13

1.3 Low level tool chain 15

1.4 Warp processor architecture 21

2.1 Single function clocking 29

2.2 Dual function clocking 29

2.3 CPUSim simulating an experimental processor, showing the contents of the instruction cache and a Mandelbrot fractal generated by software running on the simulated processor. 35

3.1 EPIC CPU core with additional RC blocks shown in red 38

3.2 EPIC CPU RC tool flow 41

3.3 Example RC data flow pipeline 44

3.4 Performance improvement of copy algorithm with loop extraction . . 50

3.5 Performance improvement of half brightness algorithm with instruction combination 52

3.6 Performance improvement of half brightness algorithm with loop extraction 53

3.7 Performance improvement of Mandelbrot algorithm with instruction combination 55

3.8 Performance improvement of Mandelbrot algorithm with loop extraction 55

3.9 FPGA tiles used to implement real hardware data flow pipeline . . . 56

4.1 Simplified Loop conversion block diagram 59

4.2 Example operation scheduling onto data flow pipeline 72

4.3 Loop constant and stage 0 shift register arrangement 75

5.1 FPGA tiles used to implement MIPS CPU, RC pipeline, and peripherals 82

5.2 MIPS system block diagram 83

5.3 Example logic analyzer trace showing RC pipeline execution 84

5.4 USB data transfer application displaying the contents of the data buffer as an image 87

5.5 Logic analyzer trace showing RC pipeline execution of FFT algorithm 91

5.6 Pixel graphics formats 93

5.7 Logic analyzer trace showing RC pipeline execution of half brightness algorithm 95

5.8 Logic analyzer trace showing RC pipeline execution of quick sort algorithm 97

5.9 Generalized effects of parallelism on performance (Software vs Hardware) 101

5.10 Picture processing and compression 102

6.1 Cray XD1 blade architecture 108

6.2 Cray FPGA interface cores 108

6.3 Hardware conversion tool flow for Cray XD1 111

6.4 Cross bar architecture for QDR memory interface 112

6.5 Operations on the hardware data flow pipeline for the low pass filter algorithm 118

7.1 Example data flow pipelines with and without the local feedback optimization 129

7.2 Block diagram of idealized CPU architecture 136

List of Tables

1.1 Warp processing test platforms 23

3.1 EPIC CPU specification 38

3.2 Test algorithms and characteristics 46

3.3 Test cases implemented in FPGA hardware 57

5.1 MIPS platform summary 82

5.2 Test algorithms used on MIPS test platform 89

6.1 Test algorithms used on the Cray XD1 115

6.2 Predicted performance improvement after resolving current Cray XD1 limitations 115

List of Listings

3.1 Sample code before instruction combination 42

3.2 Sample code after instruction combination code 42

3.3 Example code loop with dependency 48

3.4 Example code loop without dependency 48

4.1 If-else statement implemented with branches 65

4.2 If-else statement implemented with MUXs 65

4.3 Conditional store implemented with branching 66

4.4 Conditional store implemented with guarding 66

4.5 Hardware dependency present 67

4.6 Hardware dependency removed 67

4.7 Stacking ("push" first) 68

4.8 Stack operations removed ("push" first) 68

4.9 Stacking ("pop" first) 69

4.10 Stack operations removed ("pop" first) 69

4.11 Before instruction removal 70

4.12 After instruction removal 70

4.13 Sequential value combination 70

4.14 Balanced value combination 70

4.15 Program before trigger instruction insertion 77

4.16 Program after trigger instruction insertion 77

5.1 Example RC trigger instruction sequence 80

7.1 Example code with multiple nested loops 126

Glossary

AIM	Abstract Instruction Model
AMD	Advanced Micro Devices
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BSD	Berkeley Software Distribution
CISC	Complex Instruction Set Computer
CMS	Code Morphing Software
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processing
EPIC	Explicitly Parallel Instruction Computing
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GNU	GNU's Not UNIX
GUI	Graphical User Interface
HDL	Hardware Description Language
HPC	High Performance Computing

HSI	Hardware Software Interface
HW	Hardware
IEEE	Institute of Electrical & Electronic Engineers
ILP	Instruction Level Parallelism
IO	Input/Output
IP	Intellectual Property
ISA	Instruction Set Architecture
JIT	Just In Time
JPEG	Joint Photographic Experts Group
LFSR	Linear Feedback Shift Register
LOG	Logarithm
LPF	Low Pass Filters
LRU	Least Recently Used
LUT	Look Up Table
MAC	Multiply and Accumulate
MMU	Memory Management Unit
MMX	Multi-Media Extensions
MPEG	Motion Pictures Experts Group
MUX	Multiplexer
NRE	Non-Recurring Expenditure
OS	Operating System
PAR	Place And Route
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PNG	Portable Network Graphics
PRBS	Pseudo Random Binary Sequence
QDR	Quad Data Rate

RAM	Random Access Memory
RLE	Run Length Encoding
ROM	Read Only Memory
RC	Reconfigurable Computing
RGB	Red Green and Blue
RS232	Serial Interface
SAD	Sum of Absolute Differences
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multi-Threading
SoC	Systems On a Chip
SDRAM	Synchronous Dynamic RAM
SRAM	Static RAM
SW	Software
TSF	Technology Scale Factor
TV	Television
USB	Universal Serial Bus
WAV	Waveform Audio
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
VU	Vertical Unit
YUV	Y = Luminance, U = Normalised BY, V = Normalised RY

Chapter 1

Introduction

1.1 Conventional Computing Architectures

Since the birth of the "Von Neumann architecture" [1] the performance of computers has continued to increase. In 1965 Gordon Moore made the observation [2] that the number of components on an integrated circuit doubles every 18 months, leading in turn to a doubling in processor performance over the same time period. Despite this dramatic and ongoing increase in computational speed, there are applications such as games, computational fluid dynamics and realtime multimedia applications that tax even the fastest modern computers. This problem has been addressed in the past by expanding the instruction set and also adding dedicated hardware.

1.1.1 Instruction Set Expansion

Although central processing unit (CPU) instruction sets allow the programmer to perform almost any task, it can require many 10's of instructions to perform some relatively simple tasks. If an operation frequently occurs it may, under certain cir-

cumstances, be added into the instruction set to advantage. A good example of this is floating point maths. Many early instruction sets (e.g. x86, 68k, MIPS) didn't include this functionality, requiring it to be emulated using numerous integer operations. Floating point instructions were added when applications required the performance boost that they provided and when the available transistor count increased to the point where this became a practical solution.

Many modern CPU instruction sets have been extended to include single instruction multiple data (SIMD) instructions [3, 4]. This allows multiple arithmetic operations to be performed by a single instruction, by packing multiple values into a single register. An example of this would be a 32bit CPU that has a SIMD instruction that performs 4x8bit additions. This is implemented in hardware by disabling the carry propagation between the 8 bit groups which results in a significant performance improvement with very little additional hardware.

Although extending the instruction set can speedup a wide range of applications, the performance gain will always be limited by the instruction pipeline, register file and other fundamental components of the Von Neumann architecture.

1.1.2 Dedicated Hardware

As the execution units of a processor occupy only a small proportion of the total die area, CPUs are very inefficient in terms of both hardware utilization and power consumption. A popular way to address this and overcome the limitations of the Von Neumann architecture is to create dedicated hardware for specific tasks. As this solution doesn't require many of the hardware blocks found in a processor, (e.g. instruction fetch/decode, register files, etc) a larger proportion of the hardware is used to perform useful calculations.

A common example of this approach can be found in the current generation of

personal computers (PCs). Since the computational power required by modern 3D computer games is far greater than that available from even the fastest processors, dedicated hardware on the graphics card [5, 6] is used to render the 3D scenes. This releases the CPU to perform all the other tasks that are required in order to create the game environment. This "offloading" of computationally intensive tasks to hardware is even more frequently employed in embedded systems (e.g. set top boxes [7], mobile phones [8], etc) where CPU resources are often limited. Other common examples of hardware acceleration are cryptography, MPEG compression/decompression, and image improvement algorithms.

Using dedicated hardware is extremely powerful, however, it is only a practical solution where a task needs to be performed very frequently as such hardware is only capable of performing the specific task that it was designed to do. If that task doesn't need to be performed then the dedicated hardware will be idle. Rising mask costs and other non-recurring expenditures (NREs) are forcing manufacturers, especially in the embedded markets, to create devices for a much larger market segment. Consequently it is not always economically viable to design and implement hardware that is specific to only a small section of the market. A popular technique to reduce the amount of application specific hardware in these types of devices is to use one or more digital signal processing (DSP) cores [7, 8]. However there is a limit to how much processing power can be provided by this approach.

1.2 Reconfigurable Computing

By adding reconfigurable logic to a system it is possible to obtain a substantial performance improvement [9]. Although the performance achieved is very dependent on the application, speedups of 50 times or greater are not uncommon, whilst maintaining most of the flexibility associated with software [10, 11, 12, 13, 14].

In general algorithms with the characteristics listed below will benefit the most from the use of reconfigurable computing. Algorithms that possess all of these characteristics (e.g. encryption) achieve speed up factors of 1000 or more [15, 16].

High levels of instruction level parallelism (ILP) The greater the amount of ILP the more data can be processed in parallel.

Inherently parallel algorithms Some algorithms are inherently parallel in nature, as such different parts of the computation can be done in parallel (this parallelism is orthogonal to ILP). One example of this kind of parallelism is a simple brightness reduction algorithm, the calculation of the values of the pixels are independent from one another.

Simple operations Some classes of operation (e.g. bitwise AND, OR, etc) require minimal hardware resources to implement. The lower the number of resource required my an algorithm, the more instances of that algorithm can be implemented in the reconfigurable hardware.

Low memory bandwidth requirements Since the core algorithm will be instantiated many times in the FPGA, the total bandwidth required can become considerable. Since the amount of "off chip" bandwidth is restricted by physical factors (i.e. number of pins and maximum frequency) the amount of memory bandwidth available can severely limit the overall performance of a reconfigurable computing system.

1.2.1 Hardware Implementations

1.2.1.1 Direct Processor Integration

In some systems reconfigurable logic is an integral part of the processor [12, 13]. In these cases there are usually two communication channels. The first channel allows

instructions to be issued to the reconfigurable computing (RC) area, enabling the software to control the execution and setup initial parameters. The second channel gives access to memory allowing the RC area to perform direct memory access (DMA) operations, independently of the rest of the CPU core. This approach has several advantages:-

Low Latency As the RC area is directly connected to the instruction pipeline, only a few clock cycles are required to trigger execution. This is a major performance advantage for applications where the RC area is triggered repeatedly.

Cache Coherency As the DMA channel from the RC area is usually connected to the data cache, rather than directly to the memory interface, the system is naturally cache coherent.

Virtual Memory On systems that implement virtual memory, the DMA connection from the RC area can be made in the processor before the memory management unit (MMU), resulting in both the application hardware and software existing in the same virtual address space. This removes the overhead associated with translating pointers from the virtual to the physical address space and also reduces the complexity of the software to hardware conversion process.

High Bandwidth Because the computational throughput of reconfigurable computers is significantly higher than conventional processors, the memory bandwidth available will have an even greater impact on performance [17]. As a result the RC area must be closely integrated with the existing memory system to maximize bandwidth and to minimize the effect of this bottleneck.

In many systems where there is a direct connection to the instruction pipeline, the CPU will stall and remain idle while execution takes place in the RC area. However, this inefficiency is avoided on systems that implement simultaneous multi-threading

(SMT) [18] as the CPU issues instructions from multiple thread contexts at the same time. Consequently, a stall in one thread does not result in the entire CPU core becoming idle. Because the RC area is very tightly integrated with the CPU core, it must be fabricated on the same die, resulting in a significant increase to the overall die area of the device; this can also provide a significant improvement over dual core processors [19, 20, 21] which at most yield a 2x speedup for the cost of a doubling in the die area.

1.2.1.2 External RC Area

Although directly connecting the RC area to the CPU has many performance advantages, it is not always feasible to do this due to the high levels of integration required. The notable exception being CPUs that provide an interface to the instruction pipeline which allows the user to create their own execution units; this feature is not present on main stream processors, and is only usually found on CPU cores that are specifically developed for FPGAs [22, 23].

In systems where it is not possible to directly connect the RC area to the CPU pipeline, the RC area can, instead, be connected to one of the peripheral buses, usually the peripheral component interconnect (PCI) bus [24, 25, 26]. As discussed in section 1.2.1.1, the bandwidth available to the RC area has a significant impact on performance, however the standard PCI bus (32bit,33MHz) only provides 133 MBytes/s, which rises to 533 MBytes/s with the PCI-X bus (64bit,66MHz). Both are significantly lower than the 6.4 GBytes/s provided to the processor by the dual channel DDR-400 memory interface, present on modern computers. To try to address these bandwidth issues, research has been conducted into connecting field programmable gate arrays (FPGAs) directly to a computer's memory interface. This can be accomplished by mounting an FPGA onto a printed circuit board (PCB) with the same form factor as a memory module. The FPGA can then be

simply plugged into any computer system that supports the appropriate memory standard [27, 28, 29]. Although this improves both bandwidth and latency, additional problems are created:-

DMA Conventional memory buses are designed to be single master, which limits the FPGA to only accessing memory to which it is directly connected.

Obsolescence Unlike PCI, backwards compatibility is not a design goal for the memory bus, therefore any system that utilizes such an interface will need to be redesigned for each new memory standard.

Operating system (OS) integration Due to the low level and unorthodox nature of this interface, integration with the OS is required at the kernel level. This generally limits the use of the technique to those operating systems where the kernel source is available, for example Linux/BSD.

High performance computing (HPC) vendors Cray [30] and SGI [31] have introduced systems where FPGAs are directly connected to the main system bus, which results in high bandwidths. In addition, because these busses are multi master, there are no issues associated with DMA transactions. Although the latency is significantly better than in PCI based solutions, it is not as low as that in systems where the RC area is directly connected to the CPU. To reduce the performance impact caused, both the Cray and SGI systems have high bandwidth, low latency QDR-SRAM memories directly connected to the FPGAs. Both these external memories and the SRAM present inside the FPGA itself can be used to create small caches and buffers to improve performance.

1.2.2 Tools Flows

Creating the hardware infrastructure for a reconfigurable computing system is relatively straightforward and several solutions [26, 30, 31, 32] are commercially available. However a viable platform also requires a tool flow to create both the software application and the accompanying hardware configuration.

1.2.2.1 Separate SW And HW Tool Flows

Figure 1.1 shows the simplest tool flow. This has a high level language compiler (e.g. C/C++) for the software and a completely separate hardware description language (HDL) (e.g. VHDL or Verilog) in order to generate the hardware configuration image [33]. Although this approach gives the application developer great flexibility, it also exposes him/her to a split programming paradigm. This can introduce design flaws and bugs as most developers do not possess both hardware and software skill sets.

As current HDLs are very low level languages, manually porting existing software to hardware can take a considerable amount of time. One example of this is a seismic data processing algorithm, the kernel of which was ported to hardware [34]. Although this corresponds to a mere 80 lines of C code, the process took six man

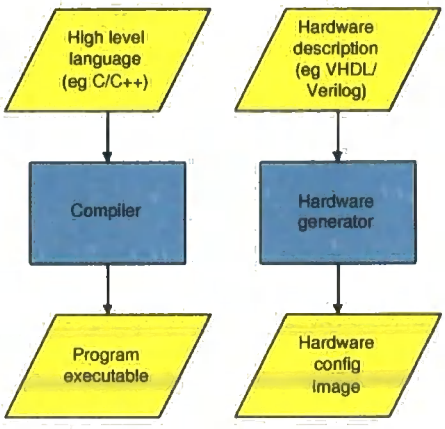


Figure 1.1: Separate SW and HW tool chains

months to complete. Although the result provided a considerable increase in performance, the effort levels and skills required represent a significant barrier to the widespread adoption of this technology.

It is possible to build library components of common functions (e.g. Fast Fourier transforms (FFTs), random number generators, sorting algorithms, etc) to reduce the amount of effort required to port applications to reconfigurable computing platforms. These library components consist of the hardware and a software wrapper to control it. This not only promotes intellectual property (IP) reuse but also provides a familiar interface to software developers requiring the performance benefits of hardware. An additional advantage of this approach is that the library function can also be implemented in software, thus allowing the system to decide, at runtime, whether functions are to be performed in software or in hardware [35].

1.2.2.2 Common High Level Language

To overcome the problems outlined in the section 1.2.2.1 the tool flow shown in figure 1.2 is becoming increasingly common. This flow uses a high level language to describe both the hardware and software [36, 37, 38, 39, 40, 41]; often a variant of C (as a large proportion of software is written in this language, porting software kernels to hardware is made considerably easier).

As with many other high level languages C [42], was originally created as a pure software language. Consequently it lacks some of the constructs required in a high level HDL. Most notably, there is no way of explicitly specifying parallelism. In addition, for reconfigurable computing platforms, it is not possible to specify whether a section of code should be implemented in software or in hardware. To overcome these limitations, most hardware C compilers extend the language with either additional keywords or pragmas. As both these directives have a significant impact on the effi-

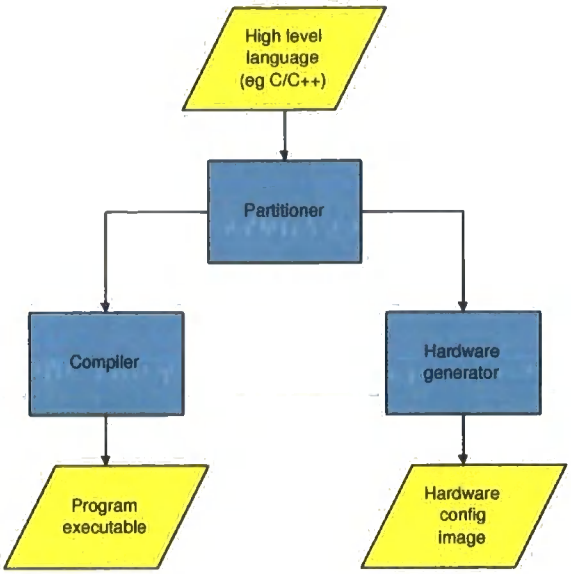


Figure 1.2: Common high level language

ciency of the resulting hardware and also have the potential to introduce bugs into the system, it is important that the developer has, at the very least, a rudimentary knowledge of both hardware in general and the target platform in particular.

The resulting executable from these types of tool flow contain the RC area configuration data which is very closely related to the underlying hardware architecture. Because of these low levels of abstraction, applications need to be compiled for the specific RC area concerned. Therefore, in order to create a viable platform one of the following options must be chosen:-

Fixed RC area By fixing the architecture, size, and speed of the RC area it would be possible to distribute applications to customers with a variety of different reconfigurable computing systems without the need to recompile and re-target. However this approach is not without its drawbacks. Given that different market segments have vastly different needs (e.g. mobile, desktop, server) it would be impractical to fix the RC area. In addition to this, since the speed of processors is increasing, as stated in section 1.1 if the parameters of the RC area were to be fixed the performance of pure software would eventually

overtake the performance of hardware.

Compile every application for every RC area If there is no requirement for backwards compatibility, the RC area may be freely changed with each version of the processor, allowing the performance of the RC area to be appropriately scaled as technology advances. However with no backwards compatibility, every application would need to be compiled for every system. Clearly this is not feasible in the mainstream computer market. However this does provide a possible solution for embedded markets were applications are typically written for specific platforms.

To address some of the above problems, it has been proposed that the hardware configuration image could be distributed in a high level, abstracted form [43]. As a result the target platform would perform both the mapping and place and route (PAR) stages and new device architectures and tools have been designed that significantly reduce the time taken to perform these steps [44], albeit at the expense of device size and operating frequency. However there are still several problems associated with this approach:-

Application vendors need to target hardware Because the program contains an abstracted hardware binary, application vendors will require additional time and hardware expertise to gain the performance advantages associated with reconfigurable computing.

Performance does not automatically scale with RC size Although the size of the RC area can be increased without the need to recompile, only those applications that have been written specifically to take advantage of the increased hardware resources will benefit.

Compiler modifications The compiler tool chain would need to be extended in order to support both the additional partitioning and synthesis steps required.

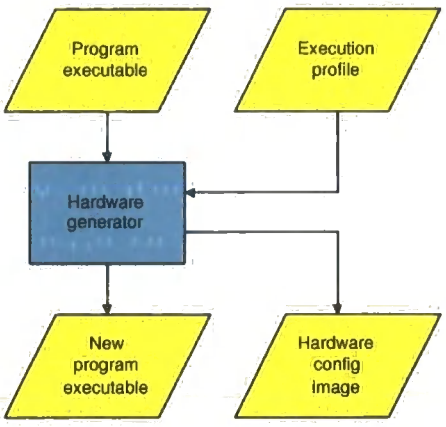


Figure 1.3: Low level tool chain

This would have to be done for every supported language (e.g. C/C++, Fortran, Java).

1.2.2.3 Low Level Tool Flows

Instead of using a high level language as the starting point for the reconfigurable tool chain as described in section 1.2.2.2, it is possible to start from an existing software binary [45]. This approach can be extended to form the tool flow shown in figure 1.3 by using runtime execution profiling [46, 47] to identify computationally intensive sections of code. The key difference between this and other tool flows is that it is run by end users on their systems and not by application developers, as is usually the case. This is similar to just in time (JIT) compiler technology, now commonplace in the software domain, where it is used to execute non native code at close to native performance [48, 49, 50]. However, instead of translating one instruction set to another, this tool flow translates computationally intensive sections of code from the native instruction set to hardware whilst at the same time generating a modified version of the application in order to utilize the newly generated hardware.

With this tool flow, the reconfigurable hardware is completely abstracted from the application software. Indeed, the software developer, is not even aware of the pres-

ence of the reconfigurable hardware. Two benefits arise from this approach: application developers do not need specific hardware knowledge and existing applications will benefit from hardware acceleration without having to be recompiled. This complete abstraction also gives hardware vendors the flexibility to change the size, speed, and architecture of the RC area without adversely affecting backwards compatibility. As a result the performance, cost, and power consumption of the system can be tailored to suit different market segments (e.g. mobile, desktop, and server) whilst still providing the flexibility to allow the system to keep up with the ever increasing pace of technological change.

1.2.3 Scheduling

Amdahl's law [51] states:-

> "If F is the fraction of a calculation that is sequential, and (1-F) is the fraction that can be parallelised, then the maximum speed-up that can be achieved by using P processors is $\frac{1}{F+\frac{1-F}{P}}$"

Although originally related to multi processor systems, Amdahl's law can also be applied to reconfigurable computing since, in order to achieve a large overall increase in performance, a large proportion of the computationally intensive code must be converted to hardware. Because computer systems often perform multiple tasks with each task containing several computationally intensive loops, there are, potentially, a large number of code sections that may benefit from hardware conversion. However merely increasing the size of the RC area to accommodate simultaneously all the intensive sections of code is very wasteful of hardware resource, since only a small proportion of the RC area will be active at any one time. To address this problem it is possible, utilizing the dynamic nature of a reconfigurable computing environment to move hardware in and out of the RC area as required [52, 53].

To minimize the idle time associated with reconfiguring the RC area, it has been proposed that multi context devices [54] be used, allowing the RC area to be active with one context whilst another context is being reprogrammed. These contexts can then be quickly swapped thus activating the newly programmed context. This is similar to the double buffering technique often used in computer graphics. Multi-context aware scheduling algorithms can be used to minimize both the memory bandwidth and the idle time associated with moving configuration data around the system [55].

1.2.4 Floating Point Maths

The hardware resource required to implement some mathematical functions can be considerable. As a result most modern FPGAs contain dedicated hardware to perform such operations as multiplication, since this and similar functions are relatively common and consume a considerable amount of hardware resource. Until recently, reconfigurable computing has been restricted to integer and fixed point arithmetic due to the large hardware resources required by floating point units. However, the combination of floating point cores that are specifically designed and optimized for FPGAs [56, 57, 58], together with the latest generation of larger devices means that floating point arithmetic is now achievable on FPGAs.

Due to their size, it is important to match floating point cores to the specific application that is to be implemented in hardware. For example, in most cases, it is advantageous to increase the latency of a core to match the surrounding hardware. Not only does this eliminate the need to pipeline the result through several stages of registers to match the rest of the data flow, but it also provides the opportunity to further optimize the core and reduce its hardware utilization. To this end, research has been conducted to produce floating point core generators [59] rather than fixed cores. These core generators allow the RC tool flow to generate different cores for

different sections of the algorithm depending on both the latency and throughput requirements.

To reduce further the hardware requirements, it is possible to change the number representations that are used depending on both the number and type of operations present in the hardware. For example, by using a higher radix floating point notation, it is possible to reduce the hardware requirements of floating point cores by 12-25% [60]. However, the decision to use a higher radix notation must be made on a case by case basis. This is due to the fact that the amount of hardware required to convert to and from the IEEE format used by the host CPU can, in some cases, be greater than the hardware saved by using the high radix notation. Using logarithm (LOG) notation has also been investigated, as this can also produce substantial reductions in hardware usage provided the algorithm contains mainly multiply and/or divide operations [61]. The decision to employ LOG notation has to be informed by the overhead associated with converting to and from the IEEE floating point format and also by the number of addition/subtraction operations involved, as these require significant amounts of hardware resource in the LOG domain.

1.3 Research Conducted

The research presented in this thesis concentrates on the development and investigation of low level tool flows (see section 1.2.2.3) with the ultimate aim of creating a reconfigurable computing system that would be suitable for mass market adoption. In order to achieve this goal it is important that the final system exhibit the following characteristics:-

Abstraction Complete abstraction between the hardware and software is required to provide backward compatibility. Without this basic requirement it is impossible to create a platform that will achieve mass market adoption.

Automatic conversion The conversion from software to hardware, and the subsequent integration of the newly created hardware must be completely automatic, and require no user intervention. This requirement is particularly important if the conversion is performed on the end user computer, rather than the original software developers (see section 1.2.2.3).

Low conversion time To obtain wide spread user acceptance the software to hardware conversion process must be performed relatively quickly. Again this is especially important on systems where the conversion is performed by the end used. Although the output of the conversion process can be cached and reused during all subsequent executions of a program, this technique can't be used during the software development process, where the program is changing frequently. In addition some users would find the relatively slow, initial execution un-executable.

Large performance increase The use of reconfigurable technology in mainstream environments would represent a significant departure from the conventional techniques used to increase performance (e.g. larger caches, wider execution pipelines, faster clock speeds, etc). If such a radical technology is be be adopted then it must offer significant performance advantages over existing approaches over a wide range of different types of software.

Much of the previous research conducted in this field has concentrated on producing systems that produce significant increases in performance [10, 11, 12, 13, 14, 15]. Although this is an important aspect to any reconfigurable computing platform, it must not be overshadow other aspects like abstraction, as without these extra features it is not possible to create a viable reconfigurable computing platform. The research presented in this these addresses all of the required features, with the exception of the time required to perform the software to hardware conversion, as this is an issue that is closely linked to the architecture of the reconfigurable

area, and not the conversion tool chain. This has however been addressed on other research [44]. In addition to the work presented in this thesis, extra development and investigation is required before a viable platform is available, details of the required work can be see in chapter 7.

1.4 Warp Processing

The concept of generating hardware at runtime from an existing software binary was termed "warp processing" by F Vahid and his team at the Department of Computer Science and Engineering, University of California, Riverside [62]. This research group performed some preliminary research into the development and use of low level tool flows that is similar to that described in section 1.2.2.3.

1.4.1 Hardware Platforms

During the course of their research, evaluations of several reconfigurable computing systems based on different host processors were performed. The general hardware architecture that they used is shown in figure 1.4. The first thing to note about this particular architecture is the use of a completely separate processor to perform the software to hardware conversion which, in many of the test cases that they presented, was a duplicate of the main host CPU [62, 63, 64]. Although not made explicit in their research, this would result in a device that is considerably larger than twice the size of the original processor core, after the additional hardware required for the RC area is taken into account. As a result, the hardware acceleration must produce substantially more than a 2x increase in performance for their platform to produce any advantage. In the majority of computer systems the CPU will spend a significant proportion of its time idle. Since this idle time, could in most cases,

be used to perform the hardware conversion without impacting performance, it is questionable, whether this second, dedicated processor is required.

1.4.2 Tool Flow

The hardware generation tool flow utilized by the "warp processor" developed by F Vahid and his team, differs slightly from the tool flow described in section 1.2.2.3 as their first step was to decompile the program that was to be converted to the C language [63, 65]. They then used the additional information available in this high level language to further optimize the hardware generation process [65, 66]. However deriving this additional information necessitated making certain assumptions based on a knowledge of the original compilation process. Such assumptions can lead to errors in the decompiled source code and this is especially true in cases where the software was originally written in a different language (e.g. Fortran). This is due to the fact that different languages can handle high level constructs (e.g. arrays) very differently. Any errors in the decompiled source code can result in bugs, data corruption, and/or system instability.

It is not always possible to decompile a program executable. The use of intricate branching, differing high level languages and high levels of optimization all increase

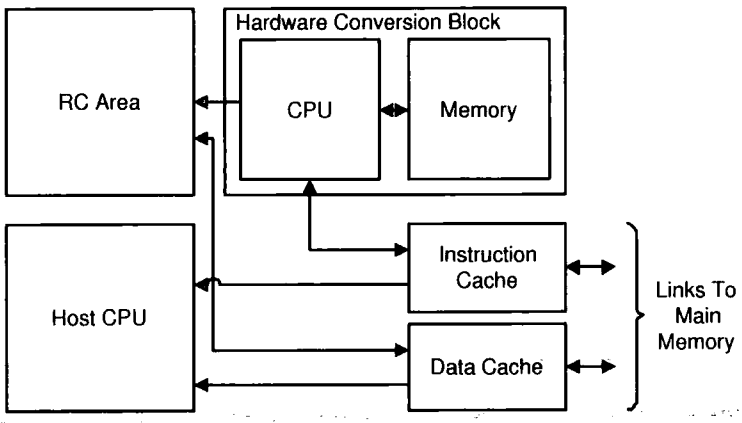


Figure 1.4: Warp processor architecture

Host CPU	Reference	Clock speeds		Speedup produced	
		CPU	RC area	Average	Peak
ARM 7	[62]	100 MHz	250 MHz	7.4x	16x
MicroBlaze	[64]	85 MHz	250 MHz	5.8x	16.9x
ARM 7	[68]	75 MHz	60 MHz	2.1x	4.2x
MIPS	[69]	100 MHz	100 MHz	1.4x	1.9x

Table 1.1: Warp processing test platforms

the complexity of the program executable, reducing the likelihood that the decompilation of a specific section of code will be successful. In a "warp processing" system, this can lead to highly computationally intensive sections of code not being converted to hardware. This is probably the reason why the majority of the compiler optimizations were turned off for some of the test cases evaluated [65]. This disabling of the compiler optimizations reduces the performance of the software and can artificially increase the speedup achieved by the conversion to hardware.

1.4.3 Performance Improvements

The "warp processing" concept was evaluated by the California team on a number of different host processors, including ARM, MIPS, and the MicroBlaze. A summary of their test platforms is shown in table 1.1. The performance increase produced by the first two test cases averaged 7.4x and 5.8x respectively. These performance increases are partially due to the fact that the clock speed of the RC area is considerably higher than that of the CPU. However, as a result of the overheads associated with the reconfigurable nature of the RC area, it is not uncommon for the RC area to be an order of magnitude slower than the CPU. One example of this is the Cray XD1 [67] where the maximum clock speed of the FPGA is 11x lower than the CPU. In the "Warp" test cases where the clock speed of the RC area is either the same as or slightly lower than the CPU, the average performance improvement drops to a mere 2.1x and 1.4x respectively.

Due to the complex nature of the hardware platforms, the performance improvements were calculated by combining the results of VHDL simulations of the RC area with behavioural simulations of the CPU cores. Due to the lack of actual hardware, several real world factors, such as the communications overhead between the CPU and the RC area, would not have been taken into account. These factors will further reduce the performance improvements that were reported.

1.5 Summary

The performance of computer systems roughly doubles every 18 months. However, since the vast amount of computing power that this provides is still not enough for many applications there has been a shift to the use of dedicated hardware. In the PC arena this is illustrated by the fact that some graphics cards now have higher transistor counts than their companion CPUs (GeForce 7800 GTX = 302 million transistors, Dual core AMD Opteron = 233 million transistors). Although this translates into extremely high levels of performance, it is limited to 3D graphics applications. A new computing paradigm is required that provides the performance offered by these dedicated hardware solutions without loosing the flexibility and backwards compatibility associated with conventional software.

Reconfigurable computing has the potential to provide the required performance without sacrificing flexibility. However, memory bandwidth, device size and tool flow issues have blocked the widespread adoption of this technology. The latest generation of larger FPGAs combined with new techniques for implementing floating point arithmetic, have finally made reconfigurable computing an achievable goal. As such, several major HPC vendors have started to incorporate FPGAs into their product ranges. The tight coupling and integration provided by these systems has dramatically increased the memory bandwidth available when compared to tradi-

tional expansion card solutions. With the emergence of runtime JIT based tool flows, reconfigurable computing is finally becoming a possibility, however significant work is required to fully realize the potential benefits.

Chapter 2

Behavioural Simulation

2.1 Background

Modern systems on a chip (SoC's) are becoming increasingly complex, and often include a mixture of dedicated hardware blocks together with real time software, running on one or more embedded processors [70]. To help ensure that the next generation of these devices hit their market windows, avoiding expensive redesigns and/or over engineered solutions, it is essential that the initial design concept is correct, and that the system is capable of performing all the specified use-cases without exceeding such device performance limits as memory bandwidth, CPU utilization, and latency. Usually the first stage in the design process is to define the system use-cases and calculate the device performance required to implement each of these. Commonly, this is determined by adding up the required bandwidth for each component of a use-case and adding a safety margin. However, this approach can be very prone to error as the bandwidths required by the different components in the system are frequently estimates. In addition, this approach will only give the average bandwidth required by the use-case, and will not take into account any peaks that might occur in the required bandwidth. These peaks may cause buffers

to overflow and data to be lost, resulting in, for example, dropped video frames or other artefacts that are not acceptable to the customer.

It has been shown [71, 72] that the use of behavioural simulation tools (with a high level of abstraction) in the early stages of a project, produces useful data which can, in turn, be used to guide architectural decisions. In addition to this, behavioural models have many other uses. For example if a complete behavioural model of a system is produced, it may be used to provide an early hardware target for embedded software engineers [73], enabling software development to run in parallel with hardware development throughout the majority of the project. This not only decreases time to market, but also reduces the knock on effect of late and/or unstable hardware on the software development cycle. Even when a stable hardware platform is available, a cycle accurate model can still be extremely useful. For example, it may be used when writing optimised DSP software [74] in order to provide detailed information on which sections of code need to be optimised, and it will also indicate the types of performance bottlenecks that are present. To date, a lot of behavioural models have been written using proprietary and often inflexible frameworks. This can hinder the re-use of blocks within the model and cause problems when trying to integrate multiple models (e.g. DSP core, MPEG decoder) into a model of a SoC. Open standards such as SystemC [75] are starting to appear that provide a common framework that enhances re-use. Unfortunately these standards often have serious flaws in their architectures and do not address many of the other problems, such as visualisation, simulator overheads and ease of use, associated with behavioural modelling.

2.2 CPUSim

A behavioural simulation system called CPUSim was developed during the course of this research to address some of the problems traditionally associated with behavioural modelling e.g. ease of use, simulation speed and re-use. This system allows the quick and early evaluation of different device concepts and also provides data on a wide variety of performance metrics. The language chosen for CPUSim was Java [76], which enables the simulation to be run, without modification, on a wide range of platforms including Windows, Linux, Mac and Unix. In addition to being used extensively throughout the course of this research, CPUSim has also been used by Philips Semiconductors to model the effects of changing the arbitration settings and the sizes of the first in first out (FIFO) buffers present in the memory hub of the PNX8550 hybrid TV processor [7].

2.2.1 Concurrency

The inherent parallel nature of hardware requires that any model incorporates some mechanism for emulating this. Commonly, this is done at the block level where the behaviour of a block is defined using normal sequential code and then each block in the system is run pseudo-concurrently. In most cases this pseudo-concurrency is implemented by running each block in its own thread [75]. This approach, unfortunately, has several side effects:-

- As most threading systems are random in nature, blocks can be executed in any order. This can result in the system being non-deterministic.
- As there is no concept of time within the system, it is extremely problematic or even impossible to get accurate data on any time related metric such as bandwidth.

- The threading mechanism is exposed to the model writer. Writing multi-threaded programs can be very error prone and can lead to deadlocks, concurrent access, and priority inversions. These are often very difficult to debug owing to the random nature of these types of problems.

- Since each block in the system exists in its own thread, a complex simulation may easily contain over 50 threads. Context switching between so many threads is a very time consuming process, leading to large overheads and poor simulation speed.

To resolve some of these problems, it is common to implement some form of synchronisation within the system; usually in the form of a clock event that triggers the actions of the blocks in the simulator [77]. However, this does not solve the problems associated with debugging a multi-threaded application, or the poor performance associated with this type of approach.

To avoid the above problems, CPUSim uses a different method to simulate concurrent execution. A clock generation block calls the "`clockPulse()`" function on each block in the clock domain advancing the state of the block by one clock cycle. Once this is complete the process is repeated. However, there is a serious problem with this simple clocking scheme. For instance, take the example of a counter which is incremented every clock cycle and a display, which is also updated once every cycle. The simulator will give different results depending on the order in which the blocks are clocked (see figure 2.1). Clearly this is unacceptable. To resolve this problem, a second clock function called "`clockSwapState()`" may be used. This, when combined with the following rules, will ensure an accurate simulation:-

- A block is only allowed to communicate to another block during the execution of the "`clockPulse()`" function.

- The externally visible state of a block is only allowed to change during the

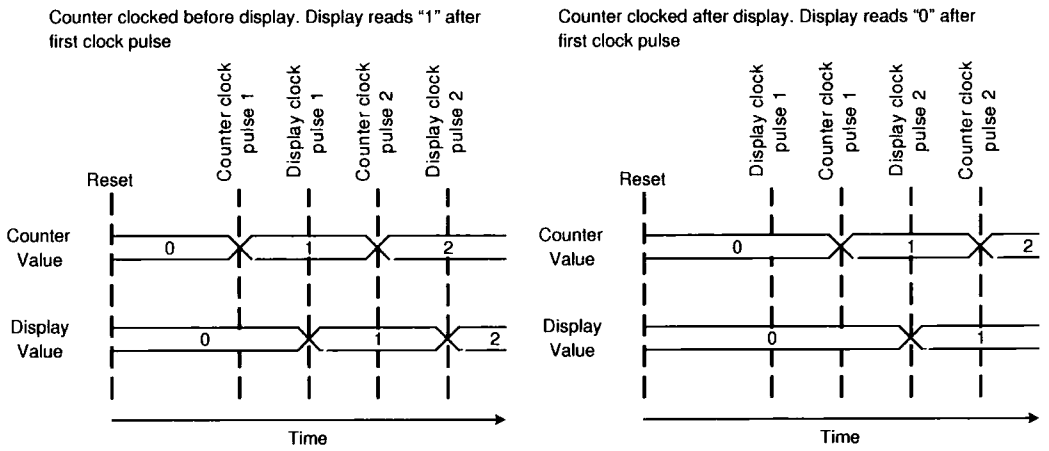


Figure 2.1: Single function clocking

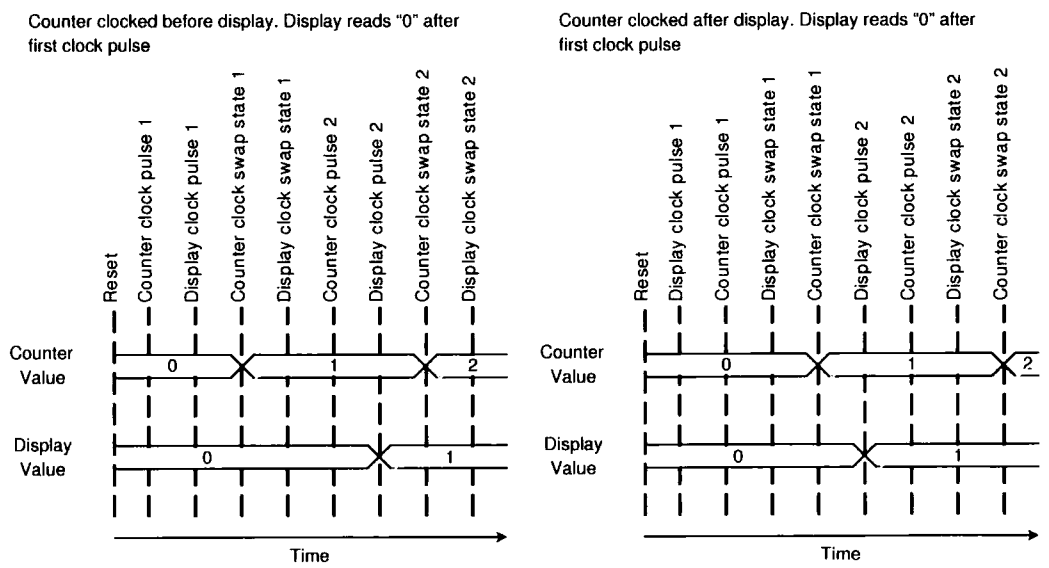


Figure 2.2: Dual function clocking

execution of the "clockSwapState()" function.

- The clock block must ensure that all the "clockPulse()" functions on the blocks have been completed before any of the "clockSwapState()" functions are called and visa versa (see figure 2.2).

This clocking scheme has the following advantages:-

- As the simulator is deterministic and cycle accurate, it can be used to provide accurate performance metrics.
- As the threading scheme is handled centrally, there is only a very small amount of multi-threading code, simplifying testing and debugging. Additionally as there is no synchronization code in the models of the blocks, the development time is also reduced.
- Since the number of threads used to clock the system can be based on the number of processors present in the host computer and not the number of blocks in the simulator (as is often the case), a lot of the simulation overheads are eliminated.

It is also worth noting that this two stage clocking strategy is similar to the use of the rising and falling edges of a conventional clock signal. This facilitates the integration of real hardware with the simulation environment as is described in section 3.4.4.

2.2.2 Flexibility

To reduce the cost of modelling a system, it is important that the blocks in the simulator are reusable. There are, therefore, two factors that should be considered before writing a block:-

Parameterization How generic is the model - how many different situations can it be used in?

Interface How many different types of blocks can the block be connected to?

The concept of having a parameterizable simulation has been used in the past to investigate different architectures [78]. However, parameterization can be taken a step further to create generic blocks which can be used in multiple situations. One example of this is the case of a simple memory block with parameterizable size, latency and number of ports. If a memory were to be instantiated with the following features: large size, high latency and one port, it could be used to simulate a synchronous dynamic random access memory (SDRAM) device. If the same memory were to be instantiated with a much smaller size, single clock cycle latency and multiple read and write ports, it could then be used to simulate the register file in a processor.

For this "write once, use anywhere" approach to work, there must be common interfaces. For example, a data cache must fetch data from the memory using the same mechanism that the instruction decoder uses to fetch data from the register file. This, in turn, means that the interface must be generic enough to allow its use in both of these situations, whilst at the same time allowing for a high degree of simulation detail. One example of this is the "Addressable" interface which specifies the following functions:-

- `boolean write( int address, int data )`
- `boolean issueRead( int address, Track trackObj )`
- `Integer checkRead( Track trackObj )`

This interface allows a connected block to issue a read or a write operation to the destination block, and in the case of the "read" obtain the result at a later point in

time by using the "checkRead()" function. This approach offers the implementer a great deal of flexibility. For example, a burst transaction could be implemented by returning the result of the first read operation after ten clock cycles together with the result of a subsequent read to an adjacent address with a one cycle latency. This type of interface also allows transactions to be rejected and, by changing the way in which transactions are rejected, the destination block can effectively alter the number of simultaneous pipelined transactions that are allowed.

Because this interface is highly abstracted, it can be implemented by a wide variety of blocks (e.g. register files, frame buffers, caches, and hardware input/output (IO) links) and as a result all of these blocks become interchangeable. This gives the user great flexibility when choosing precisely which blocks to add to a simulation and how best to connect them.

2.2.3 Instantiation and Connection

The time taken between the conception of a new system architecture and obtaining a set of performance metrics from it is one of the limiting factors which determines how many different architectures can be evaluated before market pressures force the start of the implementation phase. The two major contributing factors to this round trip time are: the time taken to implement new architectures in the simulator and the speed of the simulation. The majority of the simulation environments that are available produce static models [75]. This obliges the user to go through a laborious process of changing the source code and then recompiling the model, before the simulation can be run and the effects of the changes evaluated. In addition to being a time consuming task, this also requires software development skills that many hardware engineers and system architects lack. Because CPUSim uses common interfaces for both instantiating and connecting the blocks, it is possible to make changes to:-

- The number and type of blocks
- Their parameters
- How they are connected

at runtime from the graphical user interface (GUI) without the need to recompile the program. This fact allows the user to make changes quickly and easily to the system and to observe instantly the effects of any modifications.

At the heart of this flexibility is the run-time connection and checking interface. This is comprised of the following two functions:-

- `connectToDestination( HardwareBlock destinationBlock , int portNum )`
- `connectionsComplete()`

By calling the "`connectToDestination()`" function on a block, the simulator can create a connection between that block and the "`destinationBlock`". As some blocks have to be connected to several blocks of different types (e.g. an instruction decoder needs to be connected to the instruction fetch unit and also to the register file) a "portNum" is used to specify from which port on the block the connection is being made. The top level simulation infrastructure also enumerates these port numbers in order to present meaningful names to the user. If an error occurs (e.g. the "`destinationBlock`" is not the correct type of block), an exception is thrown. When all the connections have been made, the simulator calls the "`connectionsComplete()`" function allowing the block to throw an exception if any of the required connections are missing. Any exceptions that are thrown by these two functions are caught by the simulator and turned into a meaningful error message which is displayed to the user.

2.2.4 Visualisation and Statistics

Good visualisation of a simulated system is vital, as this can greatly speedup both the development and the debugging of the blocks in the model. Good visualisation also aids the user in understanding the behaviour of the block within the context of the wider system.

Often the visualisation of models is, unfortunately, an afterthought and is not an integrated part of many behavioural modelling systems. In some cases, the visualisation framework is not even written in the same language as the model implementation [79]. This can slow down the development of a visualisation and it can also discourage, altogether, the model writer from producing a visualisation. Not taking the time to produce an effective means of visualisation is a false economy that is far outweighed by the additional time taken to debug and understand the behaviour of a block within the context of the wider system.

To address the above, CPUSim has an integrated visualisation framework that reduces the time taken to produce the visualisation for a block, whilst also providing several visualisation modules that can be reused. For example, the visualisation of the frame buffer block provides a graphical view of the buffer together with an editable, table based view of the raw data within the buffer. This table based view reuses the same visualisation component that is used by the memory block. It is worth noting that because the simulated system is built up graphically from individual blocks using a hierarchical block diagram, the user also has an automatically generated structural tree diagram of the system.

Although visualisation is a good way to examine the state of a system at any given point in time, it is not well suited to gather information on the behaviour of the system over a period of more than a few ten's of clock cycles. In order to overcome this limitation a statistics infrastructure is incorporated into CPUSim, allowing

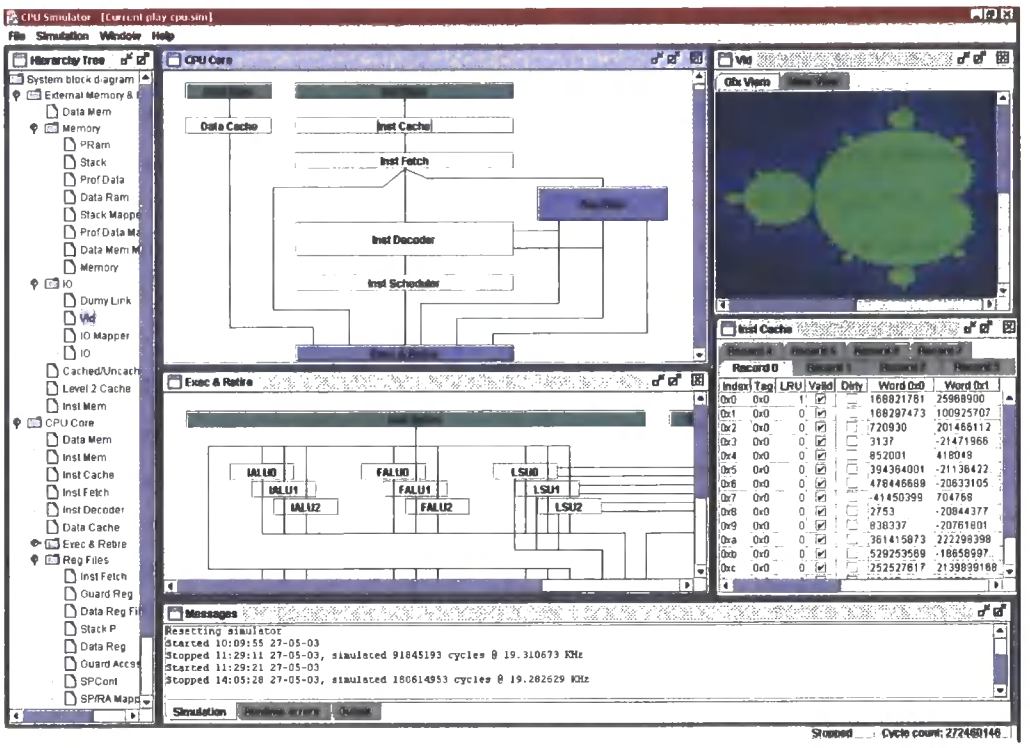


Figure 2.3: CPUSim simulating an experimental processor, showing the contents of the instruction cache and a Mandelbrot fractal generated by software running on the simulated processor.

any block in the system to generate statistics on any aspect of its operation. The underlying infrastructure can then graphically display these statistics to the user on a block by block basis and/or produce a report containing some generic details of the simulation run, together with the statistics data from every block in the system. Some of the visual elements that make up the CPUSim system are shown is figure 2.3.

2.3 Summary

Behavioural modelling is a vital tool for the exploration of device and system architectures. However, existing simulation systems lack key features such as an integral visualization framework and a statistics gathering mechanism. This not only increase model development time, but also hinder the user's understanding and usage

of the system. In addition, the reliance on threading techniques in these simulators to provide the inherent parallelism, present in hardware, is not only a source of bugs and longer development times, but also significantly decreases the speed of the resulting simulation.

CPUSim was developed to address the shortcomings of existing behavioural simulation systems. Its centralized clocking mechanism significantly improves both model development and execution times. This together with its integral visualisation/statistics frameworks and GUI based parameterization and connection system, make CPUSim a ideal tool for quickly creating and evaluating new system concepts and architectures.

Chapter 3

EPIC Simulation

3.1 CPU Architecture

To evaluate potential reconfigurable computing tool flows and architectures, a model of a CPU and its associated peripherals was created for the behavioural simulator CPUSim which was described in section 2.2. Although based on an experimental architecture, the CPU core is fundamentally an explicitly parallel instruction computing (EPIC) processor [80], sharing many design concepts with the Intel Itanium [81, 82] range of processors. However, the following differences are worth noting:-

- The instruction bundle size is variable, instead of being fixed at three instructions as is the case with the Itanium.
- There is no register stack engine [83], so as with most other CPU architectures stacking has to be handled explicitly.

The specification of the simulated CPU is shown in table 3.1.

Effective core/bus clock ratio	10
Max instruction fetches per cycle	6
Max instruction issues per cycle	5
General purpose register file	256 x 32 bit registers
Guard register file size	32 registers
Num exec units per type	3
L1 I cache	16 KBytes, 4 way set associative
L1 D cache	16 KBytes, 4 way set associative
L2 cache	256 KBytes, 8 way set associative
L3 cache	9 MBytes, 18 way set associative

Table 3.1: EPIC CPU specification

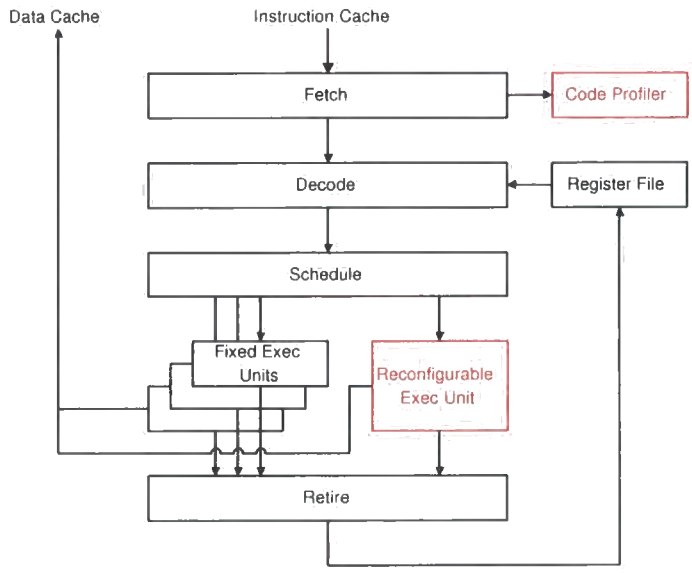


Figure 3.1: EPIC CPU core with additional RC blocks shown in red

3.1.1 Additional RC Hardware Blocks

For the purpose of evaluating potential reconfigurable computing architectures and tools flows, the direct connection topology described in section 1.2.1.1 was used. This topology has clear performance advantages and can easily be implemented in the simulation environment. To support reconfigurable computing, two additional blocks are added to the CPU instruction pipeline as shown in figure 3.1.

3.1.1.1 Code Profiler

The code profiler block is connected to the instruction fetch unit and monitors any jumps or branches that occur. This block consists of some control logic together with a cache that holds both the target address of the branch and the number of times that the branch has occurred. As new branches are encountered, additional cache entries are created and old entries are flushed out to memory, as necessary, based on a least recently used (LRU) policy [84]. This approach has two main advantages:-

- Since it is performed in hardware there is no software overhead associated with profiling the execution of a program.
- Since a cache is incorporated into the block, the memory bandwidth consumed by the profile data is minimized.

It has been demonstrated that performance profiling of this nature can be added to a MIPS 4Kp processor with a 2.4% increase in power consumption and a 10.5% increase in die area [47]. As MIPS 4Kp is a relatively small CPU core, by modern standards, the percentage die area utilized and the power consumed are both significantly higher than would be the case if a larger/modern CPU core were to be used.

3.1.1.2 Reconfigurable Execution Unit

This block presents the same interface as other execution units in the processor, but in addition it has a connection to the data cache which enables DMA operations from within the reconfigurable hardware. The exact nature of this block is defined by the output of the RC conversion tools, however, there are two distinct operating modes:-

- Behavioural simulation, using the simulation infrastructure provided by CPUSim.
- A wrapper interface to a real hardware implementation of the RC hardware as described in section 3.4.4.

3.2 RC Conversion Algorithms

The experimental RC tool flow used (see figure 3.2), is essentially the same as the flow described in section 1.2.2.3, differing only in that the hardware generation step operates on assembly code and not on the raw program binary and as such there are additional disassembly/assembly stages in the flow.

To facilitate the evaluation of different hardware conversion algorithms, the hardware generation tool contains a plug-in interface that abstracts the conversion process from the majority of the tool flow. Instruction combination and loop extraction hardware conversion plug-ins have also been developed (See sections 3.2.1 and 3.2.2). The performance improvements produced by these conversion algorithms is compared in section 3.4. As the loop conversion algorithm was found to be effective an enhanced version was developed, this is described in chapter 4.

3.2.1 Instruction Combination

As discussed in section 1.1.1, it is common practice to add instructions to processors, instructions that are specific to certain classes of applications [85, 86]. Although these operations can usually be performed using generic instructions, the number of instructions required is often greater, thus leading to lower performance. Additionally, it is not possible to predict exactly what specialist instructions will be required for every application. Furthermore, power consumption and die area limitations make it impractical to implement such a large number of additional instructions.

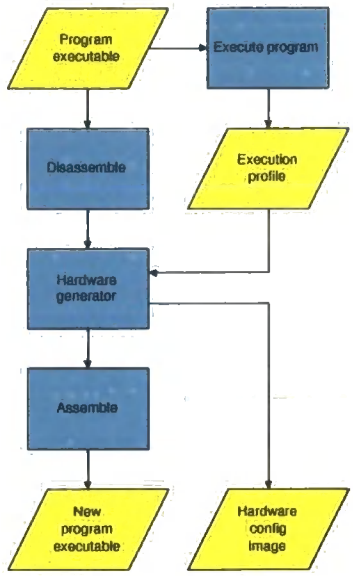


Figure 3.2: EPIC CPU RC tool flow

Processor manufacturers are, therefore, forced to implement only such commonly used instructions as the multiply and accumulate (MAC) [87].

By analyzing the instruction interrelationships in computationally intensive sections of code and by looking for patterns of instructions that occur repeatedly, it is possible to dynamically extend the instruction set to include instructions that would be beneficial to a program that is currently being executed. For example, code listing 3.1 contains two occurrences of the "iadd(iadd(a,b),imul(c,d))" instruction pattern. Once these instruction patterns have been identified, they are converted into hardware blocks with the number of instances of each hardware block being based on the number of occurrences of the instruction pattern in the software, together with the available hardware recourses in the RC area. The final stage of the optimization process is to remove all occurrences of the instruction pattern from the software and replace them with instruction(s) that refer to the newly created hardware, as shown in listing 3.2. It is worth noting, that because the instruction pattern in this example uses four different parameters, it requires two instructions to feed the data into the new execution unit.

```
and  r10, r10 0xff
imul r58, r2  r113 # Inst group 1
and  r34, r34 0xff
and  r11, r11 0xff
imul r59, r3  r113 # Inst group 2
and  r35, r35 0xff
imul r42, r34 r111
imul r50, r10 r112
imul r43, r35 r111
imul r51, r11 r112
iadd r97, r42 r50  # Inst group 1
iadd r98, r43 r51  # Inst group 2
iadd r97, r97 r58  # Inst group 1
iadd r98, r98 r59  # Inst group 2
ssr  r97, r97 0xf
ssr  r98, r98 0xf
```

Listing 3.1: Sample code before instruction combination

```
and  r10, r10 0xff
and  r34, r34 0xff
and  r11, r11 0xff
and  r35, r35 0xff
imul r42, r34 r111
imul r50, r10 r112
imul r43, r35 r111
imul r51, r11 r112
hw0  r97, r42 r50  # RC insts for group 1
hw1  r97, r2  r113 # RC insts for group 1
hw0  r98, r43 r51  # RC insts for group 2
hw1  r98, r3  r113 # RC insts for group 2
ssr  r97, r97 0xf
ssr  r98, r98 0xf
```

Listing 3.2: Sample code after instruction combination code

3.2.2 Loop Conversion

The loop conversion algorithm uses the execution profile to locate the most computationally intensive loop that is viable and then translates this in its entirety to a hardware data flow pipeline. The following three factors are considered to determine whether a loop is viable and if it can be converted into hardware:-

Hardware implementable instructions All the instructions in the candidate loop must be able to be converted into hardware. Whilst this is true of the vast majority of instructions, some classes of instructions cannot easily be implemented in a data flow pipeline (e.g. Traps, breaks, and cache operations).

None branching code The loop body must not contain any branching code as a hardware pipeline cannot implement non-linear execution flow. Since the host EPIC CPU implements guarding [88, 89], the majority of non-linear code flow is resolved to data flow by the compiler. In most cases this results in the remaining branching being due to function calls and/or nested loops.

Hardware resource usage During the early stages of the conversion process, the tools estimate the hardware resource usage and the candidate loop is rejected if this estimate is greater than the available hardware resources.

The actual mapping from software to hardware is performed by instantiating a hardware primitive for each instruction in the loop, and these are then placed onto a hardware pipeline similar to the one shown in figure 3.3. This results in parallelism in two orthogonal directions:-

- Conventional instruction level parallelism (ILP) is exploited by packing multiple operations onto the same pipeline stage.
- Loop level parallelism is exploited by having multiple data sets iterating through

the pipeline at the same time. This is equivalent to processing multiple iterations of the loop simultaneously.

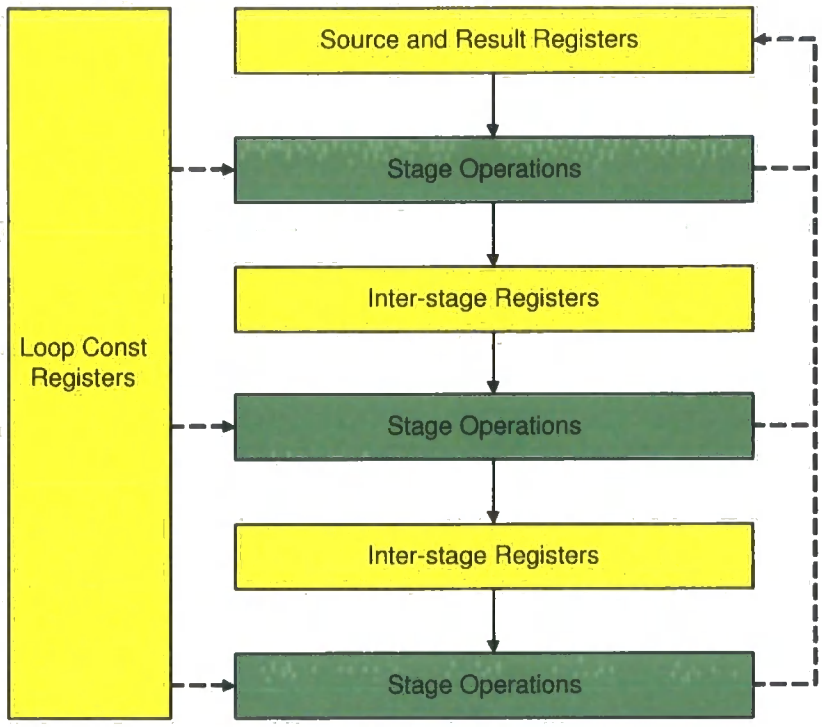


Figure 3.3: Example RC data flow pipeline

3.2.2.1 Data Flow Pipeline

As shown in figure 3.3 there are three distinct classes of registers in the pipeline: source/result, inter-stage and loop constant registers:-

Source/result registers The source/result registers are on the first stage of the pipeline. They hold all initial values and can only be read by the operations on this first stage, however they may be written to from any stage of the pipeline. Once all the required registers of this type have been written to, the next iteration can start regardless of whether the previous one has finished.

Inter-stage registers The inter-stage registers hold temporary/intermediate values used throughout the loop, and are only accessible by operations on adjacent

stages. As a result, if a value is generated on one stage but only used on a stage much further down the pipeline, data forward operations must be placed into the intermediate stages in order to carry the value down the pipeline to the stage where it is required.

Loop constant registers The loop constant registers are used to hold values that are read, but not written to, during the execution of the pipeline and these are directly accessible from every pipeline stage. Although their functionality could be implemented by a combination of the data forward operations, source/result and inter-stage registers, the use of a dedicated set of registers for this function significantly reduces the hardware resources that are required.

Because this hardware architecture uses a distributed register file instead of the centralized one found in most processors, there are no limitations imposed by the number of registers or read/write ports. It is also worth noting, that this approach is ideally suited to modern FPGAs which have flip flops distributed evenly throughout their logic fabric.

Execution in the data flow pipeline is split into three distinct phases: initialization, execution and retirement. In the initialization phase, the initial values are written into the loop constant and source/result registers. Once this operation is complete, the pipeline is triggered and the data flows down the pipeline and is processed as it passes through the operations on each stage. When the source/result registers have been filled with the all data required for the next iteration and the loop branch operation indicates another iteration is required, the next data set starts flowing down the pipeline. Once the loop branch instruction indicates no further iterations are required, and all remaining data has propagated through the pipeline, the final retirement phase is started. In this phase all the required values from the source/result registers are copied back to the appropriate registers in the host processor.

3.3 Performance evaluation

3.3.1 Test Algorithms

To evaluate the performance of the hardware conversion system described in section 3.2, three algorithms with different characteristics were chosen. These are shown in table 3.2. For each algorithm the performance of the software was compared to that of the hardware optimized version over a range of values for two critical parameters:-

Technology Scale Factor (TSF) This is a measure of how much slower the reconfigurable area of the processor is when compared to a standard fixed execution unit (i.e. the ratio of the time taken for execution in the reconfigurable logic compared to the time taken in a hardwired application specific integrated circuit (ASIC) logic). Thus the TSF is a measure of the overhead present in reconfigurable hardware when compared to standard ASIC logic. It is worth noting that the TSF does not take into account other factors that will affect performance, such as the number of execution units present in the processor.

Loop unroll factor This is the number of times that the body of the critical loop has been duplicated and is consequently equal to the reduction factor in the number of loop iterations. This increases the ILP in the code, and as this is a common software optimization technique, it becomes important to know how this effects the hardware conversion.

Algorithm	Bandwidth to instruction ratio	Loop branch point position	Instruction dependencies (no unroll)
Copy	High	Beginning	High
Half brightness	Medium	Beginning	Medium
Mandelbrot	Low	End	High

Table 3.2: Test algorithms and characteristics

3.3.1.1 Copy algorithm

This test was a simple 80 KByte copy from one area of memory to another. The main purpose of this test was to see if the hardware conversion would have a detrimental effect under the worst case scenario (i.e. a low number of highly dependant instructions which are extremely bandwidth limited).

3.3.1.2 Half Brightness Algorithm

This algorithm converts an image from the RGB colour space to the YUV colour space, halves the luminance value of each pixel and reconverts it back to the RGB colour space.

3.3.1.3 Mandelbrot Algorithm

The Mandelbrot fractal was chosen as a test algorithm because of its computationally intensive nature and minimal bandwidth requirements. In this case, performing a traditional loop unroll (i.e. replicating the body of the loop) is of little benefit, due to the high number of data dependencies. To address this, the loop is un-rolled by processing multiple pixels in a single iteration. Since there are no data dependencies between adjacent pixels, a significant performance improvement in the software case is produced and this operation is typical of the type of optimization that a software developer might perform.

3.3.2 Loop Dependencies

Software loops commonly contain variables whose values are used to determine if another iteration of the loop is required. Listing 3.3 shows a simple code loop that

```
     iadd    r2, 0 0
loop:
     aload   r5, r4 r2
     isub    r5, r5 -1
     astore  r5, r4 r2
     iadd    r2, r2 1
     neq     g2, r2 10
 g2  ajump   loop
```

Listing 3.3: Example code loop with dependency

```
     iadd    r2, 0 0
loop:
     iadd    r6, r2 0
     aload   r5, r4 r6
     isub    r5, r5 -1
     astore  r5, r4 r6
     iadd    r2, r2 1
     neq     g2, r2 10
 g2  ajump   loop
```

Listing 3.4: Example code loop without dependency

decrements the values in a ten element array. In this example the load and store operations use the value of the loop counter "r2" as an offset into the array and as a result, the loop counter cannot be incremented until the store operation has been issued. When the loop is converted into hardware, an inter-iteration dependency is created that reduces the number of simultaneous loop iterations that can be executed, impairing performance. By replicating the value of the loop counter as shown in listing 3.4, this dependency can be removed together with the associated performance limitation. Where applicable, this optimization was performed by hand on the test algorithms and the performance was then compared to the original non-optimized test case.

3.4 Results

3.4.1 Copy algorithm

Due to the small number of instructions present in this algorithm, the instruction combination conversion was not able to identify any candidate groups of instructions and as a result no hardware acceleration was performed.

The results for the copy algorithm with the loop conversion applied are shown in figure 3.4. These graphs have been scaled so that the effects of the hardware conversion can be clearly observed and as a result the minor fluctuations in performance (0.6%) that are produced by the pseudo-random nature of the cache, appear to be quite large.

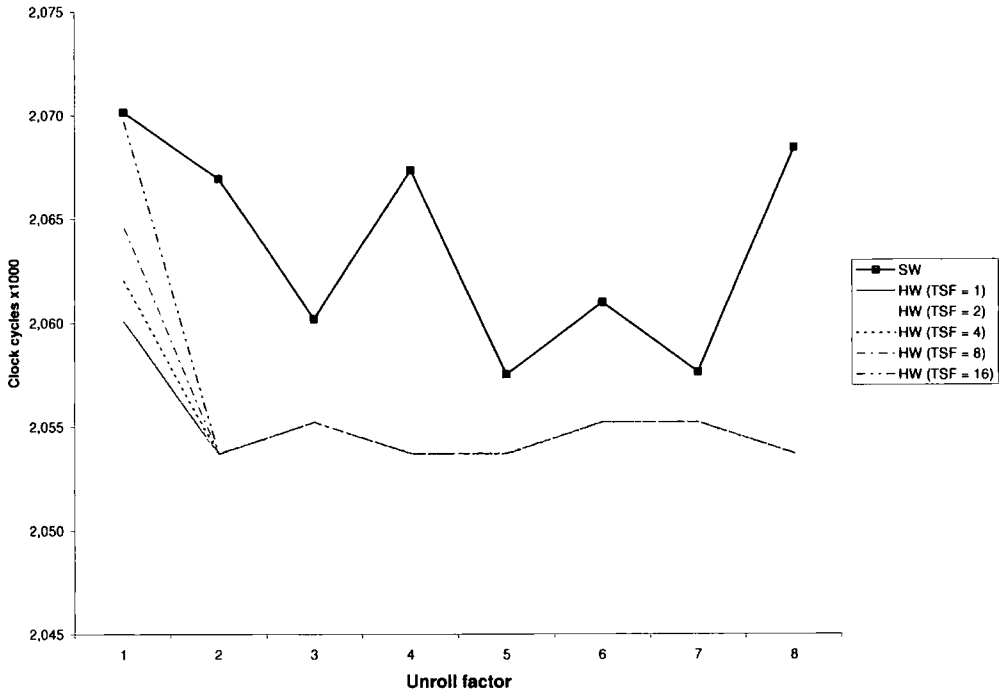
As shown in figure 3.4(a) the hardware conversion produces a marginal increase in performance even in the worst test case (Unroll factor = 1, TSF = 16). However, as the unroll factor increases, the performance of all hardware accelerated test cases quickly reaches the bandwidth limit.

By removing the loop iteration dependency, as described in section 3.3.2, the performance of all hardware accelerated test cases reaches the bandwidth limit, as shown in figure 3.4(b). Removing this dependency had no effect on the performance of the software only case.

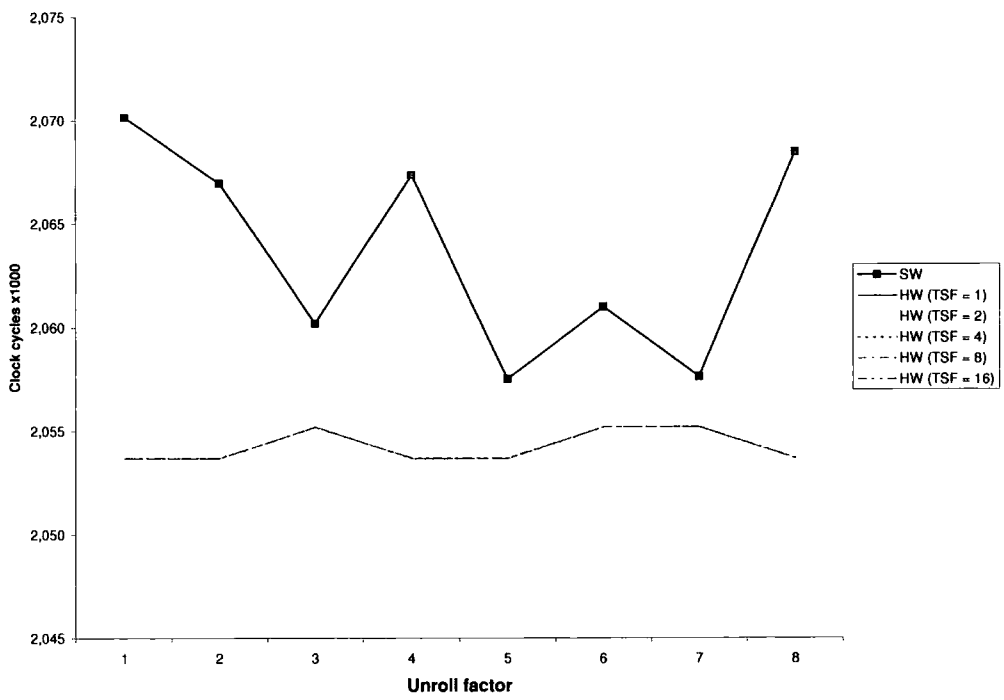
Overall the performance improvement ranged from 0.02% to 0.8%. Although these improvements are minimal and would not normally warrant the hardware resources used, this demonstrates that the loop conversion algorithm performs well even under the extreme condition of a low number of highly dependent and bandwidth intensive instructions.

3.4.2 Half Brightness Algorithm

Figure 3.5(a) shows the increase in performance generated by the instruction combination when compared to the normal software case. At best, the conversion produced a 6.8% speed improvement with a TSF value of one. Additionally, as the TSF value is increased this marginal gain quickly turns into a reduction in performance. Unrolling the loop increases the amount of ILP. This reduces the performance improvement of the instruction combination with low TSF values and also reduces the



(a) Dependency present



(b) Dependency removed

Figure 3.4: Performance improvement of copy algorithm with loop extraction

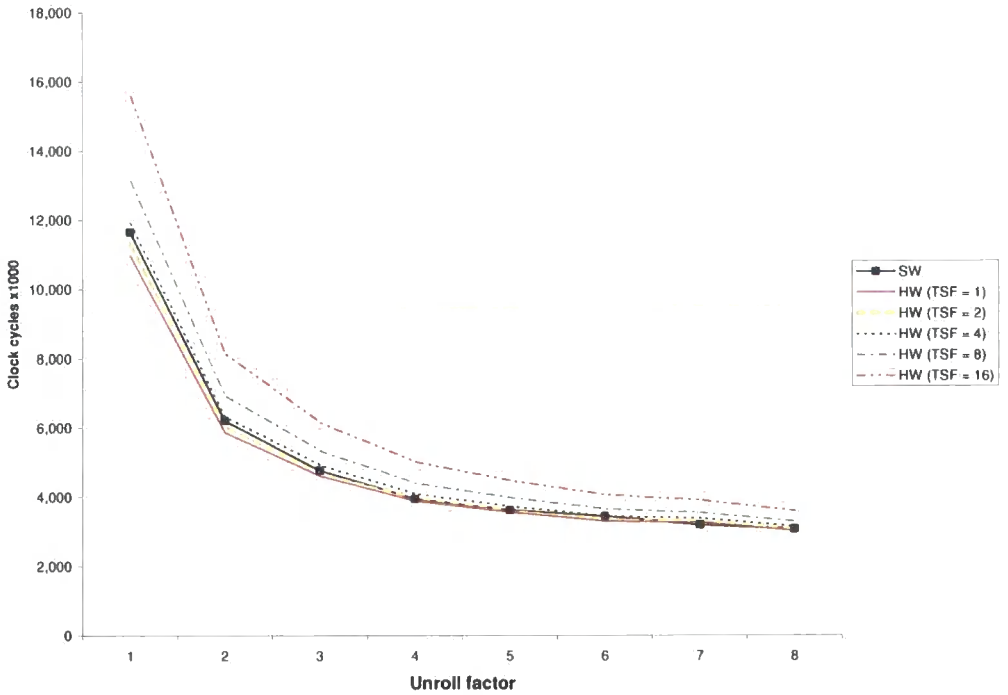
performance degradation produced at higher TSF values.

As shown in figure 3.5(b), removing the loop iteration dependency increased the performance of both software and hardware accelerated test cases by around 9%. However, this improvement in performance quickly diminishes as the unroll factor is increased. Again, this is due to the increased ILP associated with unrolling the loop.

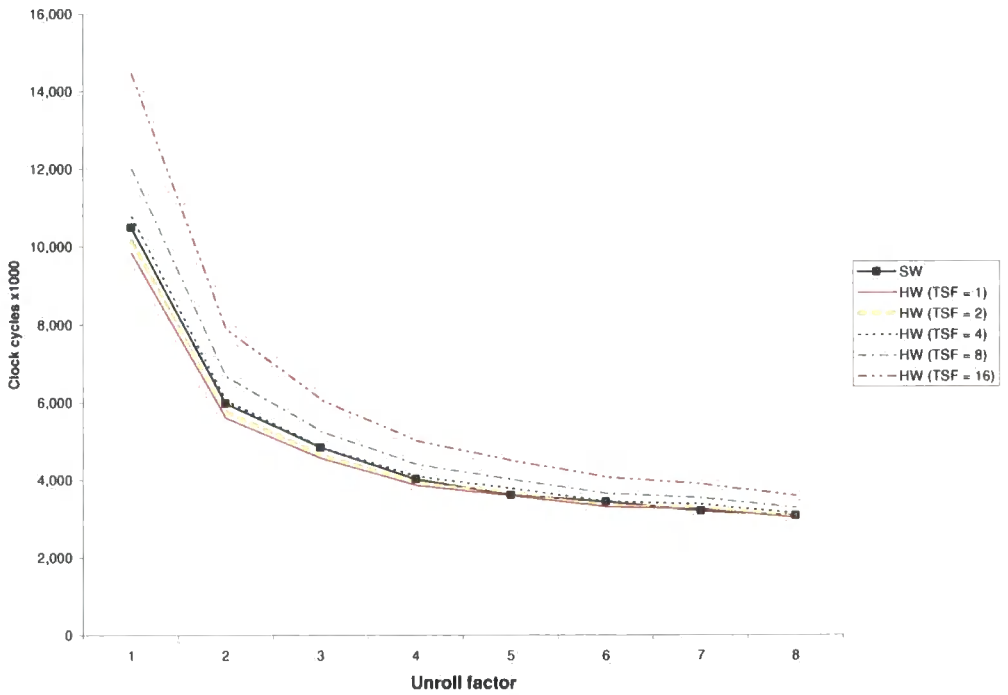
The performance improvement provided by the loop conversion algorithm is shown in figure 3.6(a). A performance improvement of 3.1x is demonstrated in the ideal case of a TSF value of one. This decreases to 67% as the unroll factor is increased to eight. More realistic values of the TSF produce a substantial reduction in performance. At high unroll factors, the performance of the software is limited by the maximum issue rate of the instruction pipeline. Since this does not affect the hardware pipeline, the performance continues to increase as the unroll factor increases. As a result, the performance degradation produced by high values of the TSF is reduced as the unroll factor increases. With the loop iteration dependency removed, all hardware accelerated test cases produced a significant performance improvement ranging from 6.6x to 1.75x as shown in figure 3.6(b). As the unroll factor is increased beyond the value of three, there is little additional improvement in the performance of the hardware due to the performance of the hardware pipeline being limited by the available memory bandwidth.

3.4.3 Mandelbrot Algorithm

As with the half brightness test, when the instruction combination algorithm is applied to the Mandelbrot test, only a marginal improvement in performance is observed with low values of the TSF (28% performance improvement, TSF = 1, unroll factor = 15). This quickly becomes a reduction in performance as the TSF

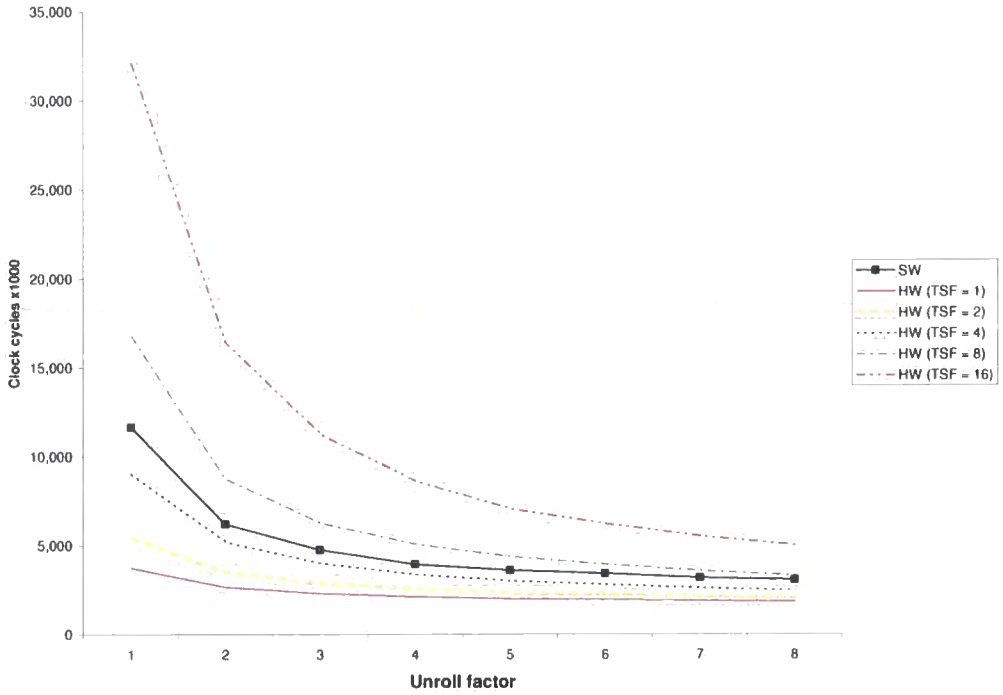


(a) Dependency present

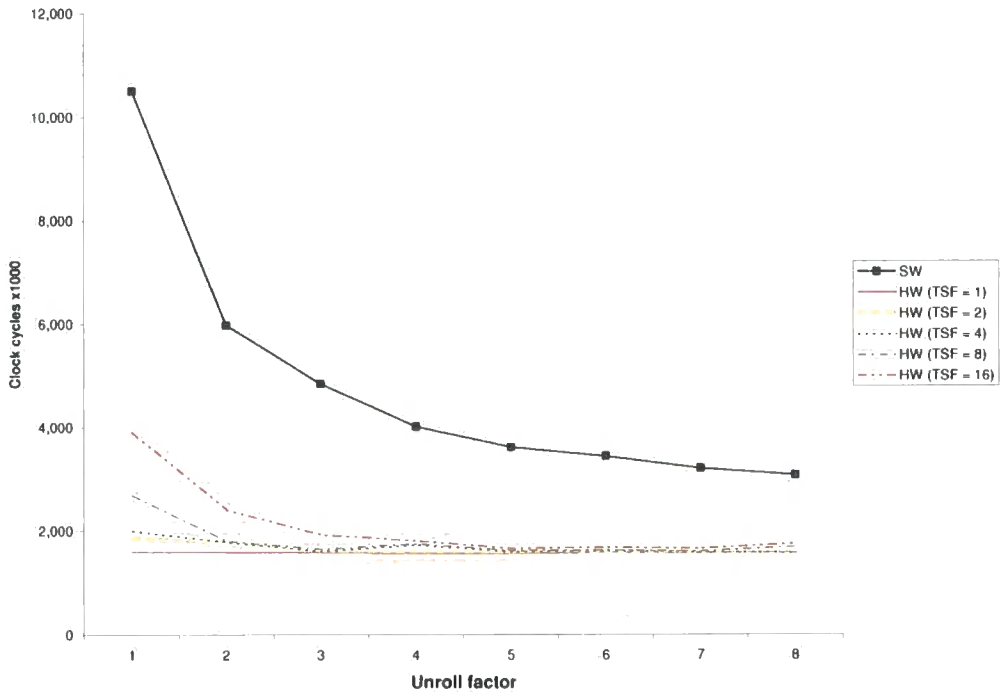


(b) Dependency removed

Figure 3.5: Performance improvement of half brightness algorithm with instruction combination



(a) Dependency present



(b) Dependency removed

Figure 3.6: Performance improvement of half brightness algorithm with loop extraction

increases, as shown in figure 3.7. It was not possible to perform loop dependency removal on the Mandelbrot algorithm, because the decision as whether or not to perform another iteration, is based on the resultant values from the body of the loop and not on values that are available earlier in the iteration.

Figure 3.8 shows the effects of the loop conversion algorithm on the Mandelbrot test. As the unroll factor increases, the performance of the software only case increases until the issue rate of the instruction pipeline becomes the limiting factor. Since the data flow pipeline is not affected by either the instruction issue rate or by the available bandwidth, its performance continues to scale almost linearly throughout all the test cases. This results in a maximum speed improvement of 6.4 times with a TSF of one and an unroll factor of sixteen. As shown previously, the performance improvement degrades as the TSF increases. With a TSF value equal to eight the performance becomes worse than that in the software only case.

3.4.4 HW Pipeline Implementation

To validate the performance improvements obtained by the loop conversion algorithm, the model of the data flow pipeline in the simulator was replaced with a wrapper block linking the simulator with a real pipeline, which was implemented using FPGAs as shown in figure 3.9. To keep the simulator synchronized with the hardware, the master simulator clock was used to clock the FPGAs. The parallel port was chosen for the interface between the FPGAs and the PC due to its low latency characteristics. This was necessary in order to achieve an acceptable simulation speed, given the clock and DMA connections between the FPGAs and the simulator. Additionally, the hardware conversion tool was modified to produce VHDL for the FPGAs instead of a simulator model of the pipeline.

The following factors determined which test cases could be implemented in the

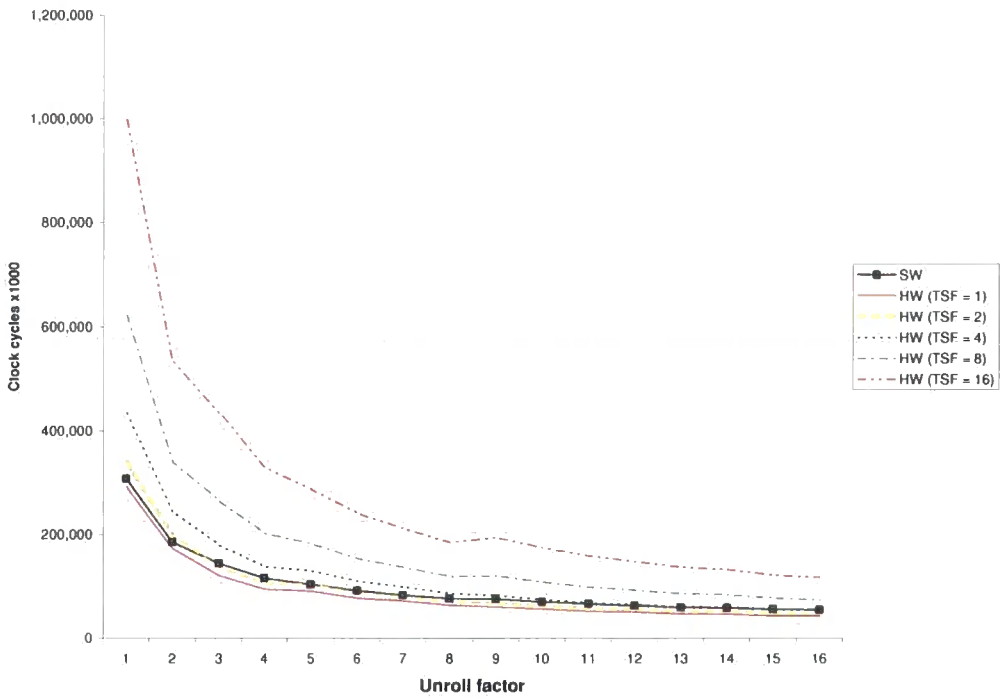


Figure 3.7: Performance improvement of Mandelbrot algorithm with instruction combination

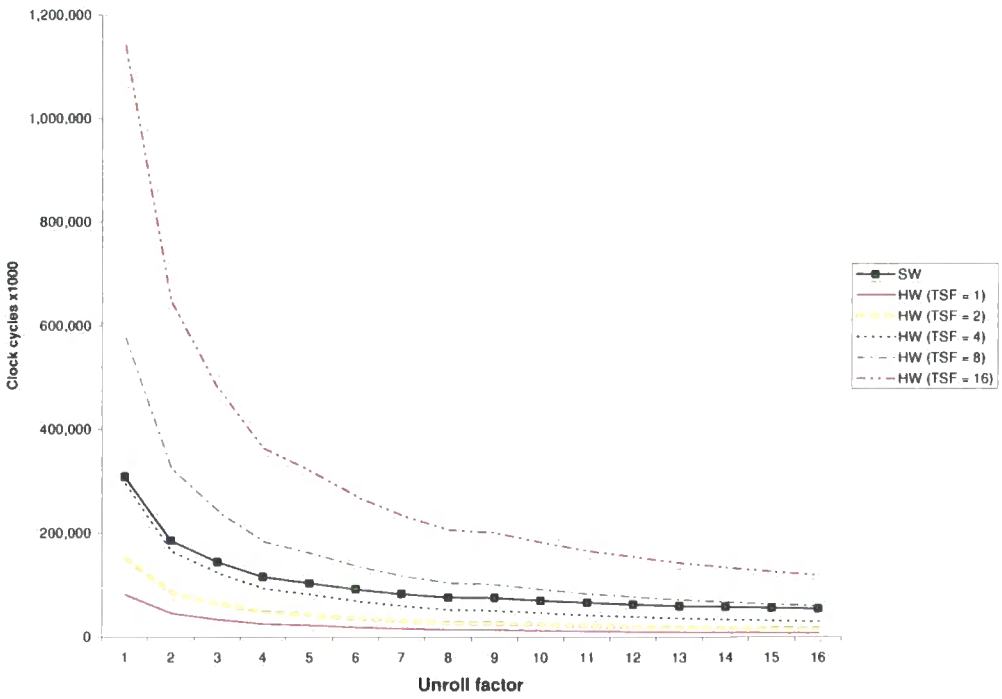
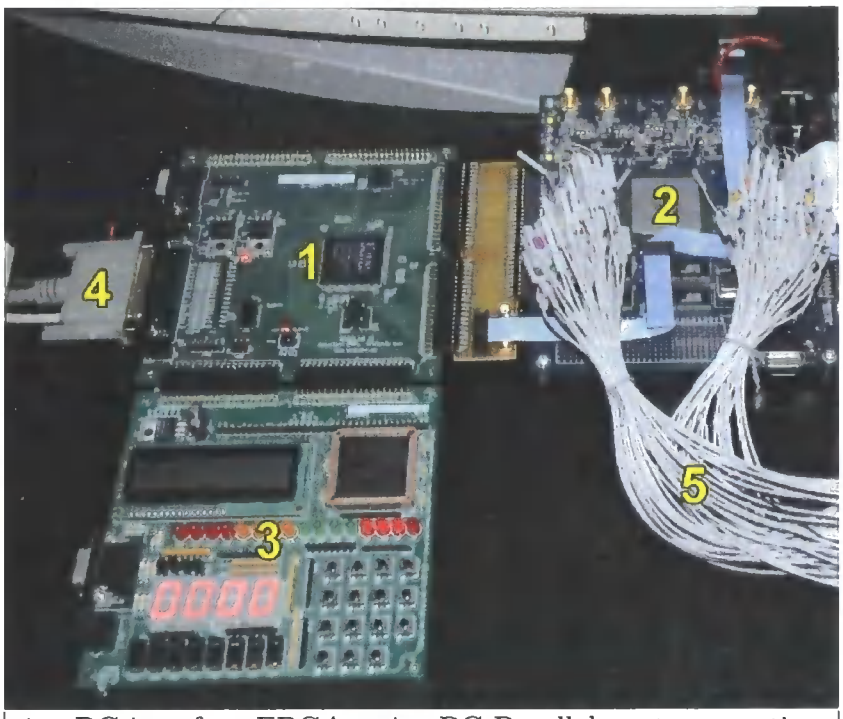


Figure 3.8: Performance improvement of Mandelbrot algorithm with loop extraction



1	PC interface FPGA	4	PC Parallel port connection
2	RC pipeline FPGA	5	Logic analyzer connections
3	Monitoring tile		

Figure 3.9: FPGA tiles used to implement real hardware data flow pipeline

FPGAs:-

Hardware resources The FPGA used to implement the data flow pipeline was an Altera APEX 20K1500. As this device contains 51,840 logic cells, it is not possible to implement a pipeline that requires a higher logic cell count.

Floating point maths Due to time constraints, floating point primitives were not created for the FPGAs. Therefore, it was not possible to implement any pipeline that contains floating point instructions.

TSF To simplify the generation of the VHDL, only pipelines with a TSF value of one were implemented in hardware.

Table 3.3 shows which tests were re-implemented using the FPGA. In each case, the number of clock cycles required to run the test was exactly the same as the simulation model. Even though only a limited number of test cases could be implemented in the

FPGA, the fact that the clock cycle counts precisely matched those in the previous simulations provides a reasonable level of confidence in the accuracy of the remaining test cases.

Algorithm	FPGA implemented test cases		
	Interaction dependency removed	TSF value	Unroll factors
Copy	No	1	1 - 8
	Yes	1	1 - 8
Half brightness	No	1	1 - 4
	Yes	1	1 - 6
Mandelbrot	None		

Table 3.3: Test cases implemented in FPGA hardware

3.4.5 Summary

The instruction combination algorithm produced only marginal improvements in performance, due to a limited exploitation of the available parallelism and also because of the bottlenecks imposed by the instruction pipeline. The improvements were only observed when extremely low and unrealistic values were chosen for the TSF.

The Loop conversion algorithm significantly improved performance over a wide range of TSF values. This is especially true when it was combined with the loop iteration dependency removal optimization. This conversion technique even managed to produce a marginal performance improvement under the extreme conditions that were present in the copy test cases (i.e. low unroll factors, high TSFs and high bandwidths). As such, this hardware generation algorithm is a viable candidate for further investigation and optimization. To this end an enhanced version of the algorithm was developed and is described in chapter 4.

Chapter 4

Loop Conversion

During the evaluation of different hardware conversion algorithms in section 3.4, it was concluded that the loop conversion algorithm had the greatest potential to form the basis of a viable reconfigurable computing platform. Accordingly, this conversion algorithm was re-implemented with additional features and optimizations that would both increase performance and also allow it to be evaluated on a wider range of test systems. In order to achieve the goal of a completely abstracted reconfigurable computing system this entire tool flow and set of optimizations would be performed at runtime. A block diagram of this loop conversion process is shown in figure 4.1. This system contains two abstraction layers which are shown in green:-

Platform abstraction layer This isolates the conversion algorithms from the host platforms, allowing the tool to be easily and quickly ported to different computer systems.

Output target abstraction layer This layer separates the pipeline generation stage from the physical implementation details of the RC area, thus enabling the same conversion system to be used with different hardware platforms, as well as the behavioural simulation system described in section 2.2.

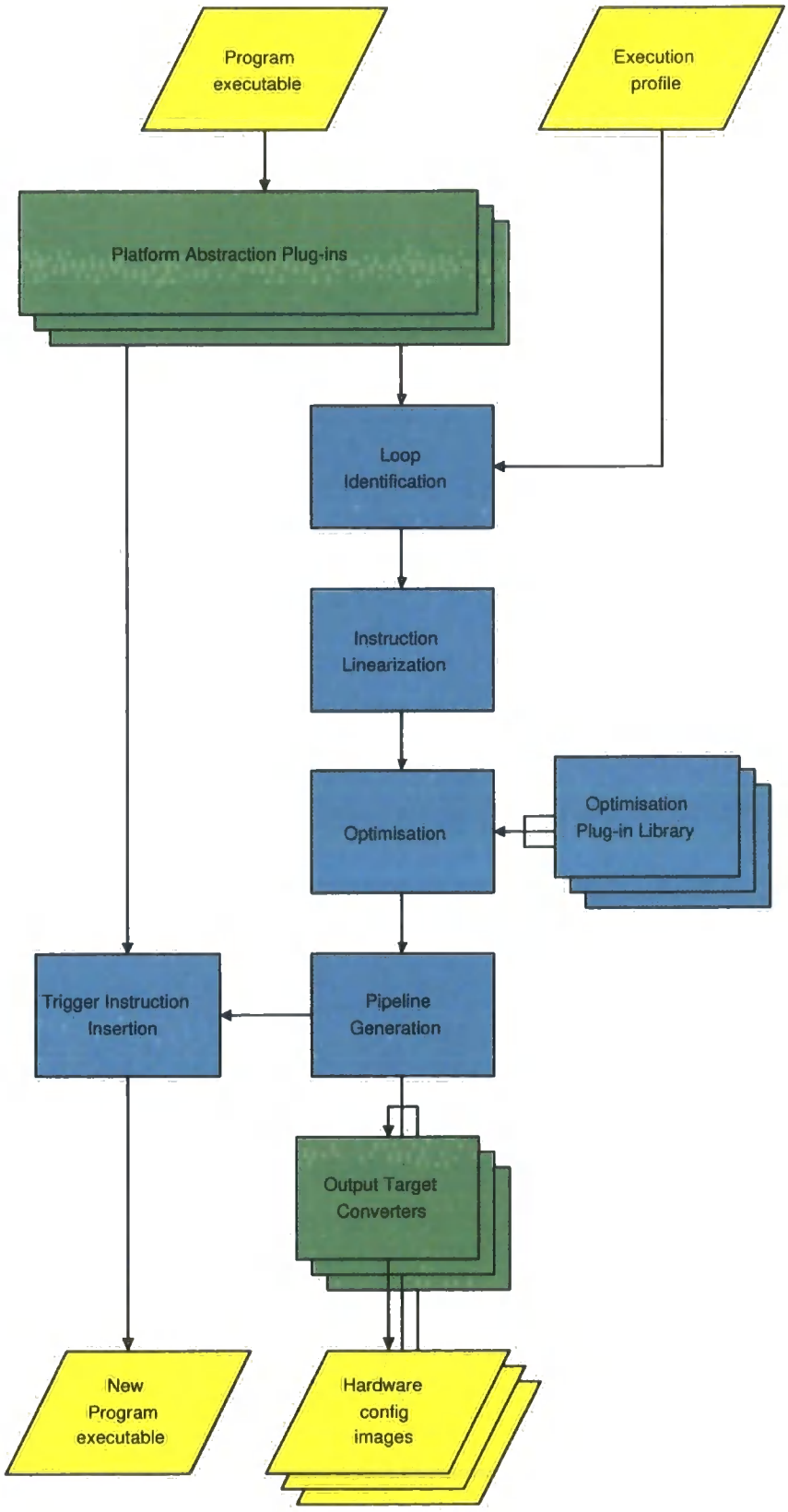


Figure 4.1: Simplified Loop conversion block diagram

In addition to having the capability of working with a variety of host platforms and target RC areas, the new loop conversion tool flow is also able to convert to hardware, several computationally intensive loops from the same program. It is worth noting that, the tool flow operates on small sections of the program in situ and therefore avoids the need to disassemble the entire program. This significantly speeds up the conversion process when it is applied to large applications.

4.1 Abstract Instruction Model

As part of the platform abstraction layer the instructions in the source program are converted to an internal abstract instruction model (AIM). The abstract instructions are generic in nature and can be used with any input parameter type (i.e. static value, register, constant register) unlike conventional instruction where the type of the parameters are usually fixed when the instruction set is defined. This flexibility reduces the number of abstract instructions that must be supported as several instructions from the same CPU ISA can be mapped onto the same abstract instruction (e.g. the MIPS XOR and XORI map to the same abstract instruction). The generic nature of the abstract instructions also means that instructions for different ISAs will map onto the same abstract instruction (e.g. both the MIPS and IA64 ISAs have 32 bit addition operations). Most of the abstract instructions correspond to hardware primitives that can be instantiated in the RC area (e.g. an 32 adder), any instructions that do not have hardware primitives must be removed by the optimisation steps in the conversion process or the hardware generation will fail.

To utilise the essentially unlimited number of registers (due to the 1:1 flipflop to LUT ratio) that can be instantiated in a hardware pipeline, the AIM must also have an unlimited number of registers. During the mapping from CPU ISA to AIM each CPU register is mapped onto a different AIM register, this therefore leads to

an initial register utilisation equal to the number of registers present in the host ISA. The various optimisation steps performed (described in sections 4.5 and 4.5.1) during the hardware conversion remap the register numbers so that a much larger number of registers are used than were present in the original host ISA, this increases ILP and therefore performance.

The AIM is a representation of what operations are performed, together with information about what data is passed between operations (and not how it is passed). Therefore the AIM can represent a wide variety of instruction sets including EPIC, RISC, and CISC. To concept of using an AIM to represent a wide variety of ISAs (e.g. MIPS [90], IA32 [86]) is a proven concept and is extencivly used by Transitive [91].

4.1.1 Supporting Additional ISAs

The following steps must be performed to add support for a new instruction set:-

Write instruction decoder An instruction decoder must be written to take the binary representation of the instructions and translate them into the op-codes and the register numbers/parameter values. The instruction decode translates from the CPU specific ISA to a format suitable for the AIM.

Create mapping tables The mapping table contains all the op-codes in the ISA together with references to the abstract instructions they are equivalent to. NULL references are placed in the tables for any any op-code that can not be supported in the AIM (e.g. cache instructions).

Implement additional abstract instructions The majority of instructions are common to nearly all CPU ISAs. As a result the same abstract instructions can be re-used for many host CPUs. However some rare instructions will be

specific to certain CPUs (e.g. vector instructions like MMX). Although such instructions could be left out of the AIM this would prevent the conversion system from processing any code that contained such instructions. The best approach is to add the CPU specific instructions to the AIM. This is a simple matter of writing a small VHDL fragment (called a primitive) that performs the same function at the instruction, and added it to the AIM table along with various estimates of its latency and hardware resource usage.

General information In addition to detailed information about the instructions, the hardware conversion system requires some general information about the host platform. This includes the number of branch delay slots, the register the stack pointer is stored in, and which instructions will trigger execution in the RC area.

Physically the AIM is separated from the CPU ISA by Java interfaces [76]. This enforce the strict rules defined by the abstraction layer, and provides a starting point when implementing front ends for new host CPUs.

4.2 Targeting The Hardware Pipeline

The final hardware is a pure data flow pipeline it does not process instructions in the same way as a conventional CPU. Instead each instruction in the original software becomes a separate piece of hardware in the pipeline. The following characteristics enable the creation of an efficient hardware pipeline that produce significant increases in performance:-

Linear non-branching code A hardware pipeline can not handle non-linear instructions streams, however software typically contains branching code. Therefore all branching must be removed (see section 4.4).

High levels of ILP The higher the level of ILP the more parallelism will be exploited be the final hardware pipeline. Several optimizations are performed during the conversion process to increase the levels or ILP (see sections 4.5.5, 4.5.1 and 4.6.1).

Low memory bandwidth requirements In many cases the performance of the hardware pipeline is limited by the available memory bandwidth. Although memory bandwidth is primarily a characteristic of the original algorithm and the way the software was written some optimizations can be performed that will reduce the bandwidth required (see section 4.5.2).

No inter-iteration dependencies As the hardware pipeline can simultaneously process several iterations the presence and position of inter-iteration dependencies (see section 3.3.2) will significantly reduce performance.

Instruction complexity The latency of each instruction is related to its complexity (e.g. a multiply has a much higher latency than a bitwise AND). The lower the latencies the more instructions can be packed onto a single pipeline stage, therefore leading to higher levels of ILP and performance.

Sections 4.3 to 4.6.1 describe the processes and optimizations that are performed to make the software suitable for implementation in a hardware data flow pipeline.

4.3 Loop Identification

The first stage in the loop conversion process, is to inspect those sections of the program surrounding the branch target addresses in the execution profile. This process allows the algorithm to determine which branches are due to loops and which branches are due to other control flows, such as "`if`, `else`, `switch`" statements. Candidate loops are then discarded if they fail to meet the following criteria:-

Hardware implementable instructions All the instructions in the candidate loop must be capable of being converted into hardware (whilst this is true of the vast majority of instructions, some classes of instructions cannot easily be implemented e.g. traps, breaks, cache operations).

Function calls or nested loops The loop must not contain any nested loops or function calls (although this appears to be a major limitation, small functions will usually be in-lined by the compiler, and if a nested loop is present that loop itself will still be considered as a candidate for conversion).

Once the candidate loops have been identified, a measure of their computational intensity is calculated by the hardware conversion tools (equation 4.1a shows the simplest way of doing this). However, as many loops contain "`if`, `else`, `switch`" statements, the total number of instructions can be very different from the number of instructions that are executed per iteration. To allow for this, a worst case value is calculated using equation 4.1b. This rough measure of execution time is used, together with an estimate of the required hardware resources, when deciding which loops should be implemented in the RC area.

$$\text{Loop exec time} \approx \text{Num iterations} \times \text{Total num instructions} \tag{4.1a}$$

$$\text{Maximum exec time} \approx \text{Num iterations} \times \text{Num instructions in longest path} \tag{4.1b}$$

4.4 Instruction Linearization

The target RC area is used to implement a data flow pipeline similar to the one described in section 3.2.2.1. Since this type of pipeline cannot directly implement the branches in the execution flow that are common in the software domain, the loop conversion system removes all branches by using a combination of two techniques: multiplexer (MUX) insertion and instruction guarding. This process is performed

```
  blez r2 elseBody
  r3 add r3 1
  b end
:elseBody
  r3 sub r3 1
  r1 mov r2
:end
```

Listing 4.1: If-else statement implemented with branches

```
g0 blez r2
r5 add r3 1
r6 sub r3 1
r4 mov r2
r1 mux g0 r1 r4
r3 mux g0 r5 r6
```

Listing 4.2: If-else statement implemented with MUXs

recursively, in order that complex program flows containing sets of nested branch instructions can be converted to hardware. The end product of the instruction linearization stage, is a serial stream of non-branching instructions that perform the same function as the original loop body.

4.4.1 MUX Insertion

Listing 4.1 shows the way in which an "if/else" statement is coded in Assembly. First, the branching is removed by executing both branch paths, and then one or more MUXs, as shown is listing 4.2, are appended (note that the MUXs are controlled by a boolean value generated by the original branch instruction). If a register is changed in both branch paths (e.g. "r3"), then the corresponding MUX will select between the two newly generated values. If however, the register value was only changed on one of the branch paths (e.g. "r1"), then the MUX will select between the original value and the value generated by the branch path. MUX insertion may lead to redundant instructions being present, an example of this is the "mov" instruction in listing 4.2, however this will be removed by the optimization process described in section 4.5.4.

```
  blez r2 end
  sw r3 r4
:end
```

Listing 4.3: Conditional store implemented with branching

```
g0 blez r2
sw g0 r3 r4
```

Listing 4.4: Conditional store implemented with guarding

4.4.2 Instruction Guarding

MUX insertion cannot be used on instructions that have effects external to the CPU core/RC area, since both paths are executed regardless of the branch condition (e.g. store instructions). A guard [88] value is generated by the original branch instruction to control whether or not the store instruction performs the memory operation when it is executed (see listings 4.3 and 4.4). Although not necessary for correct pipeline function, guarding of the load instructions is also carried out to reduce memory bandwidth and consequently increase performance.

4.5 Optimization

A plug-in interface is provided to accelerate the development of additional optimizations, allowing any number of optimizations to be applied to the abstract representation of the serial instruction stream. It is interesting to note that, generally, as the number of optimizations is increased, the overall conversion time decreases. Although these optimizations slightly increase the time taken to generate the pipeline, they also simplify its structure thus reducing the time required to perform the PAR (by far the most time consuming stage of the entire process). The following optimization plug-ins have been developed:-

- Hardware dependency removal
- Stack removal

- Iteration dependency removal
- Instruction removal
- Tree re-balancing

4.5.1 Hardware Dependency Removal

The amount of ILP present is often limited by hardware dependencies, where instructions share a common hardware resource that prevents them from being executed in parallel. This can be seen in listing 4.5 where the temporary register "r2" is used by two sets of otherwise independent instructions. This dependency can be removed by remapping some of the instructions so that they use a new register "r5" as shown in listing 4.6. This is similar to the register renaming [92] technique used in superscalar processors. However, the increase in ILP produced by register renaming may be limited by the number of physical registers present in the CPU. Since, it is possible to create as many registers as may be required in an RC area, hardware dependency removal can yield higher levels of ILP in a reconfigurable computing environment than it would in a conventional superscalar processor.

```
r2 srl r1 1
r3 or  r2 1
r2 add r1 1
r4 and r2 255
```

Listing 4.5: Hardware dependency present

```
r5 srl r1 1
r3 or  r5 1
r2 add r1 1
r4 and r2 255
```

Listing 4.6: Hardware dependency removed

4.5.2 Stack Removal

Since CPUs have fixed numbers of registers, if a section of code requires more registers than are present, temporary values must be "pushed" and "popped" to and from

```
loop:
  sw r1  0(sp)
  r1 srl r2 1
  r2 or  r2 31
  r2 or  r2 r1
  r1 lw  0(sp)
  bne r3 r1 loop
```

Listing 4.7: Stacking ("push" first)

```
loop:
  r5 srl r2 1
  r2 or  r2 31
  r2 or  r2 r5
  g0 bne r3 r1 loop
  sw g0  r1 0(sp)
```

Listing 4.8: Stack operations removed ("push" first)

the stack, as shown in listing 4.7. Stacking, increases the number of instructions that need to be executed, decreases ILP, and increases the amount of bandwidth required. Stack operations may be removed on a reconfigurable computing platform by creating additional registers, as shown in listing 4.8. This increases the amount of parallelism and also reduces the bandwidth required by the RC area. To maintain consistency with the original code, a single "push" to the stack is performed on the last iteration of the loop, so as to leave the stack in exactly the same state as the software which the hardware pipeline replaced.

In some situations the stack "pop" occurs before the "push", as shown in listing 4.9. The majority of the stack operations may still be removed by creating an additional register. However, the "pop" operation remains in the loop, guarded by a flag that is true for the first iteration (see listing 4.10). This allows the initial value to be retrieved from the stack. The "push" operation is again guarded so it only occurs on the final iteration of the loop.

4.5.3 Iteration Dependency Removal

As shown in section 3.4, removing the loop iteration dependency (see section 3.3.2) can have a significant effect on the overall performance of the system. As a result, the optimization is automatically performed by this plug-in.

```
loop:
  r1 lw   0(sp)
  r1 add  r2 r1
  sw r1   0(sp)
  r1 srl  r2 1
  r2 or   r2 r1
  bnez r3 loop
```

Listing 4.9: Stacking ("pop" first)

```
loop:
  r6 lw   isFirst 0(sp)
  r1 mux  isFirst r1 r6
  r1 add  r2 r1
  r5 srl  r2 1
  r2 or   r2 r5
  g0 bnez r3 loop
  sw g0   r1 0(sp)
```

Listing 4.10: Stack operations removed ("pop" first)

4.5.4 Instruction Removal

Because of the limited capabilities of CPU instruction sets, it is quite common for code to contain instructions that can be removed without affecting the functionality. These instructions fall into two categories:-

Immediate values Many instructions can only take values directly from the register file. In addition, instructions that are able to take immediate values can, usually, only accept small numbers (16 bits in the case of the MIPS instruction set [90]). As a result, it is common to find groups of instructions that set the value of a register, followed by further instructions that use this value (shown in listing 4.11).

Copy instruction Due to the centralized register file architecture present in most processors, copying values from one register to another is a common process. An additional source of these copy instructions, is the linearization process described in section 4.4.1.

Due to the flexible nature of the RC area compared to that of a fixed instruction set, it is possible to remove both of these types of redundant instructions as shown in listing 4.12.

```
r1 lui 0x802
r1 or  r1 0xa578
r1 lw  r1
r4 mov r1
```

Listing 4.11: Before instruction removal

```
r4 lw 0x802a578
```

Listing 4.12: After instruction removal

```
r1 or r1 r2
r1 or r1 r3
r1 or r1 r4
```

Listing 4.13: Sequential value combination

```
r5 or r1 r2
r6 or r3 r4
r1 or r5 r6
```

Listing 4.14: Balanced value combination

4.5.5 Tree Re-balancing

If a series of values need to be combined, it is common for compilers to generate a sequential stream of instructions, as shown in listing 4.13. Although this approach minimizes the number of temporary registers required and consequently the need for stacking, the resultant code has very low levels of ILP. By using additional temporary registers and rearranging the way in which the values are combined, it is possible to increase the ILP as shown in listing 4.14. This is similar to the concept of balancing binary trees [93].

If all the source values for the tree are produced on the same stage of the data flow pipeline, then the tree could be balanced so that each data path is of the same length. However, this is not always be the case. For example, if one of the source values is not available until much further down the pipeline, the result would be a reduction in performance. To compensate for this, the balancing algorithm estimates at which point in the pipeline each source value becomes available and skews the tree accordingly.

It is important to note that the number of stages required for the pre optimization

sequential case, scales linearly with the number of input values (see equation 4.2a), whilst in the post optimization balanced tree case this scaling becomes logarithmic (see equation 4.2b). Therefore, the performance improvement produced by this optimization increases dramatically as the number of input values increases.

$$\text{Num stages (Sequential)} = \text{Num values} - 1 \tag{4.2a}$$

$$\text{Num stages (Balanced)} = \log_2 (\text{Num values}) \tag{4.2b}$$

4.6 Pipeline Generation

After the code linearization and optimization steps have been performed, the pipeline is generated. A simplified overview of this procedure is shown in figure 4.2. This pipeline generation process can be broken down into the following distinct stages:-

- Operation scheduling
- Data forwarder addition
- Register remapping

4.6.1 Operation Scheduling

The position of the operations in the pipeline may be calculated by analyzing the data dependencies, the results of this calculation are shown in figure 4.2(b). RC areas are based on a regular array of fundamental logic elements. In the case of FPGAs each element usually consists of a single flipflop and a 4 input Look Up Table (LUT) [94], whereas most software instructions have only 2 input parameters. Consequently placing a single operation between registers, can result in half the inputs to the logic elements remaining unused. This leads to high latencies and inefficient

```
r10 srl r12 29
r8 srl r12 26
r8 and r8 0x1
r10 and r10 r13
r10 xor r10 r8
r9 srl r12 30
r9 and r9 r10
r1 xor r8 r10
```

(a) Sequential operations

Stage 0	r10 srl r12 29	r8 srl r12 26	r9 srl r12 30
Stage 1	r8 and r8 0x1	r10 and r10 r13	
Stage 2	r10 xor r10 r8		
Stage 3	r9 and r9 r10	r1 xor r8 r10	

(b) Scheduled operations

Stage 0	r10 srl r12 29	r8 srl r12 26	r9 srl r12 30
Stage 1	r8 and r8 0x1	r10 and r10 cr13	r9 ← r9
Stage 2	r10 xor r10 r8	r8 ← r8	r9 ← r9
Stage 3	r9 and r9 r10	r1 xor r8 r10	

(c) Scheduled operations with forwarders

Stage 0	r0 srl r0 29	r1 srl r0 26	r2 srl r0 30
Stage 1	r0 and r1 0x1	r1 and r0 cr0	r2 ← r2
Stage 2	r0 xor r1 r0	r1 ← r0	r2 ← r2
Stage 3	br1 and r2 r0	r1 xor r1 r0	

(d) Pipeline with remapped registers

Figure 4.2: Example operation scheduling onto data flow pipeline

usage of the available hardware resources. To reduce both latency and hardware usage, the estimated propagation delay of each operation is calculated based on the types of its input parameters (i.e. register or static value). If the combined delay of two or more dependent operations is less than the target propagation delay between registers, then the operations are packed into the same pipeline stage. Since this optimization is performed when the pipeline is generated, a considerable reduction in the overall length of the pipeline is produced. This is similar to the technique of register retiming [95, 96] where registers are moved forwards or backwards through the logic to even out propagation delays. However, with this technique, the total number of registers and therefore the number of pipeline stages must remain constant.

4.6.1.1 Pointer Aliasing

In addition to conventional data dependencies, pointer aliasing can also cause operations to be dependent upon each other. For example, when the addresses of two memory operations are the same, the original order of these memory operations must be maintained. Unfortunately, it is very difficult to identify which pointers are prone to aliasing [97, 98]. Simply assuming that all pointers have the potential to alias, is the safest solution. However, this approach significantly limits both ILP and loop level parallelism. One potential solution, is to perform aggressive optimizations and to implement hardware alias detection. If pointer aliasing is detected, the state of the hardware is rolled back to the point before the occurrence of the alias and the code is then re-executed with less aggressive optimizations. This technique has been successfully implemented in the Crusoe range of processors from Transmeta [99, 50]. However, this procedure could not be implemented in the current hardware generation system because the error detection stage would have to be based on exceptions and/or interrupts, which are either not implemented or not accessible on

the current host target platforms. To overcome this limitation, the user is able to specify whether or not pointer aliasing is present. This is the only user intervention that is required in the entire conversion process and this can be eliminated with the next generation of target platforms.

4.6.2 Data Forwarder Addition

As discussed in section 3.2.2.1, the lack of a centralised register file means that values generated on one stage need to be piped forward to the stage where they are to be used. This can be seen in figure 4.2(c), where the value of register "r9" is generated on stage 0 but is not used until stage 3. As a result of this, data forwarders have been added to stages 1 and 2.

If a value is read but never written to during the execution phase of the pipeline, then additional data forwarders are not required. Instead, the input to the operation is directly connected to the loop constant centralized register file, which can only be read by the pipeline. This can be seen in figure 4.2(c) where the second operation on stage 1 has the input parameters "r13" flagged as a constant.

4.6.3 Register Remapping

Initially, the instructions all accessed a global register file. Once the operations have been placed into the data flow pipeline, they are only able to access the loop constant registers and the registers on adjacent pipeline stages. The final stage of the pipeline generation process is to remap the register numbers so that each set of registers is numbered sequentially, as shown in figure 4.2(d). During this process, the number of registers required on each stage is also calculated.

Operations that produce either: the final result values or the data required for the

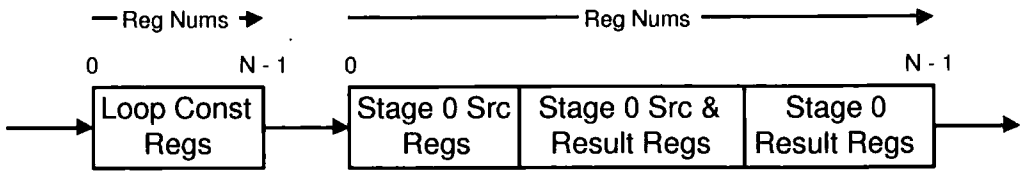


Figure 4.3: Loop constant and stage 0 shift register arrangement

next iteration of the pipeline, are remapped so that the data is written to stage 0 instead of to the next pipeline stage, as would normally be the case. An example is operation 1 on stage 3, shown in figure 4.2(d).

To eliminate the need for complex clock enables and large MUXs, the loop constant and stage 0 registers are connected together to form a shift register. Additionally, stage 0 parameters are assigned register numbers according to their usage, as shown in figure 4.3. By grouping registers that are used as source values close to the start of the shift register and by also grouping result registers close to the end of the shift register, the number of values that must be shifted into and out of the RC area is minimized.

4.7 Target Implementation

The final stage of the loop conversion process, is to generate a target specific pipeline configuration from the abstract representation produced by the pipeline scheduling phase. In addition to this, a new version of the program executable is created that utilizes the newly created hardware.

4.7.1 Pipeline configuration generation

The configuration generation is abstracted from the rest of the tool flow by a plug-in interface. This allows multiple configuration formats to be generated for different

target platforms. These include:-

MIPS hardware This plug-in generates a VHDL representation of the pipeline that can be directly synthesized using the Altera Quartus [100] software, thus producing the FPGA configuration bitstream.

MIPS simulation A simulation model of a MIPS CPU and its peripherals was created using the behavioural simulator described in section 2.2. The RC area in the simulation is configured from a file that is generated by this plug-in.

Cray XD1 hardware In order to target the FPGAs present in the Cray XD1 [67] this output plug-in creates a VHDL representation of the pipeline, which in turn is converted to a FPGA configuration bitstream by the Xilinx ISE [101] software.

In addition to producing the configuration for the RC area, these plug-ins also provide the following information to the rest of the tool flow:-

- The RC area size
- The maximum number of pipelines that the RC area can contain
- The latencies of the operations for each input configuration
- The sizes of the operations for each input configuration

4.7.2 Program modification

The register remapping information produced during the pipeline generation phase, identifies which register values need to be sent to the pipeline before execution, and which values should be copied back to the CPU register file. This information is also used to determine the order of the parameters in the trigger instructions.

```
loop:
    r1 sub r1 1
    r2 mul r2 r1
    bne r1 0 loop
    <Rest of program>
```

Listing 4.15: Program before trigger instruction insertion

```
loop:
    b trigInsts
    r2 mul r2 r1
    bne r1 0 loop
endOfLoop:
    <Rest of program>
trigInsts:
    r2 rcpt0 r1 r2 7
    b endOfLoop
```

Listing 4.16: Program after trigger instruction insertion

These trigger instructions are generated and, together with a jump instruction, are appended to the program (as shown in listing 4.15 and 4.16). Once the pipeline has finished executing, this jump instruction returns the program execution back to the main body of the software. Placing the trigger instructions at the end of the program provides two main advantages:-

- The hardware acceleration can be enabled or disabled by simply swapping the first instruction in the loop with a jump to the start of the trigger instructions at the end of the program.
- In some cases, the number of trigger instructions required will be greater than the number of instructions in the original loop. Since, only a single instruction needs to be replaced in the original loop, this removes the need to move the rest of the program up in the address space and consequently removes the need to relink.

Header information is also added to the trigger instructions at the end of the program. This can be read by the program at runtime and is used to give the end user the ability to enable or disable the hardware acceleration.

Chapter 5

MIPS Test Platform

To evaluate the performance of the loop conversion system outlined in chapter 4, a test platform was created using the MIPS architecture [102]. This was chosen due to the widespread availability of compilers, open source CPU cores, and reference manuals.

5.1 Platform Details

The MIPS test platform consists of three separate targets:-

Hardware A standalone hardware implementation of the MIPS platform and RC area was created using FPGAs. All the performance evaluations were performed on this target.

Simulation A behavioural simulation of the MIPS CPU, RC area and peripherals was created using the simulator described in section 2.2. The high level of visibility provided by this environment makes it an ideal debug tool for both the MIPS and the hardware conversion software. Because the hardware soft-

ware interface (HSI) exactly matches the standalone hardware target, the same software binaries can be run on both without either modification or porting.

Mixed simulation and hardware pipeline This target was used as both a debug system and as an intermediate stage to the full hardware solution. It combines the behavioural simulation of the CPU core with its associated peripherals with an RC area implemented in an Altera APEX 20K1500, and is similar in concept to the mixed hardware/simulation system described in section 3.4.4.

5.1.1 RC Area Integration

The MIPS test platform uses the directly connected topology outlined in section 1.2.1.1. As a result, the RC area is connected to the instruction pipeline of the CPU as an additional execution unit. To control the RC area, four extra trigger instructions have been added to the MIPS instruction set. All four of these instructions are identical except for the fact that each one refers to a different data flow pipeline within the RC area. The trigger instructions have four parameters:-

Destination register This is set to the register number where value shifted out of the pipeline will be stored. If the instruction does not return a value, then this parameter should be set to "$0", which corresponds to the hard wired zero register in the MIPS ISA.

Source register 1 & 2 The next 2 parameters are the source registers that contain the values to be loaded into the pipeline. If a source value is not required the parameter should again, be set to "$0".

Flags The final parameter is the bitwise OR of three flags that control the execution of the instruction. The first bit is set to trigger the start of the pipeline

execution; the second bit is set if the source parameters of the instruction need to be shifted into the pipeline; the third bit is set if a result parameter needs to be shifted out of the pipeline. An instruction which triggers execution in the data flow pipeline will stall until the RC area has completed execution.

An example trigger instruction sequence, for data flow pipeline zero, is shown in listing 5.1. The first instruction loads two parameters into the pipeline; the second instruction loads two additional source parameters and then triggers execution. Once execution in the RC area has completed, the first result parameter is stored in the register "a1" and the final instruction stores another result parameter in the register "a0".

```
rcpt0  $0, $v1 $v0 0x2
rcpt0 $a1, $a1 $a0 0x7
rcpt0 $a0,  $0  $0 0x4
```

Listing 5.1: Example RC trigger instruction sequence

5.1.2 Hardware

Figure 5.1 shows the hardware used to implement the standalone target. By recompiling the code for the RC area with different interface libraries, the pipeline can also be connected to the behavioural simulator for use in a mixed simulation/hardware target. This connection is made via the parallel port due to the low latency requirements of the connection.

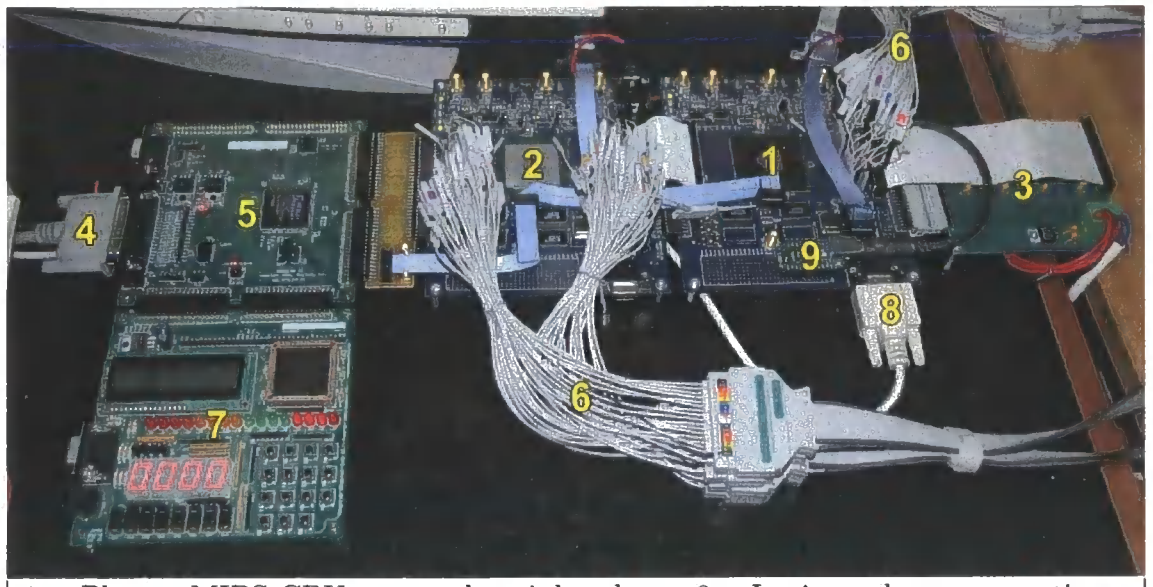
The specification of this system is outlined in table 5.1. As shown in figure 5.2, one FPGA contains the open source Plasma MIPS CPU core [103], together with its peripherals, and the second FPGA contains the reconfigurable data flow pipeline(s). An RS232 port is used to provide a command line interface to the software running

on the target. Due to the low data rates provided by this interface a USB port is also present to aid the transfer of large sets of test data to and from the target. A 64 bit clock cycle counter is present in the system, the value of which can be read by the CPU and is used to time, accurately, the execution of the various test algorithms.

The Plasma CPU core used, is open source and as such the VHDL code for the processor is freely available. Modifying the instruction pipeline to incorporate the RC area was, therefore, a relatively simple task. The processor, based on a three stage pipeline design, is capable of issuing one instruction per clock cycle and the majority of instructions will execute without stalling the pipeline. However, the multiplier, divider, and load/store units all take multiple cycles and can, therefore, stall the instruction pipeline. Since the memory interface takes multiple cycles to perform an operation, and is not pipelined, the memory bandwidth available is limited. This limited bandwidth prevents the high computational rate of the RC area from being fully realized. However, due to the relatively low computational rate of the CPU, this limitation does not significantly impact on the performance of this processor. As the memories connected to the CPU are low latency SRAMs, the lack of caches in the core has only a marginal effect on performance.

Due to the limited number of block RAMs in the APEX 20K200 used for the CPU, it was not possible to implement the code profiler block described in section 3.1.1.1. Consequently, the code profiling was performed on the simulation target. This, together with the fact that it is not possible to run the Quartus [100] synthesis tool on the platform, prevents the system being used in a runtime self adaptive mode. Instead the conversion process is performed pre-runtime using only the program binary and profile report, no access to the original source code is required.

The clock speed of both the processor core and the RC area is 16MHz, with the limiting factor being the path from the load/store unit in the processor to the



1	Plasma MIPS CPU core and peripherals	6	Logic analyzer connections
2	RC area FPGA	7	Monitoring tile
3	2 MByte SRAM & power distribution tile	8	RS232 console interface
4	PC Parallel port connection to simulator	9	USB data transfer interface
5	Misc (Monitor, CPUSim interfaces, etc)		

Figure 5.1: FPGA tiles used to implement MIPS CPU, RC pipeline, and peripherals

CPU core	Plasma MIPS CPU @ 16 MHz in Altera APEX 20K200 FPGA
RC area	Data flow pipeline @ 16 MHz in Altera APEX 20K1500 FPGA
IO	RS232 for console command line interface USB for data transfer
Memory	8 KByte ROM (read = 1 cycle) 256 KByte RAM (read = 1 cycle, write = 2 cycles) 2 MByte RAM (read = 1 cycle, write = 2 cycles)
Misc	64 Bit cycle counter for timing program execution

Table 5.1: MIPS platform summary

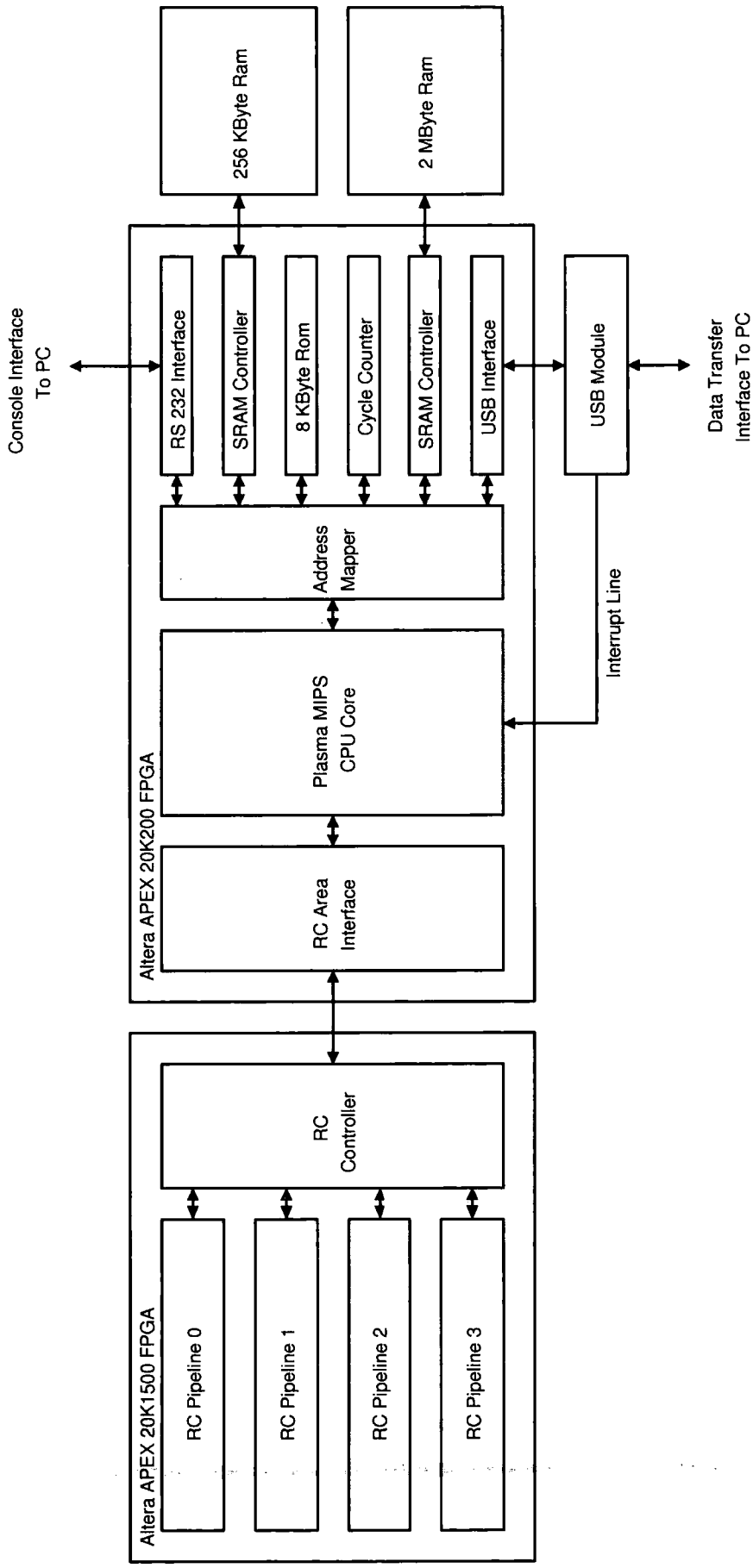


Figure 5.2: MIPS system block diagram

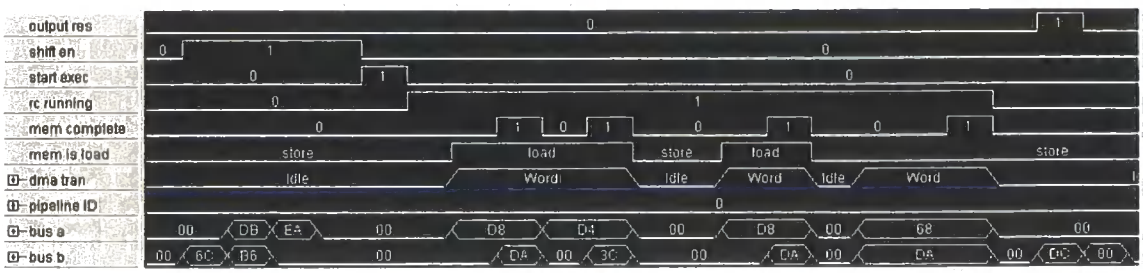


Figure 5.3: Example logic analyzer trace showing RC pipeline execution

external SRAMs. The timing report for the CPU core indicate that, it should be capable of running at 30 MHz. This would be considerably higher if the CPU core were to be implemented in ASIC logic rather than in an FPGA. In comparison, the timing report for the RC area, suggests that clock speeds in excess of 100 MHz are achievable, however, this is very dependent on the particular algorithm that is being converted to hardware (note that, routing delays are minimized as most operations in the data flow pipeline only access registers on adjacent stages, thus minimizing the length of the signal paths). In practice, the potentially higher clock rate of the RC area is unlikely to increase performance, as most algorithms will be limited by the available bandwidth, which remains constant. Consequently, it was decided to minimize the complexity of the system by using a single clock domain, which results in a TSF value of one.

5.1.2.1 CPU/RC Area Interface

A logic analyzer trace of the interface between the Plasma MIPS CPU core and the RC area is shown in figure 5.3. Execution in the pipeline is split into four stages as detailed below:-

Setup During the setup stage, initial values are loaded into the constant and stage 0 registers as described in section 4.6.3. Because the MIPS instruction set architecture (ISA) allows instructions to have two source parameters, two parallel shift registers are used to set up the data flow pipeline. On every clock

cycle where the "shift en" signal is high, two new values from "bus a" and "bus b" are shifted into the pipeline as indicated by the "pipeline ID" signal.

Trigger Once all the required parameters have been loaded into the pipeline, execution can be triggered. This is accomplished by setting the "pipeline ID" signal to the number of the pipeline that is to be triggered and by holding the "start exec" signal high for one clock cycle. On the following clock cycle, the pipeline will drive the "rc running" signal high to indicate that the pipeline is busy.

Execute During the execution stage, the RC area can issue DMA operations to the main memory. A DMA operation is started by driving the "DMA tran" signal with one of three possible values (i.e. "Word", "Half Word" or "Byte") which indicate the size of the operation that is to be performed. At the same time as the "DMA tran" signal is asserted, the "bus a" signal is used to indicate the address of the operation and "is mem load" then indicates whether the operation is a load or a store. The signal "bus b" is used to transfer the data associated with the DMA operation and is driven by either the CPU or the RC area, depending on whether the operation is a load or a store. Once the DMA operation is complete, the "mem complete" signal is driven high by the CPU. The RC area is then free to post the next DMA operation or return to the "IDLE" state. Once the pipeline has finished executing, the "rc running" signal will go low.

Retire The retirement stage is optional, and is only entered if results from the RC area need to be moved back into the central register file in the CPU. This is done by the CPU asserting the "output res" signal. At this point, both "bus a" and "bus b" are driven by the RC area with the last values from the two parallel shift registers that make up the constant and stage 0 registers. If additional results are required, the CPU will drive the "shift en" signal high

to shift out the next two values.

The limited number of connections between the MIPS CPU board and the board containing the RC area, restricts the "pipeline ID" signal to a width of 2 bits. As a result, the maximum number of simultaneous data flow pipelines that can be implemented is four. This is more than adequate for most applications.

5.2 Software

5.2.1 Console Software

The main application that runs on the MIPS platform is a console program, written in C. This presents a simple command line interface that allows the user to run a variety of test algorithms (e.g. low pass filters (LPF), FFTs, image processing, sorting, etc). In addition, the console application contains: low level drivers for the USB interface, a cycle counter and functions to test/diagnose any hardware problems that are found during the platform bring up phase.

A cut down version of the console application, without the test algorithms, is placed in the 8 KByte ROM inside the FPGA. This boots the CPU and is then used to download the much larger version of the console application, complete with the test algorithms, to the external 256 KByte SRAM (this memory is also used for the stack and all non-constant variables). This approach removes the need to resynthesize the FPGA bit stream every time that the program is updated. The remaining 2 MByte external SRAM is reserved for test data (e.g. images, audio, raw data, etc).

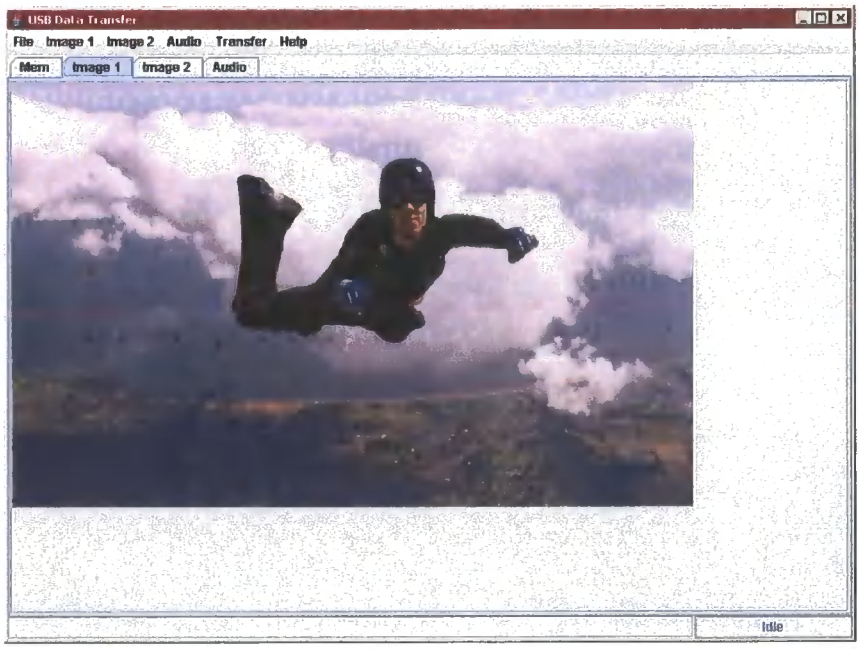


Figure 5.4: USB data transfer application displaying the contents of the data buffer as an image

5.2.2 Data Transfer Software

The data transfer application is used to transfer test data to and from the hardware target via the USB interface. At the heart of the application is a 2 MByte buffer, which can be uploaded/downloaded to/from the 2 MByte SRAM on the target. Multiple view windows, display the buffer contents in the following three formats: raw, image and audio waveform (see figure 5.4). By default, there are two image views each displaying a different area of the buffer. Test data can be loaded into the buffer from a variety of different image and audio file formats (e.g. JPEG, PNG, WAV, etc). In addition, the data can also be loaded or saved in a raw format.

5.3 Test Algorithms

The loop conversion algorithm, described in chapter 4, was applied to a series of test algorithms with different characteristics such as: the number of instructions, the bandwidth requirements, the levels of ILP, etc. The source code for all the test

algorithms (kernels of the algorithms are included in appendix A) was combined with that of the console software (see section 5.2.1) and compiled into a single application. The types of benchmark algorithm used was ristricted due to the various platform limitations outlined below:-

None static arrays None static arrays that are local to a function are allocated to the stack, to reduce the amount of stack space consumed the GCC C compiler uses miss-aligned loads and stores to access the array elements. However for patent reasons these instructions were not implemented by the author of the open source CPU core used [103]. As a result software that contains none static arrays can not be used.

Standard C functions The console software does not include a full C runtime library. Therefore most of the standard C functions can not be used (e.g. malloc, sin, fopen, sqrt, etc).

Floating point maths The CPU core does not contain any floating point maths hardware. Normally floating point support would be emulated by a maths library, however the console software does not contain such a library. As such software that requires floating point arithmetic could not be executed.

For the purposes of the evaluation, all the algorithms were run on the full hardware target. A summary of the speedups produced, together with the hardware resource utilization for the various test algorithms, is shown in table 5.2.

5.3.1 PRBS Generator

Linear feedback shift registers (LFSRs) are often used to create pseudo-random binary sequences (PRBS) [104]. The high number of bitwise operations present in this algorithm often leads to highly efficient hardware implementations. In comparison,

Algorithm	Sub algorithm	Number of loops converted to HW	Hardware usage		SW Exec time (x1,000,000 cycles)	HW Exec time (x1,000,000 cycles)	Speedup factor
			Logic cells	% of FPGA			
PRBS generator	Standard	1	218	0.4	156.2	26.2	6.0
	Unrolled	1	638	1.2	86.5	1.6	54.9
FFT	Forward	3	8207	15.8	350.5	63.5	5.5
	Inverse	4	9184	17.7	497.3	90.1	5.5
Low pass filter		1	1943	3.7	30.3	4.2	7.2
Normalization		2	2769	5.3	44.6	4.5	10.0
Block search	Packed	1	5154	9.9	856.4	273.4	3.1
	Planar	1	7005	13.5	935.2	126.1	7.4
Mandelbrot		1	2805	5.4	720.7	33.4	21.6
Half brightness		1	7319	14.1	57.9	1.3	44.3
Factorial		1	1150	2.2	220.0	20.0	11.0
Series sum		1	256	0.5	50.0	10.0	5.0
Copy		1	606	1.2	3.7	1.3	2.8
Sort	Bubble	1	939	1.8	37.7	12.6	3.0
	Heap	1	2374	4.6	39.9	17.3	2.3
	Quick	2	1551	3.0	30.7	25.5	1.2

Table 5.2: Test algorithms used on MIPS test platform

the fixed instruction pipeline present in CPUs, can often severely limit the performance of a software solution. Although there are several classes of application (e.g. Monte Carlo simulations) that use random numbers directly [105, 106], there are many other classes of application (like encryption [107]) that share with the PRBS the characteristic of having large numbers of bitwise operations.

The random number generation test, used a 32 bit LFSR to generate 2 MBytes of random numbers (see appendix A.1) and took 156.2 million clock cycles to perform in software. On application of the automatic hardware conversion, this was reduced to 26.2 million cycles. Although this represents a significant factor of 6 speedup, there is still scope for improvement, as the hardware occupied only 0.4% of the FPGA and the algorithm used was not limited by the available bandwidth. To increase the level of ILP, the inner loop was removed by completely unrolling it (see appendix A.2) and as a result each iteration of the remaining outer loop produced a completely new 32 bit value. Performing this common optimization technique, improved the software only performance by 55.4%. Although this is a major improvement in performance, it is dwarfed by that obtained by the hardware case, which is over 99 times faster than the original non-optimized algorithm and is 54.9 times faster than the optimized software version.

The performance of the hardware is limited by the poor memory bandwidth of the MIPS host. If this limitation were to be removed, the hardware would be capable of generating a new 32 bit value every clock cycle, which would triple the performance. Because the software only test case is highly instruction limited, its performance is unlikely to increase significantly. As a result, the hardware would be approximately 160 times faster than the software. Since this hardware occupies only 1.2% of the FPGA and the algorithm can be linearly scaled with increased hardware resources, the potential increase in performance, given unlimited bandwidth, is extremely large.

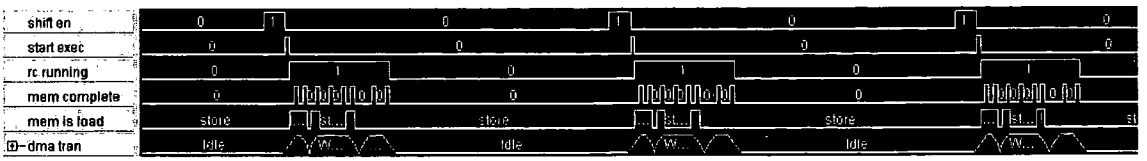


Figure 5.5: Logic analyzer trace showing RC pipeline execution of FFT algorithm

5.3.2 FFT

This test performs an integer fixed point FFT on 2 MBytes of audio data (16 bits per channel @ 44.1 KHz). Since the FFT algorithm used (see appendix A.3) can only process data sets up to a maximum size of 1024 samples, the audio data is split into blocks of this size which are then processed independently. The conversion to hardware resulted in a 5.5x speed improvement for both the FFT and the inverse FFT. To achieve this, the tools automatically converted the four innermost loops present in the FFT algorithm to hardware.

During the course of the FFT, multiple passes of the data set are performed. Due to the butterfly nature of the FFT algorithm [108, 109], the later passes of the data set result in the RC area being triggered repeatedly with only a few data values being processed on each occasion. As is shown in figure 5.5, this results in a significant amount of idle time between RC executions. This and the bandwidth intensive nature of the algorithm, are the two main performance limiting factors.

5.3.3 Low Pass Filter

A low pass audio filter was implemented using a finite impulse response (FIR) [110] design, the coefficients of which are shown in equation 5.1. As the audio data was arranged so that a single 32 bit word contains both the left and right 16 bit samples, the algorithm was implemented to filter both channels simultaneously (see appendix A.4). This reduces the number of passes of the data set and therefore increases the

efficiency of the memory sub-system.

$$y_n = \frac{4x_n + 3x_{n-1} + 3x_{n-2} + 2x_{n-3} + 2x_{n-4} + 2x_{n-5}}{16} \tag{5.1}$$

The conversion to hardware yielded a 7.2 times increase in performance and although this is a significant improvement, it is by no means the maximum that is achievable. Computing the new sample values for both channels requires eight clock cycles, five of which are spent idle while the pipeline is stalled whilst awaiting data from/to the memory. The overall performance of the system could be further increased by improving the memory controller and also by implementing a data cache in the MIPS CPU core.

5.3.4 Normalization

Normalization of audio data is commonly performed to maximize the dynamic range, thereby improving the signal to noise ratio during playback. This requires two passes of the data set. The first pass determines the maximum sample value and the second scales all the data so that the maximum value is set to the maximum possible value (see appendix A.5). The loop conversion tool was used not only to extract these loops but also to convert them into hardware. An order of magnitude speedup was obtained, using only 5.3% of the FPGA, despite the fact that the performance of the hardware is bandwidth limited.

5.3.5 Block Search

Motion estimation is a key component in such video processing systems as MPEG [111] and picture improvement algorithms [112]. The basic approach, is to divide up the image into macro blocks (16x16 pixels in the case of MPEG) and a sum of

absolute differences (SAD) algorithm is then used to determine the closest match to each block in the next frame. A motion vector for each block is then derived from the difference in position of the blocks in the two frames. Typically, this operation is only performed on the luminance component of the pixels in the YUV colour space, as this considerably reduces the processing workload with little effect on the accuracy of the resultant motion vectors.

Image data is commonly arranged in one of two pixel formats in computer graphics: packed and planar:-

Packed All the values for a single pixel are grouped together in the packed format, with the data for adjacent pixels on a line being stored in adjacent memory locations (see figure 5.6(a)).

Planar In the planar format, the colour components are grouped together to form planes, with each plane able to exist in a completely different section of memory (see figure 5.6(b)).

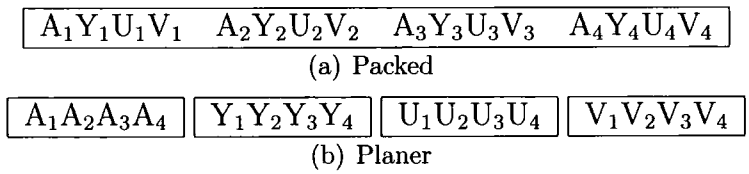


Figure 5.6: Pixel graphics formats

The block search test searches an image for the closest match to a particular 16x16 pixel macro block. This test is performed on both planar and packed pixel images (see appendices A.6 and A.7 respectively for the source code). The packed pixel implementation executed fewer instructions but necessitates a much greater number of memory operations. This is reflected in the performance results, where the planar format is 8.4% slower than the packed pixel format when both are performed in software. This situation is reversed when the algorithms are converted to hardware, as the higher number of operations present in the planar version, translates into

a higher FPGA resource utilization rather than into an increased execution time. Additionally, the significantly lower number of memory operations also contributes to a clear performance advantage for this data format. Overall, the packed pixel search was 3.1 times faster, and the planer search 7.4 times faster in hardware when compared to the software implementation. In both cases, the hardware performance was severely limited by the available memory bandwidth.

5.3.6 Mandelbrot

Rendering the Mandelbrot fractal is a highly intensive computational task, with minimal bandwidth requirements. Due to the fact that the MIPS CPU core and therefore the RC conversion tools do not support the use of floating point instructions, the algorithm was modified to use fixed point arithmetic (see appendix A.8). By converting the core loop to hardware, a speedup factor of 21.6 was obtained using only 5.4% of the FPGA.

The use of fixed point arithmetic can cause values to saturate and become corrupted, if the outer loop is unrolled to process multiple pixels simultaneously (as it was in section 3.3.1.3). Although this problem can be resolved by adding additional code, the overhead and register stacking that this introduces results in no significant increase in performance.

5.3.7 Half Brightness

The half brightness algorithm converts an image to the YUV colour space where the luminance value is halved, the image is then converted back to the RGB colour space see appendix A.9, to produce a correctly colour balanced reduction in brightness.

The performance improvement produced by the hardware conversion is due to a

Figure 5.7: Logic analyzer trace showing RC pipeline execution of half brightness algorithm

combination of two factors: performing an average of 7.2 operations in parallel on each pipeline stage and processing up to 10 loop iterations simultaneously due to the automatic removal of the loop iteration dependency. However, in practice the hardware only performs 44.3 times faster than the software solution. A logic analyzer trace shows that, the "DMA Tran" signal never returns to the idle state after the pipeline has started (see figure 5.7), indicating that the hardware is severely bandwidth limited. If this limitation were to be removed, the resulting performance would be in the order of 220 times faster than the original software.

Since only 14% of the FPGA was used to create the pipeline, the complexity of the algorithm could be increased without reducing the performance of the hardware solution, provided that the independent nature of the loop iterations was maintained. This type of scalability is un-matched in the software domain, where every additional instruction increases the execution time.

5.3.8 Factorial and Series Sum

The factorial calculation and series sum tests are both very tight loops with low levels of ILP (see appendix A.10). In order to increase the execution time and therefore obtain an accurate measure of the speedup factor, both tests were run with a starting value of 10,000,000 and, as a result, the multiplication result register overflowed. Although this means that the value calculated for the factorial is invalid, this does not affect the time taken to calculate the value.

The speed increases obtained for the series sum and the factorial were 5x and 11x

respectively. The significantly higher performance increase of the factorial, was mainly due to the poor performance of the multiplier used by the software in the MIPS CPU core. As expected, the hardware utilization for both algorithms is quite low, with that of the factorial hardware loop being slightly larger due to the multiplier.

5.3.9 Copy

The low number of highly data dependent, bandwidth intensive, instructions present in the copy test (see appendix A.11) represents one of the worst test cases that a reconfigurable computing platform would have to handle. It is important to know, therefore, what performance improvements, if any, are produced under these extreme conditions. The hardware conversion tools successfully exploit the minimal amount of ILP and loop level parallelism that is present to produce a speedup factor of 2.8.

5.3.10 Sort

Data sorting is a common task in modern computing, and consequently many different sorting algorithms have been developed [113]. Three such algorithms (bubble, heap, and the quick sort, see appendix A.12) have been tested with the hardware conversion tools. In each case a data set generated by a PRBS was used, allowing the sorting algorithms to be consistently tested with the same set of random numbers. The heap and quick sorts were performed on a 262144 element data set, whereas the bubble sort was performed on a 8192 element data set. The reduced number of elements for the bubble sort results from the very slow nature of this algorithm and the need to keep the test times manageable.

The speedup produced by the conversion to hardware for the bubble, heap and quick

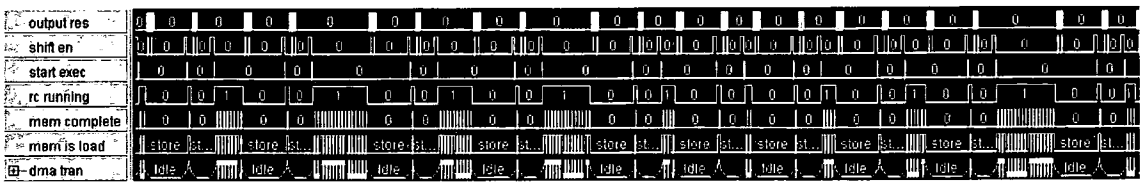


Figure 5.8: Logic analyzer trace showing RC pipeline execution of quick sort algorithm

sorts are: 3x, 2.3x and 1.2x respectively; in all cases under 5% of the FPGA was used. As expected, the quick sort is the faster algorithm in software. However, when the algorithms are translated into hardware the heap sort is 32% faster than the quick sort, due to the fact that the hardware conversion can only operate on the innermost loops. In the case of the quick sort algorithm, these loops are very small and are executed many times, resulting in the hardware pipeline being triggered repeatedly. This can be deduced from the fact that the "start exec" signal frequently goes high as shown in figure 5.8. The overhead associated with this has a significant impact on performance, as the "output res" and "shift en" signals are active for a considerable amount of time.

5.4 Performance Scalability

The conversion of kernel loops to hardware, increased performance by up to a factor of 54.9. Although this is a significant increase, since the average hardware utilization was only 6.3%, there is considerable room for further improvement. Furthermore, the current generation of FPGAs [114, 115] on the market, are up to four times larger than the APEX 20K1500, used for this evaluation.

5.4.1 Bandwidth

Many of the algorithms tested here are highly bandwidth limited and this is partly due to the non-pipelined nature of the memory interface. As a result, load operations

take two clock cycles and store operations take three. By pipelining the memory interface, a potential and additional 2-3x increase in performance might be obtained.

The current hardware conversion system extracts parallelism in two orthogonal directions and, in the majority of test cases, the resulting parallelism is enough to saturate the available bandwidth. The parallel nature of many algorithms when combined with the additional optimizations outlined in section 7.1.3, enables the utilization of all available hardware resources. However, considerable bandwidth may be required. A good example of this, is the random number generation test which is capable of producing 64 MBytes/s of data at a clock speed of 16 MHz whilst occupying a mere 1.2% of the FPGA. If the hardware were to be scaled to use all the available logic cells in the FPGA, it would generate over 5.3 GBytes/s of bandwidth. Memory systems that can handle this level of bandwidth are readily available (e.g. dual channel DDR-400 = 6.4 GBytes/s), however, the limiting factor soon becomes the bandwidth available out of the FPGA itself. The FPGA used in the MIPS test platform has 488 user IO pins. At 16 MHz, this results in a maximum bandwidth of 967 MBytes/s, which is 5.5x lower than the data rate that the system requires. This scales to a required bandwidth of 83.3 GBytes/s compared to an available bandwidth of 15.3 GBytes/s when the clock speed is raised to its maximum of 250 MHz (the fastest that the IO pins will allow). Although it is possible to create memory systems with this level of performance, this is impractical in a mainstream computing environment.

5.4.2 Parallelism

Increasing the amount of parallelism in code will produce dramatically different results, depending on whether the algorithm is to be performed in software or in hardware. This is because processors and FPGAs are based on completely different computing paradigms:-

Processors Even the most modern superscalar and EPIC CPUs only operate on small sections of code at a time, executing instructions pseudo-sequentially. In general, since a single processor can only exploit ILP, the number of instructions that can be simultaneously executed is limited by both the data dependencies and the maximum issue rate of the instruction pipeline. As a result, increasing the number of instructions to be executed leads directly to an increase in execution time. Equation 5.2 demonstrates this by the fact that the number of instructions executed, is directly proportional to the execution time.

$$\begin{aligned} \text{SW exec time} &\approx \frac{\text{Num loop iterations} \times \text{Num instructions in loop}}{\text{Average issue rate} \times \text{Clock speed}} \\ &\approx \frac{\text{Number of instructions executed}}{\text{Average issue rate} \times \text{Clock speed}} \end{aligned} \tag{5.2}$$

FPGAs Once converted to hardware each instruction is represented by its own block. The clock rate is limited by the speed of the slowest block and not by the number of blocks. Because hardware is inherently parallel, several orthogonal types of parallelism can be exploited (e.g. ILP, loop iteration and inter-loop parallelism). Therefore, the number of operations that can be performed in any one clock cycle, is limited only by the data dependencies and not by the issue rate of an instruction pipeline. The parallelism is demonstrated by the fact that the number of operations that the pipeline performs has no effect on performance and this is, therefore, not a factor in equation 5.3, which calculates the execution time. Consequently, as the number of instructions increases the hardware utilization also increases, whilst the clock speed remains roughly

constant.

$$\text{HW exec time} \approx \frac{(C \times I) + L}{\text{Clock speed}}$$
$$C = \text{Cycles between iteration starts}$$
$$L = \text{Pipeline length} \tag{5.3}$$
$$I = \text{Number of loop iterations per run}$$

Several common optimization techniques, like loop unrolling and loop pipelining, are designed to increase the ILP in the code. The effects of increasing the level of parallelism, for both hardware and software, are shown in figure 5.9. The speed of the software only case increases until the maximum issue rate of the pipeline is reached, at which point the performance remains constant. If the parallelism is increased further by unrolling, the performance will eventually start to decrease as instruction cache misses and register stacking reduce the efficiency of the processor. Because the hardware is not limited to only exploiting ILP, its performance will increase more rapidly than the performance of the software. At some point, the performance of the hardware will level off as the bandwidth limit is reached. Any further increase in parallelism due to unrolling, will only result in additional hardware utilization without any further increase in performance, and should therefore be avoided. Ultimately, the hardware conversion will fail, as the resources required will become greater than those available.

5.4.3 Algorithm Complexity

An initial analysis of the results, suggests that reconfigurable computing platforms will always be bandwidth limited. However, by increasing the size and/or the complexity of the section of code that is to be converted into hardware, it is possible to increase further the performance of bandwidth limited algorithms.

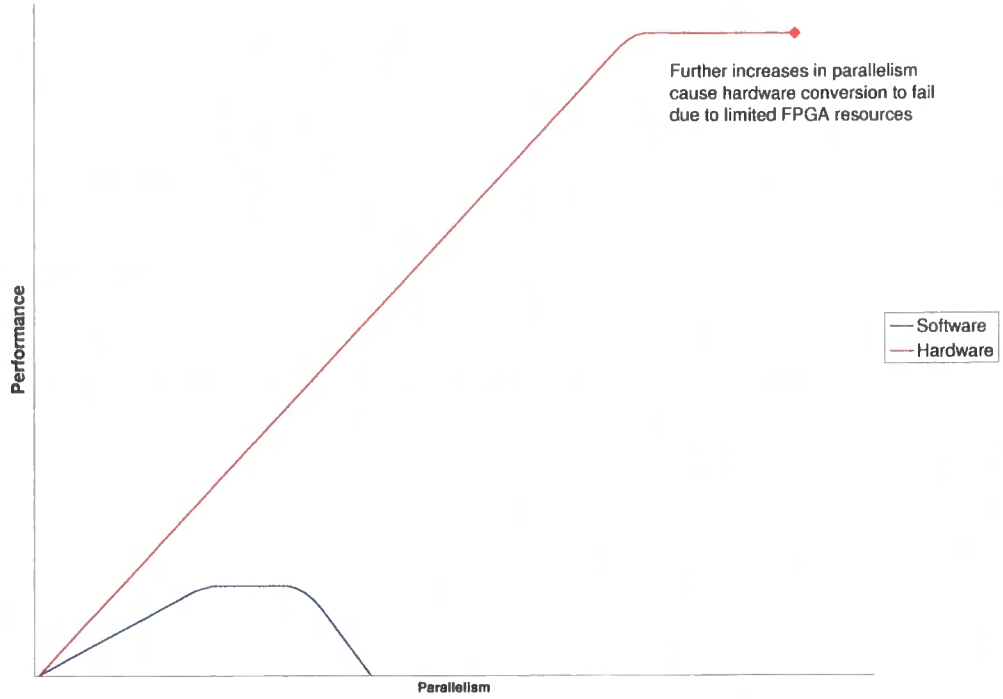


Figure 5.9: Generalized effects of parallelism on performance (Software vs Hardware)

5.4.3.1 Increased section size

Generally, the larger a section of code is (that is to be converted into hardware), the greater the parallelism and therefore the higher the performance (see section 5.4.2), even if the system is already bandwidth limited. If the additional code being added to the RC area operates on a data set that is already processed or produced by the RC area, then the bandwidth requirements may not change significantly. In some cases, the off chip bandwidth requirements will actually be reduced. One example of this is picture processing and compression, a simple example of which is shown in figure 5.10. If the RC area contains a histogram correction algorithm, adding run length encoding (RLE) compression will reduce the off chip bandwidth. Similarly, adding a low pass filter to the RC area will not change the bandwidth requirements but would increase performance. The process of adding additional sections of code to the RC area can be automated using the procedure described in section 7.1.1.

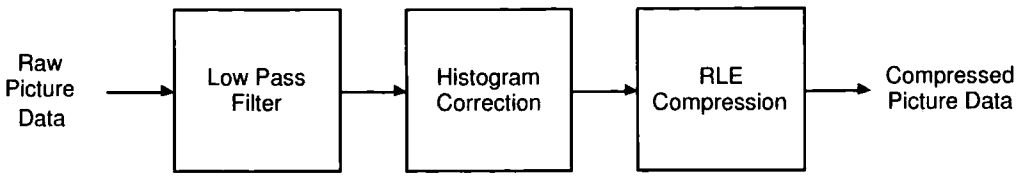


Figure 5.10: Picture processing and compression

5.4.3.2 Increased complexity

A cursory analysis suggests, that there is little or no benefit to be obtained from reconfigurable computing on a real time system such as MPEG decode and playback, since once the real time constrains have been met, there is little advantage in increasing performance. However, in many environments the complexity of an algorithm or data processing system is limited by the available computational power. A good example of this is 3D games, where, with the advent of ever faster CPUs and 3D graphics cards (that offload the rendering of the display to dedicated hardware), the amount of processing power available has increased dramatically. However, the amount of idle time has remained roughly constant due to the ever increasing complexity of the physical models on which the games are based. The same trend can be seen in HPC environments, where the time required to perform intensive tasks, like simulation, has also remained approximately constant. This is due to the fact that the complexity and therefore the accuracy of these systems has increased inline with processing power.

The bandwidth constraints outlined in section 5.4.1, can severely limit the performance improvement obtained when reconfigurable computing is applied to existing algorithms. However, in many cases, bandwidth does not limit the size of the algorithm. As a result, in addition to producing significant increases in performance, reconfigurable computing could yield dramatic increases in the complexity and therefore the accuracy of many computing tasks/models.

5.5 Platform Evaluation

Section 1.3 outlined the features required to produce a viable reconfigurable computing platform, in general these can be summarized by the following requirement:-

- Abstraction
- Automatic conversion
- Low conversion time
- Large performance increase

5.5.1 Abstraction

Because the conversion process used (see chapter 4) is designed to be performed at runtime instead of compile time, the original software executable contains no information specific to the target RC area. In addition to providing complete abstraction this also has the benefit that existing legacy software will also benefit from hardware acceleration.

5.5.2 Automatic conversion

With the expecting of the user setting for pointer aliasing (see section 4.6.1.1) the tool flow is completely automatic and autonomous. Since section 4.6.1.1 outlines a solution that eliminate the need for any user intervention it is expected that the requirement for an automatic conversion process can be fulfilled by the tool flow presented in this thesis.

5.5.3 Low conversion time

The total time required to perform the conversion process is largely dominated by the time required to perform the FPGA place and route, and is therefore outside the control of the tools presented here. However research has already been conducted on possible methods to significantly reduce this overhead [44]. A considerable amount of effort has gone into ensuring the conversion tools do not require large amounts of CPU time (currently a few seconds, even for the most complex conversions), so that, in the future the overall conversion time should be minimal.

5.5.4 Large performance increase

The use of reconfigurable computing offers the opportunity to greatly accelerate the performance of a system. With speed up factors of 55 times achieved, the new tools flow presented in this thesis is no exception. In the future much greater performance improvement should be achievable once the bandwidth restrictions outlined in section 5.4.1 have been resolved.

5.6 Summary

In most cases the performance enhancement is constrained by the memory bandwidth available. Although this is a general problem with all computing paradigms, it is especially true of reconfigurable computing platforms because of the higher computational throughput, and therefore the requirement for faster data transmission. Unfortunately, the memory interface/controller that was present on the host MIPS CPU used, was only designed to support the minimal bandwidth requirements of a single issue CPU. As a result, it requires 2 cycles to perform a load and 3 cycles to perform a store operation and this is further compounded by the fact that the

memory interface is not pipelined. Although this severely limits the performance produced, the conversion to hardware still achieved speedups in the range of 1.2x to 54.9x.

In the software domain, the execution time is governed by the number of instructions that are executed. However, in the hardware domain, this translates into the amount of hardware used in implementing the loop, and as such does not directly affect the execution time. Instead the hardware execution time is determined by both the amount of parallelism that can be exploited and also by the available bandwidth. The average FPGA unitization was only 6.3% (3200 logic cells) showing that there is a considerable amount of unused hardware resource which could be used to increase the algorithm complexity/parallelism. This would directly lead to additional increases in the speedup produced.

During the execution of some of the test algorithms, it was noted that the hardware pipeline was triggered repeatedly, often with large amounts of idle time between executions. This is due to the fact that the hardware conversion can only process the innermost loops of an algorithm. Substantial, further improvements could be obtained by optimizing the hardware triggering method and by changing the hardware generation process, so as to be able to handle nested loops

Chapter 6

Cray XD1 Platform

In order to evaluate the performance potential of reconfigurable computing when utilized in a high performance computing environment, the hardware conversion tools outlined in chapter 4 were used in conjunction with the Cray XD1 supercomputer.

6.1 Cray XD1 Overview

The Cray XD1 [67] is the first of a new generation of computers that contain FPGAs as an integral part of the system. In these, the FPGAs are tightly coupled to the processor, have high bandwidth and low latency access to memory. The XD1 is a modular and scalable system, consisting of one or more 3 VU rack mount chassis. Each chassis contains 6 compute blades, with each blade consisting of an FPGA and two Opteron CPUs running Linux (as shown in figure 6.1). Although XD1s, based on the new generation of dual core Opteron processors and Virtex4 FPGAs are now available, the system used during the course of this research contained single core CPUs clocked at 2.2 GHz together with Virtex II Pro 50 FPGAs.

In addition to providing a high bandwidth link between the FPGA and the rest

of the system, the chipset also provides a set of software accessible registers to control the supporting hardware for the FPGA. Included in this, is a programming interface which allows the user to reconfigure the FPGA. Since the user defined logic in the FPGA is in a different clock domain from the interface to the CPU and main memory, the user logic is able to run at any clock speed between 63 and 199 MHz, in 1 MHz increments. This can be changed whilst the system is running and is controlled by the chipset.

To interface the FPGA to the rest of the system, Cray provides several interface cores. These are combined with the user defined logic to form the FPGA configuration, as shown in figure 6.2. Together, these cores occupy approximately 8% of the Virtex II Pro 50 FPGA leaving a substantial number of logic cells and other hardware resources available for the implementation of user logic.

Clock & reset core This block generates all the clock signals required by the interface hardware and user logic, derived from the reference clock which is generated by the chipset. The block also provides additional reset signal generation and distribution.

RT core The data interface from the chipset is a simplified version of the HyperTransport protocol [116]. This interface is very complex and runs at a high clock rate. The RT core translates this into a wide, low clock rate interface that simplifies the design of the user logic section in the FPGA.

QDR interface There are four instances of the QDR interface - one for each external memory. Like the RT core, they provide a simple, easy to use interface to these otherwise complex devices.

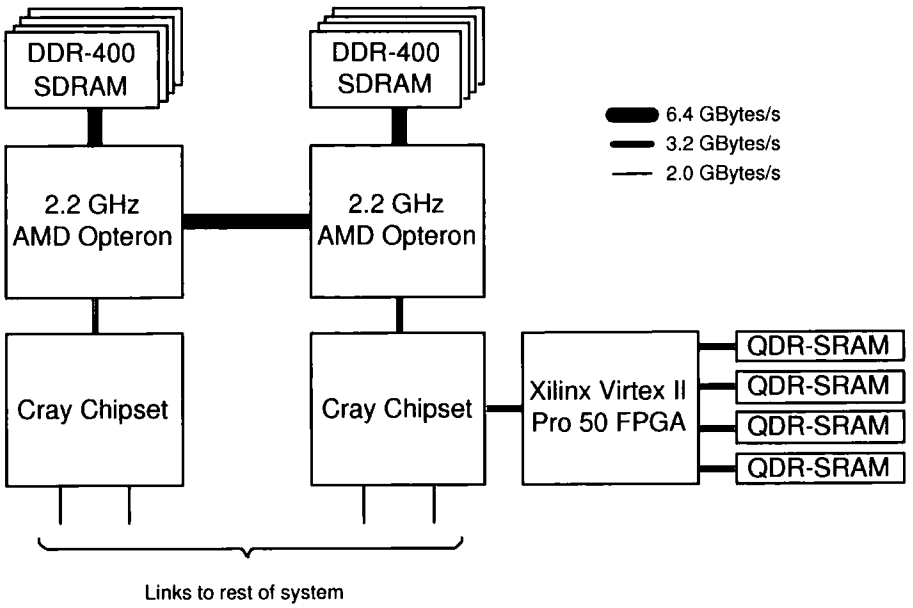


Figure 6.1: Cray XD1 blade architecture

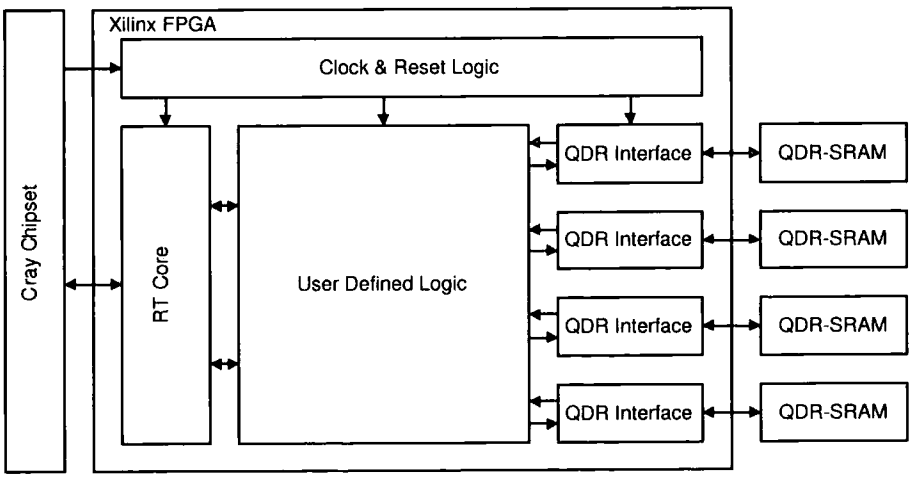


Figure 6.2: Cray FPGA interface cores

6.2 Platform Details

6.2.1 Execution

The user defined logic section of the FPGA is used to implement a data flow pipeline similar to the one described in section 3.3. Unlike the MIPS test platform described in chapter 5, the FPGA is not integrated into the instruction pipeline of the processor. As a result, execution in the FPGA is triggered by accessing memory mapped control registers within the FPGA and not by executing specific RC trigger instructions, as is the case with the MIPS test platform.

To measure the performance increase of both the software only, and hardware accelerated test cases, the clock cycle counter present in the Opteron processor was used [117]. This was the most accurate way available to measure the execution time.

The hardware in the XD1 contains a link between the FPGA and the interrupt controller. This would normally allow the CPU to trigger execution in the FPGA and then continue conventional software execution. When the FPGA finishes its task or requires software intervention, it should be able to trigger an interrupt that would cause the CPU to jump to the FPGA control code. However, the software to support the hardware interrupt connection is unfortunately not yet available, although it is scheduled for a future release of the Cray FPGA software application programming interface (API). This, together with other platform limitations, reduces the functionality and performance of the hardware conversion tools described in chapter 4. In particular, the lack of interrupt capability results in the CPU having to continuously poll the FPGA registers to determine when the execution has finished.

6.2.2 Tool flow

Like many other modern computing platforms, the XD1 implements a virtual memory system. Since the FPGA is connected directly to the main system bus instead of to the MMU, it exists in the physical address space, whereas the software is executed in one or more virtual address spaces. This makes it extremely difficult to produce an x86 front end for the hardware conversion tools. As a result, the C source code is compiled to a MIPS executable and is processed using the existing MIPS front end for the hardware generator (as shown in figure 6.3). As outlined in section 7.2.2, once the next release of the Cray support software is made available, this problem can be resolved and an x86 front end produced. Since a MIPS executable is the starting point for the hardware conversion process, it must be accompanied by a MIPS execution profile. This is because the location of the branches will be very different in the x86 executable, due to the differences in the compiler and the complex instruction set computer (CISC) nature of this processor. Therefore, the execution profile is generated by the behavioural simulation of the MIPS platform, described in section 5.1. The original source code is compiled, without modification, using the x86 GCC compiler; the execution of this code provides a reference which is then used to determine the speedup factor produced by the hardware acceleration.

The use of the MIPS front end instead of an x86 front end will have an effect on the results, however it is expected that any differences will be minimal, and would be dwarfed by other factors, such as available bandwidth and the amount of parallelism present in the code. This is due to the fact that both of the C compilers used (MIPS and x86), are variants of GCC, and therefore the same types of optimisations will be performed in both cases.

Due to the complex and time consuming nature of handwriting the RC trigger instructions, the conversion process is limited to accelerating only a single software loop.

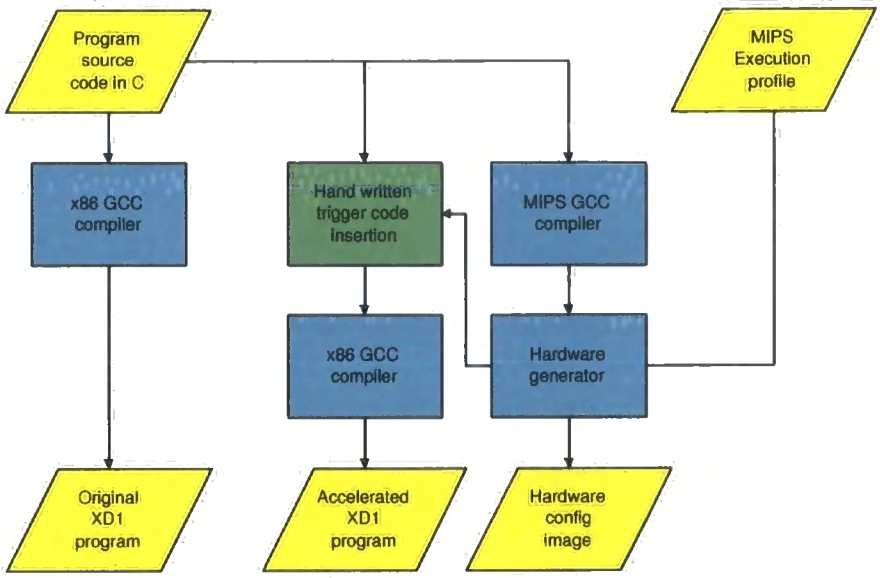


Figure 6.3: Hardware conversion tool flow for Cray XD1

6.2.3 Memory Access

Due to the issues associated with virtual memory that were outlined in section 6.2.2, all the DMA operations in the data flow pipeline access the QDR-SRAM memories, instead of the main SDRAM memory that is connected to the processor. Even if the virtual memory issues were to be resolved so that DMA operations could access the main SDRAM, the performance of the FPGA would still be severely limited owing to the high levels of bandwidth consumed by the CPU polling the FPGA, described in section 6.2.1. A connection between the RT core and the QDR-SRAM cross bar switches is present, which allows the host CPU to read and write test data to these memories.

Each Cray QDR-SRAM interface core, provides independent 72 bit read and write ports capable of running at up to 200 MHz. Consequently, the combined bandwidth to each SRAM is 3.2 GBytes/s. All four SRAMs are interleaved to form one 16 MByte address space. This is accomplished by the two cross bars that are used to interface the memory operations in the data flow pipeline to the SRAMs, as shown in figure 6.4. The cross bars analyse the addresses of the memory transactions

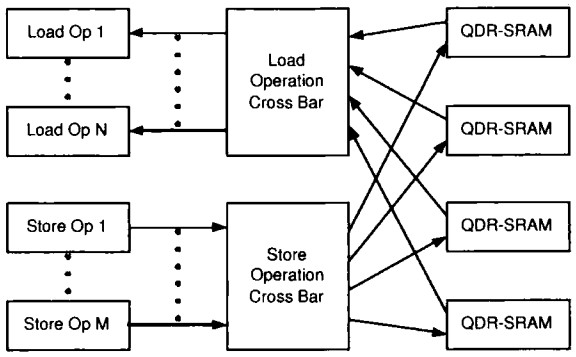


Figure 6.4: Cross bar architecture for QDR memory interface

and allocate them to the corresponding memories. Each cross bar can issue up to four operations per cycle, provided that the addresses of pending operations do not reside in the same physical memory device. As a result, the maximum theoretical bandwidth is 12.8 GBytes/s, but in practice this is reduced to 6.4 GBytes/s, as the maximum width of a value in the pipeline is 32 bits compared to the 64 bit width of the memories. However, this reduced bandwidth exactly matches the bandwidth to the main DDR-SDRAM memory, available to the Opteron processors, which confers more validity to the performance comparison between the software only and the hardware accelerated test cases. It's worth noting that the latency of the SDRAM is dependent on many factors, which include previous access patterns and which rows, columns, and pages are active. In contrast, the latency of the QDR-SRAM, including the controllers and cross bars, is always 10 clock cycles.

Although in theory, the minimum operating frequency for the RT interface and other support cores is 63 MHz, due to a bug in the implementation of the QDR core provided by Cray, clock speeds lower than 130 MHz resulted in memory access errors. Algorithms that contain DMA operations are, therefore, limited to a frequency range of 130 - 199 MHz instead of 63 - 199 MHz, as both the QDR memories and the data flow pipeline are in the same clock domain.

To minimize the performance impact due to the latency of the memory interface, all the DMA operations in the data flow pipeline are fully pipelined. For example,

a load operation spans ten pipeline stages and can have active data in every stage. As the latency of the DMA operations in the pipeline matches the latency of the memories, the pipeline will only stall if the addresses of multiple operations issued on the same cycle, reside on the same SRAM device. In the ideal case, the pipeline is able to issue four load and four store operations on every clock cycle without stalling. This is a considerable improvement over the MIPS test platform where every load operation stalled for 2 clock cycles and each store operation stalled for 3 cycles.

6.3 Performance Evaluation

All the test algorithms are based on the same source code and run under the same test conditions as those used on the MIPS platform (See section 5.3 and appendix A). Each test was executed both in software and also with the hardware acceleration enabled; the number of clock cycles required was recorded and then used to calculate the improvement in performance. For algorithms that contain memory operations, the software case accessed a 16 MByte buffer allocated in the main DDR-SDRAM of the system, while the hardware accelerated version of the algorithm accessed the local QDR-SRAMs. The effect of the different memory latencies between the FPGA and CPU is minimized, due to the intelligent cache prefetching performed by the CPU, together with the streaming nature of many of the test algorithms. This combined with the identical memory bandwidths (as described in section 6.2.3), make the results of the software only and hardware accelerated test cases approximately comparable. Both memories were initialized with the same test data, and a bitwise comparison was performed on the results, once the test had completed. In every case, the contents exactly matched, indicating that the hardware present in the FPGA functioned correctly.

For each test case, the average number of parallel operations executed in the pipeline was recorded. This measure of parallelism can be used to give a rough indication of the performance increase that is to be expected. It is worth noting, that the number of pipeline operations will be different from the number of software instructions, due to the nature of the optimization steps performed during the hardware conversion. Because the frequency of the FPGA is set at the maximum for each algorithm, the TSF value is different for each test case. The TSF values, hardware utilizations, and performance improvements for each algorithm are shown in table 6.1.

Due to the bugs in the Cray XD1 outlined in section 6.2 and summarised in section 6.5.1 the following restrictions were placed on the test algorithms:-

- Only a single software loop can be converted to hardware.
- Complex algorithms that require repeated triggering of the hardware can not be evaluated.
- The hardware resulting from the conversion process must be able to run at 130MHz or greater.

The predicted results with these limitations resolved are shown in table 6.2.

6.3.1 PRBS Generator

The PRBS test generates 2 MBytes of random numbers using an LFSR. The predicted number of operations that can be performed per clock cycle is 161, this is due to the high levels of ILP present in the algorithm. The simple bitwise nature of this algorithm, results in a high operating frequency of 196 MHz which leads to a relatively low TSF value of 11.2. As the amount of parallelism is considerably higher than the TSF value, a significant increase in performance was expected when

Algorithm	Unroll factor	Hardware usage		Parallelism	FPGA frequency	TSF	SW Exec time (x1,000 cycles)	HW Exec time (x1,000 cycles)	Speedup factor
		Logic cells	% of FPGA						
PRBS generator	32	3981	8.4	161.0	196	11.2	106535	5872	18.14
Half Brightness	1	5749	12.2	56.0	161	13.7	18737	7150	2.62
	2	7186	15.2	110.0	146	15.1	17371	3945	4.40
Low pass filter	1	6223	13.2	4.2	145	15.2	13200	95185	0.14
Normalization	1	4637	9.8	17.0	149	14.8	5268	7724	0.68
Copy	1	4479	9.5	7.0	159	13.8	2113	3621	0.58
	2	5044	10.7	9.0	162	13.6	2158	1779	1.21
Series sum	1	3845	8.1	3.0	190	11.6	201	1159	0.17

Table 6.1: Test algorithms used on the Cray XD1

Algorithm	Unroll factor	Hardware usage		Parallelism	FPGA frequency	TSF	SW Exec time (x1,000 cycles)	HW Exec time (x1,000 cycles)	Speedup factor
		Logic cells	% of FPGA						
PRBS generator	128	4389	9.3	648	196	11.2	106535	1468	72.56
Half Brightness	4	10060	21.3	218	116	18.9	16005	2486	6.44
Low pass filter	4	13357	28.3	16.8	116	18.9	13200	29830	0.44
Low pass filter with prefetch	4	13357	28.3	54.6	116	18.9	13200	9179	1.44
Normalization	4	7013	14.8	8.6	116	18.9	5268	2485	2.12
Copy	4	6174	13.1	23	126	17.3	2202	1135	1.94

Table 6.2: Predicted performance improvement after resolving current Cray XD1 limitations

the algorithm was converted into hardware. This was found to be the case as the hardware accelerated version of the algorithm was in excess of 18x faster than the pure software case.

As the software case uses a mere 41 MBytes/s out of the 6.4GBytes/s of available bandwidth, it is clear that the CPU is instruction limited, rather than bandwidth limited. In comparison, the hardware accelerated case only issues one memory write per cycle out of a possible four. This combined with the low hardware utilization of 8.4% and the parallel nature of the algorithm, suggests that the hardware could be further improved to give a 72x overall increase in performance, when compared to the software only case.

6.3.2 Half Brightness

The half brightness test algorithm produces a reduction in brightness that is correctly colour balanced; this is accomplished by performing the operation in the YUV colour space instead of in the RGB colour space. The complex operations present in this algorithm, lead to a reduced clock speed of 161 MHz and therefore to a TSF value of 13.7; a value that is higher than that in the PRBS generator test described above. This, combined with the subsequent lower amount of parallelism, results in a 2.6x increase in performance when compared to the software only case.

As the hardware implementation of this algorithm merely issues one load and one store operation per clock cycle, it is only utilizing a quarter of the available bandwidth. The body of the loop was unrolled by a factor of two to increase the amount of parallelism in the section of code. This, consequently, improved the performance by using more of the available bandwidth. At first sight, it was expected that this would double the performance of the system, but in fact the performance only increased by an additional 1.7x, equating to a 4.4x increase over the pure software

case. This is due to the increased complexity of the hardware reducing the maximum clock speed from 161 MHz to 146 MHz. In theory, the unroll factor could be increased to four, which would result in an overall performance increase of approximately 6.4x. However, this configuration could not be tested as the clock speed would fall below 130 MHz and the bug described in section 6.2.3 would then produce memory corruption.

6.3.3 Low Pass Filter

Converting the audio low pass filter algorithm to hardware, produced a 7x reduction in performance due to the combination of a low clock speed (145 MHz) and the relatively low number (4.2) of operations per clock cycle. Because of the data dependencies present between iterations of the loop, the next iteration cannot be started until the majority of the calculations for the previous iteration have been completed. This, combined with the high latency of the load operation, leads to eight clock cycles of idle time per iteration, as is shown by the lack of any data processing operations between stages 1 and 8 in figure 6.5. A potential way to increase the efficiency, is to unroll the loop, so that the algorithm would occupy more than the 13% of the FPGA utilized in this test. Again, it was impossible to evaluate the effects of this optimization due to the clock speed bug in the QDR interface core. Alternatively, it may be possible to further increase the performance by changing the way, in which, the hardware is generated so that the data is pre-fetched from memory, thus greatly reducing the effects of the high memory latency.

6.3.4 Normalization

This algorithm is similar to that of the audio normalization test, performed on the MIPS platform (see section 5.3.4 and appendix A.5). However because the XD1

Stage 0	lw d2 0 sd5 add d7 sd2 bd3 dFwd sd0 sd10 add sd9 d0 sd12 sltu sd11 cd0	sd0 sll d6 1 sd6 add sd3 sd4 bd9 dFwd d5 d3 add sd10 d5 sIt bne sd12 0	sd1 sll d1 1 d0 add sd5 sd6 sd7 sll d10 1 bd10 dFwd d0 bd2 add d2 4	sd2 add sd0 d6 bd8 dFwd d7 sd8 add sd7 d10 bd5 dFwd d4 d1 dFwd d9	sd3 add d8 d3 bd6 dFwd d1 sd9 sll d0 1 bd4 dFwd sd7 d4 dFwd d2	sd4 add sd1 d1 bd7 dFwd d3 d2 add d4 sd8 bd11 sd11 add d11 1
Stage 1	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 2	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 3	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 4	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 5	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 6	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 7	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 8	Pipe: lw	d0 dFwd d0	d1 dFwd d1	d2 dFwd d2	d3 dFwd d3	d4 dFwd d4
Stage 9	Pipe: sd0 lw bd1 dFwd sd2 d6 dFwd d4	sd1 sll sd0 16 bd0 dFwd sd3	sd2 sra sd1 16 d0 dFwd d0	d1 sll sd2 2 d3 dFwd d1	sd3 sra sd0 16 d4 dFwd d2	d2 sll sd3 2 d5 dFwd d3
Stage 10	sd0 add d0 d1 sd5 sra sd4 16	sd1 add d2 d3 sd6 add sd1 d4	sg0 bgez sd0 d1 and sd5 65535	sd2 add sd0 15 d0 add sd6 d5	sd4 sll sd3 12 d2 dFwd d6	sd3 mux sg0 sd2 sd0
Stage 11	sg0 bgez d0 d0 or sd4 d1	sd0 add d0 15 d1 dFwd d2	sd1 mux sg0 sd0 d0	sd2 sll sd1 12	sd3 sra sd2 16	sd4 sll sd3 16
Stage 12	sw d0 d1 0					

Figure 6.5: Operations on the hardware data flow pipeline for the low pass filter algorithm

platform is limited to implementing one loop in hardware, only the value scaling section of the algorithm is tested.

As the normalization algorithm has very few inter-iteration dependencies, a new iteration of the loop may be started on every clock cycle. This not only eliminates the effects of the memory latency but also increases the parallelism to 17 operations per clock cycle. This conversion to hardware actually produced a 32% reduction in performance. This is because the parallelism that was extracted was not sufficient to overcome the fact that the CPU used can execute multiple instructions per cycle and has a clock speed approximately 15x higher than the FPGA.

The above test algorithm issues two memory operations per cycle out of a possible eight. Given that hardware utilization is a mere 9.8%, the main loop is a prime candidate for unrolling. Once again, the effects of this could not be evaluated due to the bug in the QDR interface core. However, it is expected that the hardware would be 2.1x faster than the pure software case, if the loop were to be unrolled by a factor of four. Because this case would use the maximum available bandwidth, further increases in performance could only be achieved by either optimizing the hardware to increase the clock speed, or by increasing the complexity of the algorithm (see section 5.4.3).

6.3.5 Copy

The copy algorithm is extremely bandwidth intensive and only contains a small number of instructions. When the original version of the source code is converted into hardware, only two memory operations per cycle out of a possible eight are issued. This confers a significant performance advantage to the CPU. However, in practice, the hardware solution is only 41% slower than the software test case. When the loop is unrolled by a factor of two, the performance of the software

remains constant, whereas the performance of the hardware doubles. This results in a speedup of 20% when compared to the software case. Although it couldn't be tested due to the bug in the QDR interface, the performance of the hardware is expected to increase to a factor of 1.9x if the loop were to be unrolled by a factor of four.

The most likely cause of the poor performance of the CPU, is the "read before write" architecture used in the caches. This architecture, reduces the complexity of the cache by using only one data valid bit per cache line. However, this can result in up to a 100% increase in the bandwidth required for some classes of algorithms, because a cache line fill must be performed before any cache writes are allowed to take place.

6.3.6 Series Sum

The series sum test consists of a very tight loop with an extremely low level of ILP, consequently the hardware accelerated test case only performs 3 operations per clock cycle. The simple nature of this algorithm, leads to the relatively high clock speed of 190 MHz (TSF = 11.6). Overall, the algorithm produced a 5.9x reduction in performance when it was converted into hardware. On account of the high numbers of data dependencies together with the tight nature of the loop, it is unlikely that unrolling the loop would produce a significant increase in performance.

6.4 MIPS RC Platform Comparison

Although the MIPS and Cray XD1 platforms share the same hardware conversion tools, there are significant differences in the hardware architectures of these two systems, which are detailed below. The most obvious difference, is the way in which

their FPGAs are connected to their processors. In the MIPS platform, the FPGA is directly connected to the instruction pipeline of the CPU, which enables the RC area to be triggered with a minimal overhead. In the XD1 platform, the FPGA is connected to the system bus via a Cray proprietary chipset, as such, execution in the RC area is triggered by accessing a series of memory mapped registers. Although the Cray approach enables reconfigurable computing to be added to an existing system, it considerably increases the number of clock cycles required to trigger execution in the RC area. Consequently, a much higher overhead is experienced by applications that repeatedly trigger the RC area. In the MIPS platform, because the FPGA is directly connected to the processor, it utilizes the same load/store interface as the CPU. This interface severely constrains the performance of the data flow pipeline, and this is due to the memory interface being designed to support a single issue CPU core and not a highly parallel RC area. At peak performance the MIPS FPGA can issue one DMA operation every 2-3 clock cycles. By comparison, the memory subsystem present in the XD1 platform, is fully pipelined and parallelized, and is therefore capable of processing up to 8 DMA operations per clock cycle. To fully exploit the available bandwidth in the XD1, the data flow pipeline instantiated within the FPGA must be highly parallel in nature.

The FPGA used to implement the RC area in the MIPS platform was an Altera Apex 20K1500. This is a relatively old device compared to the Virtex II Pro 50 used in the Cray XD1. Consequently, the Apex device does not contain any hardware multiplier blocks, so multiple operations have to be implemented using standard logic cells. This reduces the clock speed of the system and leads to significantly higher hardware resource utilization, than that found in the Cray XD1. The Virtex device used in the XD1 is also capable of running at much higher clock frequencies than the Apex device, further increasing the performance of the Cray platform. Since the FPGA in the XD1 exists in a separate clock domain from the rest of the system, its clock speed can be set to the optimal value for the specific hardware

that is to be implemented. However, in the MIPS platform the FPGA runs from the same, fixed frequency clock source as the CPU core.

6.5 Summary

6.5.1 Cray XD1 Platform Limitations

Due to a series of bugs, unimplemented features, and architectural problems associated with the XD1 host platform, the scope of the performance evaluation was limited in the following ways:-

Manual integration The FPGA is connected to the system after the MMU. As a result, the FPGA exists in the physical address space, whereas the software exists in a virtual address space. This complicates the integration of the automatically generated hardware with the software. As a result, this integration is currently done manually, limiting the system to converting a single loop to hardware. The need for manual integration also prevents the XD1 system running in a runtime self adaptive mode. Like the MIPS platform (see chapter 5) the conversion process in performed pre-runtime using only the program binary and a execution profile. No access to the original source code is required.

No concurrent execution The current release of the Cray software does not support the use of the hardware interrupt line between the FPGA and the CPU. Consequently, the processor must continually poll the FPGA to determine if the execution has finished, or if there are any outstanding requests that need servicing. This prevents the CPU working concurrently with the FPGA.

No DMA to main memory DMA operations must be performed to the QDR-SRAM memories that are connected to the FPGA instead of to the main

SDRAM memory, as the majority of the available bandwidth is consumed by the CPU polling the FPGA.

Restricted clock speed range Due to a bug in the QDR-SRAM interface core provided by Cray, clock speeds less than 130 MHz, resulted in data corruption. Therefore, some algorithms could not be tested with higher loop unroll factors, despite the abundance of hardware resources.

Although these factors do influence the results, in many cases the effect is to artificially reduce the performance produced by the hardware acceleration. Therefore these limitations do not change the conclusion, that using automatic hardware conversion tools on HPC platforms like the XD1, can produce significant increases in performance.

6.5.2 Clock Speeds

During the course of the evaluation, it was noted that algorithms that contained multiply operations had lower clock speeds. To increase the clock speed, multiply operations were pipelined. This was achieved by placing additional sets of registers after each multiply operation and ensuring that the register re-timing feature was enabled. Although this was in accordance with the Xilinx guidelines, it did not result in a significant increase in clock speed, despite the fact that the timing reports indicated that multiply operations were still the limiting factor. This could be resolved by hand coding the multiply operations to explicitly specify the placement of the additional pipeline registers. In addition, since a new range of FPGAs, containing dedicated cascade logic between the hardware multiplier blocks has become available (e.g. Virtex4 [114]), it is now possible to create 32 bit wide multipliers that are capable of running at up to 500 MHz.

It was noted that algorithms containing large numbers of DMA operations also had

lower clock speeds. This is because, as the size of the crossbar switches increases, so does the MUXs that they contain, causing their propagation delays to rise. In order to maintain high clock rates for algorithms with large numbers of DMA operations, it would be possible to modify the data flow pipeline generation system to also generate the crossbar switches, so that additional registers are automatically added as required.

6.5.3 Performance Improvements

The Opteron processor present in the Cray XD1, can issue multiple instructions per cycle, and has a clock speed of 2.2 GHz. Since the FPGA that is present has a maximum clock speed of 200 MHz, a high level of parallelism is required for this FPGA to produce a significant increase in performance. In general, if the conversion to hardware produces more than 30 parallel operations, a significant increase in performance will result. Some algorithms, such as the series sum test, are not well suited to hardware conversion due to the low levels of parallelism present. However, since the number of parallel operations is calculated at an early stage in the conversion process, this calculation can be used to both identify and screen out non-ideal sections of code from the hardware generation process. Although the current hardware conversion system produced performance improvements of over 18x, it is clear that there is scope for further enhancement.

Chapter 7

Optimisations Of The Reconfigurable Computing System

The results from both the MIPS embedded platform in chapter 5 and the XD1 HPC platform described in chapter 6, demonstrate that a considerable increase in performance can be achieved by converting computationally intensive sections of software into hardware. However, it is also clear that further improvements can be made by changing both the conversion software and the hardware platform.

7.1 Hardware Conversion Tools

7.1.1 Loop Extraction

As the hardware conversion tools are only capable of processing the innermost loops, a large proportion of the execution time may still be spent outside of these loops, resulting in some computationally intensive code not being converted to hardware. This is one of the major causes of the poor performance of the hardware in the

```
for ( y = 0; y < 600; y++ )
{
    // Inner loop A
    for ( x = 0; x < 800; x++ )
    {
        if ( pictureA[y][x] > max ) max = pictureB[y][x];
    }
    // Inner loop B
    for ( x = 0; x < 800; x++ )
    {
        if ( pictureB[y][x] < min ) min = pictureB[y][x];
    }
    // calculate sum of absolute differences
    diff = pictureA[y][0] - pictureB[y][0];
    SAD += (diff >= 0) ? diff : -diff;
}
```

Listing 7.1: Example code with multiple nested loops

quick sort algorithm on the MIPS platform (see section 5.3.10). Another side effect, is the high level of overheads for algorithms where the inner loop is triggered repeatedly, one example of this being the FFT test (see section 5.3.2). It is possible, however, to alter the hardware conversion system so that these outer loops might also be converted to hardware. In addition to removing the overhead associated with triggering the inner loop multiple times, this would also enable the conversion system to exploit parallelism in an additional, orthogonal direction. This can be seen in listing 7.1. When using conventional software execution, the two inner loops and the calculation of the sum of absolute differences is done sequentially. However, as there are no dependencies between them, they may be executed in parallel in the hardware domain. This can produce a substantial increase in the amount of parallelism, and consequently will fully exploit the performance increase enabled by hardware acceleration.

The current hardware generation tools are not capable of converting loops that contain function calls. The loop extraction algorithm could easily be modified to automatically inline suitable functions. This would increase the performance of the

system by increasing the number of loops that could be converted to hardware.

7.1.2 Floating Point Operations

The use of floating point operations is currently not supported. The implementation of this feature is a simple matter of adding the VHDL primitives, for the various operations, to the operation library in the hardware generation software. There are several commercially available floating point cores on the market that are specifically optimized for use in FPGAs [58, 57]. Consequently, adding support for floating point arithmetic, is a relatively simple task. However, as discussed in section 1.2.4, the use of higher radix notations [60] and logarithm formats [61] can significantly reduce the hardware utilization in some circumstances. The decision about which floating point notation/format provides the optimal implementation for a specific data flow pipeline, can be made at run time, by the hardware generation tools. This is possible because the conversion tools possess detailed information about the number, type and connectivity of all the operations in the pipeline.

7.1.3 Optimization

As demonstrated in sections 6.3 and 3.4, loop unrolling can substantially increase the speedup factor produced when the code is converted to hardware. Although loop unrolling is a common software optimization technique, the optimum unroll factor is different depending on whether a section of code is being executed in software or in hardware. In addition, the unroll factor can have a dramatic effect on performance; too small a value can result in low levels of parallelism, thereby limiting the performance of the system; too large a value can result in inefficient usage of the available hardware resources, which in extreme cases can prevent the loop from fitting into the RC area. Since the optimal unroll factor is dependent on several parameters

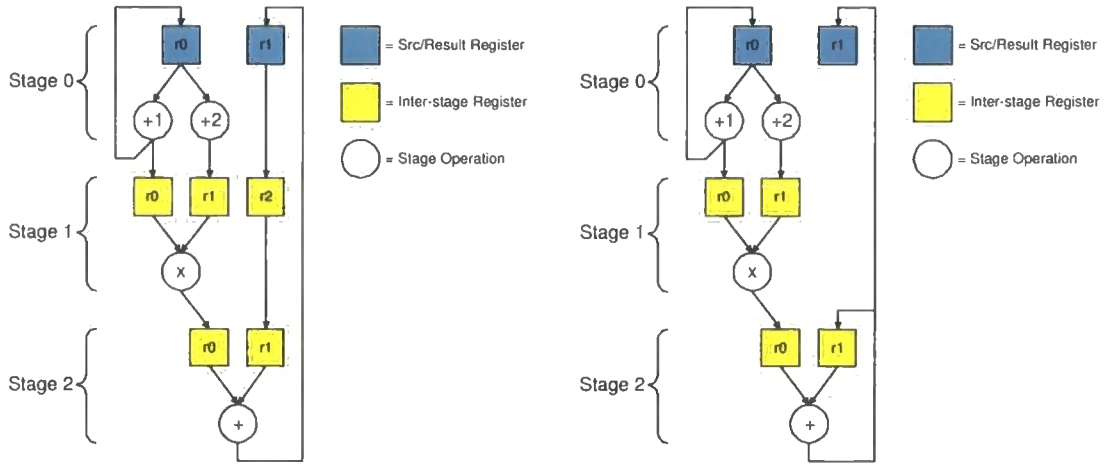
that may differ from one system to another (e.g. bandwidth, memory latency and RC area size), it is not possible to determine the most appropriate value at compile time. A potential solution to this problem is for the hardware generation tools to examine the code and to calculate the most suitable unroll factor. The conversion tools could then either unroll or re-roll the loop as appropriate, producing code which is specifically optimized for the particular system, without reducing the level of abstraction.

When a host platform becomes available that supports exceptions/interrupts, an error detection and roll back system could be implemented. This would resolve the pointer aliasing issue outlined in section 4.6.1.1, and also allow more aggressive optimizations to be performed. In this case, if the optimization were not suitable for a specific section of code, producing errors and therefore invalid data, the hardware execution would be terminated and the state of the system rolled back to a point before the error occurred. The hardware conversion system would then re-implement the hardware without the optimization that caused the error.

7.1.4 Scheduling

7.1.4.1 Operation Variants

Complex operations like multipliers, dividers and barrel shifters, can take a considerable number of logic cells to implement. Operations can be implemented using different methods, depending on the latency, throughput requirements and device usage constraints. The hardware generation software described in chapter 4, currently implements all operations with the minimum possible latency. Since in some cases the result of an operation is not required until further down the pipeline, data forwarders are added to the pipeline, as described in section 4.6.2. This provides the potential for the hardware generation tools to decrease the hardware utilization



(a) Feedback to stage 0

(b) Local feedback

Figure 7.1: Example data flow pipelines with and without the local feedback optimization

of an operation, albeit at the expense of increasing its latency without, however, affecting the overall performance of the system.

7.1.4.2 Local Feedback

In several of the test algorithms evaluated (e.g. Low pass filter), the number of clock cycles between iterations is artificially high, since the resultant values can only be passed backwards to the very first stage of the pipeline; this limits the performance. One such example is shown in figure 7.1(a), where the result of the addition operation on stage 2 is forwarded back to stage 0, delaying the next iteration until after the previous iteration finishes. This performance restriction can be removed by modifying the pipeline to include local feedback, so that the value of register "r1" on stage 0 is not required to start the next iteration (as shown in figure 7.1(b)).

The number of clock cycles between starting iterations in the pipeline has a significant effect on performance and is, as shown in equation 5.3, roughly proportional to the overall execution time. Performance will, therefore, be significantly enhanced by adding the capability to perform local feedback optimizations, such as in the simple example shown in figure 7.1, where this optimization has tripled the performance.

7.1.4.3 DMA Operation Scheduling

The current scheduling algorithm places operations onto the pipeline based solely on the data dependencies that are present. However, if the scheduling algorithm places more DMA operations onto a stage than the system can support, the pipeline is forced to stall. By altering the placement of DMA operations within the pipeline, the amount of idle time could be reduced, and thus performance increased. The exact placement of these operations would be influenced by the following factors:-

- The DMA issue constraints of the RC area.
- The number and type of DMA operations in the section of code being converted to hardware.
- The stages in the pipeline where data from load operations is required.

7.1.4.4 FPGA Tool Integration

Estimates of both the latency and also the hardware utilization for the operations that make up the data flow pipeline, are used extensively during the hardware generation process. In particular, the estimated latency is used during the scheduling phase to determine which operations can be packed onto the same pipeline stage, without exceeding the target propagation delay, as this would reduce the clock speed. Although the estimation process takes account of the effects of the four possible input configurations, its accuracy is severely limited due to the following factors:-

Logic cell packing The majority of the operations that will be scheduled onto the pipeline have two inputs, whereas the logic cells present in most FPGAs have four inputs. The tools provided by FPGA vendors automatically combine operations and pack them into the available logic cells. Exactly how this

is accomplished, has a considerable impact on both latency and hardware utilization.

Hardware resource type used Some structures, like memories or multipliers, can be implemented using either general purpose logic cells or dedicated hardware blocks present in the FPGA. The type of resource to be used is determined by the FPGA tools, and can have a significant effect on both latency and resource usage.

Routing delays The position of hardware blocks relative to that of connected blocks within the FPGA, is the major factor that influences routing delays, and as a consequence affects the overall delay between register stages.

In the current system, the hardware conversion tools are completely separate from the FPGA vendor's synthesis and PAR tools. By closely integrating these two tool chains, the limiting factors, listed above, may be eliminated, producing an optimized pipeline layout. In addition, the PAR tools could be made to relay information both about the routing congestion and also about any unmet timing constraints. This would enable the hardware generation tools to insert additional registers and thus, reorder the pipeline in an iterative process to produce the optimal configuration.

7.1.5 Hardware Software Integration

The instructions that trigger execution in the RC area are currently placed at the end of the program. As described in section 4.7.2, this approach avoids the need to relink the program in the event that the trigger instructions are larger than the loop that they replace. However, if the RC pipeline is triggered repeatedly, the additional jump instructions that are required can introduce a significant overhead. In cases where the number of trigger instructions is less than the number of instructions that the RC area replaces, it is possible to embed the trigger instructions into the body

of the program without relinking. This proposed optimization reduces the overhead associated with starting execution in the RC area, however this is at the expense of increasing the time required to enable or disable the hardware acceleration. Since the number of times that the RC area will be triggered is considerably higher than the number of times that the hardware acceleration will be enabled/disabled, this trade off is of overall benefit.

As mentioned in section 7.1.1, it is possible to automatically inline and thus incorporate function calls into the hardware data flow pipeline. Although this allows the conversion of sections of code that would otherwise be ineligible, it can lead to inefficient usage of hardware resources, in cases where the function is called conditionally on a small number of loop iterations. Such cases can easily be recognized by further analysis of the profile data that is used to identify candidate loops. These inefficiencies can be avoided by replacing the function call with a interrupt trigger. This enables the data flow pipeline within the RC area to be paused and execution of the function to be carried out in the software domain. Once the function call completes, the results would be passed to the data flow pipeline and execution resumed. This would not only improve the efficiency of the hardware utilization, but would also enable a loop to be converted to hardware that would otherwise be too large to fit into the RC area.

7.2 Platforms

7.2.1 MIPS

The results in section 5.3, indicate that substantial increases in performance can be obtained by performing the conversion to hardware. However, significant additional speed improvements can be obtained if the memory bandwidth to the RC area is

increased. By simply pipelining the memory interface, it is possible to perform one DMA operation per clock cycle which, in some test cases, results in a further tripling of performance. Overall, this enables performance improvements in the excess of two orders of magnitude, whilst still consuming a level of memory bandwidth which is inline with the capabilities of most modern embedded platforms.

Currently the FPGA exists in the same clock domain as the CPU core, forcing the two components to operate at the same clock speed. The maximum clock speed of the CPU is fixed at design time, however the maximum clock speed of the RC area is determined at runtime when its configuration is generated. As a result, in the majority of cases, the RC area is not operating at the optimal clock speed. Moving the FPGA to a separate clock domain would enable the RC area to run at its optimum clock speed, as determined by the hardware generation and synthesis tools that are described in section 7.1.4.4.

7.2.2 XD1

As described in section 6.5.2, pipelines with high numbers of DMA operations have lower maximum clock speeds due to the increased complexity of the cross bar switches that connect the load/store operations to the memory interface. Since the cross bars are inside the RC area, they can be easily modified to suit the specific data flow pipeline to which they are connected. Extra registers could be added to the cross bars for pipelines with high numbers of DMA operations, increasing the maximum clock speed at the expense of increasing the memory latency. Clearly, this is advantageous for highly parallel algorithms, as the additional latency would not decrease the overall throughput. However, for some applications which contain high numbers of data dependencies, the additional memory latency might result in an overall reduction in performance. Since, the level of parallelism is calculated during the early stages of the hardware conversion process, the conversion tools would be

able to determine whether adding additional registers to the cross bars is likely to increase performance. This information may then be used to decide whether or not the optimization is performed.

Once software support for the FPGA interrupt line is made available by Cray, many additional enhancements can be made to the hardware conversion process and accompanying execution environment. One key improvement would be the introduction of virtual memory. This could be effected by implementing an MMU inside the FPGA to perform the virtual to physical address translation. Any DMA transactions to memory pages that are not present in physical memory would pause execution in the RC area and trigger an interrupt that would cause the main CPU to re-load the page from disk. Once this had been accomplished, execution in the RC area would resume. This would allow DMA operations to be performed on the main SDRAM memory instead of on the local QDR-SRAMs. To reduce the effective latency of the main SDRAM, the QDR-SRAMs would be used to implement a cache. Since the RC area and the CPU would exist in the same virtual address space, the software/hardware integration could be performed automatically, as described in section 4.7.2. With the integration process fully automated, it would also be feasible to implement multiple loops inside the RC area, as in the case of the MIPS platform (see section 5.1).

7.2.3 Benchmark Algorithms

Once the various platform limitations outlined in sections 5.3 and 6.5.1 have been resolved, it would be possible to run a range of standard benchmarks and real applications. The relevant sections of these pieces of software could then be automatically converted to hardware. This would provide a much more detailed understanding of the performance improvements that can be obtained in real world applications. In addition to aiding the understanding and future development of the runtime tech-

niques outlined in this thesis, using standard benchmarks would also enable the performance to be compared with other reconfigurable computing platforms and tool sets.

7.3 An Ideal Reconfigurable Computing Platform

The majority of current reconfigurable computing platforms (such as those available from Cray [67] and SGI [31]), are based on existing, commonly used hardware architectures, with FPGAs grafted onto the system. Although these systems are capable of producing considerable performance improvements in excess of an order of magnitude, their performance will always be limited by the system architecture. To fully exploit the potential of the reconfigurable computing concept, the hardware conversion tools must be combined with a hardware platform that has been specifically designed with the reconfigurable computing environment in mind. The block diagram of the ideal reconfigurable processor is shown in figure 7.2.

7.3.1 Processor Integration

In many of the test algorithms evaluated, the RC area was triggered repeatedly (e.g. quick sort, FFT, Mandelbrot). As a result, the time taken to trigger execution can have a considerable impact on the overall performance of the system. To minimize the trigger time, the RC area can be integrated into the same die as the CPU core. This eliminates the high latency associated with slow, off chip busses. Additionally, if the CPU core were to be placed on the same die as the RC area, the RC area may be interfaced directly to the instruction pipeline via an APU port [22], further decreasing the time taken to transfer execution from the CPU core to the RC area.

It is worth noting that, the proposed system has only one CPU core, whereas many

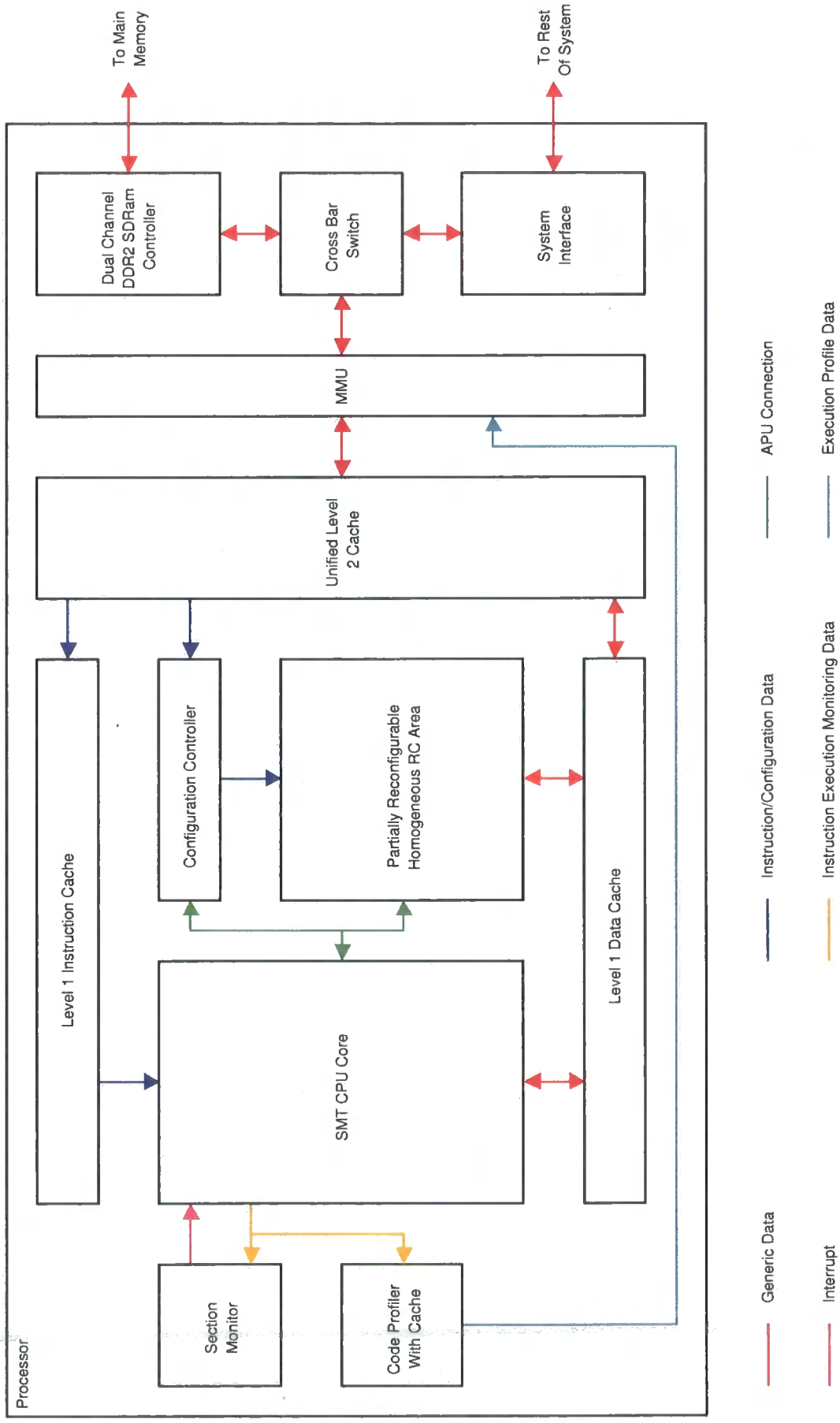


Figure 7.2: Block diagram of idealized CPU architecture

modern processors are dual or multi core [19, 20, 21, 118]. Moving to a dual core architecture, more than doubles the die area giving a maximum theoretical performance improvement of 2x. The results from both the MIPS and XD1 platforms (see sections 5.3 and 6.3 respectively), indicate that the die area used to implement the second CPU core would produce a significantly larger increase in performance, if it were to be used to implement an RC area. In addition to increasing performance, replacing a CPU core in a dual core system with an FPGA will also dramatically reduce the power consumption and thus the heat generated. This is due to the lower operating frequencies of FPGAs. This is highly beneficial as the thermal envelope and thus the operating temperature, is a major design consideration, with modern systems generating up to 130 watts of heat [119].

Execution in the RC area is started by executing one or more trigger instructions, as outlined in section 5.1.1. Once these trigger instructions have been issued, the CPU core will stall until execution in the RC area has completed. Having such a large area of silicon idle for even short periods of time, will significantly reduce the efficiency of the system. In addition, the CPU must be active in order to perform auxiliary tasks like: virtual memory paging and the execution of sub-functions (described in section 7.1.5). In order to improve efficiency and also to accommodate the execution of vital tasks, the CPU core in the ideal system implements SMT [18]. This enables the CPU to simultaneously handle and issue instructions from multiple thread contexts. Since the RC trigger instructions only stall the execution of a single thread, the CPU can continue to execute instructions from other threads, thus enabling the RC area and the CPU core to run in parallel.

7.3.2 Homogeneous RC Area

The following is a list of ideas and concepts that should be incorporated into the RC area of an ideal reconfigurable computing platform:-

- Partial reconfigurability
- Homogeneous structure
- Configuration controller
- Specialized hardware
- Design for PAR
- Clock domains

7.3.2.1 Partial Reconfigurability

The FPGA resources used to implement most software loops in hardware, are typically, significantly lower than the resources available (in all the results thus far obtained, the FPGA utilization did not rise above 20% for any of the test algorithms). To obtain the highest possible improvement in performance, the RC area can be used to implement several sections of code at the same time. Due to the dynamic nature of computing environments, the specific hardware blocks that the RC area will be required to implement, changes over time. The reconfiguration required by this, involves the RC area being idle. To minimize the associated performance impact, the RC area can be partially reconfigured so that one section of the hardware is actively processing data, whilst another area is being configured for the next task.

7.3.2.2 Homogeneous Structure

As a direct result of the runtime scheduling and reconfiguration (described in section 7.3.2.1), it is not possible to determine exactly where, in the RC area, a hardware block will be placed during the synthesis and PAR stages. To avoid the need to re-run the PAR process every time the RC area is reconfigured, the RC area must

have a homogenous structure that allows location independent hardware blocks to be synthesized.

7.3.2.3 Configuration Controller

To decrease the time taken to reconfigure the RC area, the configuration controller is connected to a cache instead of directly to the MMU. As such, pre-fetching and caching of configuration data will greatly speedup the reconfiguration process. Since the configuration data for even small hardware blocks can be larger that the level 1 instruction cache, "cache trashing" is likely to occur (large amounts of useful cache data, flushed out to make room for a large, infrequently used data sets). To prevent this "cache trashing" the configuration controller is connected directly to the much larger, level 2 cache.

7.3.2.4 Specialized Hardware

Early FPGAs contained only logic cells together with their associated routing matrices. However, modern FPGAs also contain block RAMs and hardware multipliers [114, 115]. These additional, specialized hardware resources can significantly increase the performance of hardware implemented on these devices. Due to the complex nature of commonly used floating point operations, FPGAs designed for reconfigurable computing would benefit from additional specialized hardware primitives. As mentioned in section 7.3.2.2, the RC area must have a regular, homogenous structure to allow for the relocation of hardware blocks. Although the addition of specialized hardware resources, like multipliers, disrupts the homogenous structure locally, if they are evenly distributed throughout the RC area a regular structure can be maintained at the global level. Accordingly, it is still possible to create relocatable hardware blocks whilst maintaining the advantages of specialized hardware.

In order to maintain the appearance of homogeneity, FPGA resources could only be allocated at the granularity of the smallest repeating hardware unit. The hardware utilization within a block will decrease slightly owing to the increased size of the repeating units, caused by the addition of memory blocks, multipliers etc. This minor decrease in efficiency, is more than outweighed by the increase in performance produced by the presence of specialized hardware primitives.

7.3.2.5 Design For Place And Route

Modern FPGAs are optimized to provide the most efficient usage of die area at the expense of ever increasing synthesis and PAR times. Although this trade off is ideal for situations where the hardware is compiled once, and used to configure many devices, it is not well suited to environments, such as runtime reconfigurable computing, where the hardware compilation process occurs repeatedly. By changing the structure of the routing matrix in the FPGA, it is possible to decrease the amount of memory used during the PAR process by a factor of 18, whilst at the same time reducing the time taken, by a factor of 3 [44]. This kind of optimization will have a significant impact on the system, given that the PAR dominates all the other stages of the conversion process, in terms of both execution time and memory usage.

7.3.2.6 Clock Domains

Although not shown in figure 7.2, the RC area exists in a separate clock domain from the CPU core, and all other blocks in the processor. This is primarily due to the fact that the reconfigurable nature of the RC area, prevents it from running at the higher clock speeds used by the CPU. However, the presence of a clock domain boundary between the RC area and the rest of the system, allows its clock speed to be set at the optimal value for the specific hardware being implemented. As the

RC area may be used to simultaneously implement multiple hardware blocks, the RC area also contains multiple clock distribution trees. This enables each block to be clocked at its optimum frequency, independently of the surrounding blocks. The presence of multiple, independent clock trees is a common feature of many modern FPGAs [114, 115].

7.3.3 Memory Sub-System

As shown in figure 7.2, the DMA channel from the RC area connects to the memory hierarchy at the same point as the CPU core (i.e. the level 1 data cache). Not only does this mean that the RC area will benefit from both the level 1 and the level 2 caches, but it also results in the system being naturally cache coherent. There is therefore, no overhead associated with the snoop traffic that would otherwise be required to keep the data in parallel caches synchronized. The configuration data for the RC area is read directly from the unified level 2 cache, instead of from either of the level 1 caches. As described in section 7.3.2.3, this eliminates the possibility of "cache trashing". However, as a result, newly generated configuration data will not be visible to the configuration controller, unless it has been first flushed back to main memory from the level 1 data cache (this is similar to the cache coherency problem faced by applications using self modifying code). One possible solution, could be to implement additional snoop hardware to ensure cache coherency, however, the additional hardware might result in a lower overall clock speed for the system. Since only a small section of software, that runs infrequently, handles the hardware generation and configuration, the cache coherency can be easily and efficiently be managed in the software domain.

Since the RC area is connected to the memory hierarchy before the MMU, any hardware inside the RC area exists in a virtual address space instead of in the physical address space. This eliminates many of the integration and hardware conversion is-

sues that are faced by existing reconfigurable computing platforms such as the Cray XD1 [67].

The computational rate of the RC area is considerably higher than that of the CPU core, resulting in memory bandwidth requirements that are significantly higher than those of CPUs [17]. This is demonstrated by the fact that, many of the algorithms tested on both the MIPS and XD1 platforms (see sections 5.3 and 6.3 respectively) became bandwidth limited after they were converted to hardware. To provide the RC area with the maximum possible bandwidth, the RC area is connected, through the caches, to the latest dual channel DDR2 memory architecture [120]. Additionally, to decrease memory latency and to further increase performance, the memory controller is integrated directly into the processor. This approach has already been successfully implemented in the AMD Athlon and Opteron lines of processors and Intel plan to incorporate it in future products.

7.3.4 Code Profiling

A hardware code profiler, similar to the one described in section 3.1.1.1, monitors the software execution. Because the profiling operations are performed by dedicated hardware, there is little or no performance impact. Since the profiler, contains its own small cache, the bandwidth consumed by the profile data is minimal. As the profile data will never be read back by the profiler block, connecting the block to one of the data caches would not result in an increase in performance. In fact, due to the "read before write" policy of these caches, the overall performance would be reduced. In order to prevent this, the code profiler is connected directly to the MMU. The metrics gathered by the profiler are used by the hardware conversion software to determine which sections of code consume the most CPU time, and therefore to determine which sections should be converted to hardware.

The code profiler monitors the execution of all the code running on the system, and as a result the hardware conversion software, itself, will be profiled. This, in turn, will result in the hardware generation software converting computationally intensive sections of itself, to hardware. This will further speedup the generation of hardware for all other applications.

7.3.5 Hardware Scheduling

As mentioned previously in section 7.3.2.1, the RC area is capable of simultaneously implementing multiple sections of code in hardware. This, combined with the fact that the hardware required will change as the execution of the application progresses, means that a runtime scheduling algorithm is required to load the hardware blocks into the RC area, at the correct time. This process is described in section 1.2.3 [52, 53].

Due to the limited size of the RC area, it is not possible to load all the hardware blocks generated by the conversion software. As it is not always possible to predict which sections of code will be executed, it may be that the corresponding hardware block is not present in the RC area, when it is required. If this occurs, the section will be executed in the software domain by the SMT CPU core. A dedicated hardware block called the, "section monitor" (see figure 7.2), monitors the code being executed by the CPU core. If the code being executed matches one of the sections that the monitor is configured to detect, an interrupt will be raised. This interrupt then triggers the hardware scheduling software which will then load the corresponding section of hardware into the RC area. Once this has been completed, the scheduler will modify the application software to use the newly instantiated hardware. The section monitor may also be configured to only raise an interrupt, if the number of times a section of code is used in a set time period, exceeds a threshold value. This allows the system to detect changes in the amount of execution resource used by

different sections of code, without the generation of large numbers of time consuming interrupts.

The scheduling software supported by the two hardware monitoring blocks would gather information about code usage and RC area congestion. This information, may then be used by the hardware generation tools to inform the generation of additional variants of the hardware blocks. For example, if the RC area can only accommodate 2 out of 3 blocks that are simultaneously required, the system could automatically adapt and generate new versions of the blocks with lower unroll factors, thus allowing all 3 blocks to be concurrently implemented in the RC area.

7.4 Heterogeneous Computing

Many different types of computational resource have been developed since the birth of the "Von Neumann architecture" in 1945 [1]. A few existing architectures, together with their advantages and disadvantages, are listed below:-

CISC/RISC CPU Conventional CISC and RISC CPUs are by far the most common form of computational resource. This is due to their flexibility, scalability and backwards compatibility. Their basic architecture is based on an instruction pipeline that executes a series of sequential instructions, the generic nature of which, enables this type of processor to perform an almost limitless number of different tasks. However, this flexibly comes at the cost of efficiency, for this type of processor has a very low computational rate, compared to the die area used. Common examples of this architecture include: the AMD Athlon [121], the Intel Pentium 4 [122] and the MIPS [102] range of processors.

VLIW Very Long Instruction Word (VLIW) processors (e.g. the TriMedia [85] and Efficeon [123]) have a lot in common with CISC and RISC CPUs, as they

are also based on an instruction pipeline. However, they differ in that, the instructions are grouped together to form bundles, where the instructions in each bundle are independent and are executed in parallel with each other. This explicit parallelism, leads to greater performance per unit die area, due to the absence of complex dependency checking and scheduling hardware. The use of VLIW cores has been limited to embedded applications where the code is compiled for a specific processor, due to the inherent lack of backwards compatibility and scalability provided by this architecture.

FPGA In contrast to CISC, RISC and VLIW processors, FPGAs do not contain an instruction pipeline, instead they contain an array of logic elements that can be connected together to form a great variety of different circuits. Although they are capable of extreme computational workloads, a very high level of data parallelism is required to enable this. This limitation, is further compounded by the fact that FPGAs inherently have considerably lower clock speeds than those exhibited by conventional processors. In addition, the simple nature of the logic cells used in FPGAs, can lead to large inefficiencies when implementing such complex functions as floating point arithmetic. The low level of abstraction between the configuration data and the device itself, leads to a total lack of backwards compatibility.

Array processor Array processors (e.g. ClearSpeed CSX600 [124]) are similar to FPGAs, as they also consist of regular arrays of elements, connected together by a routing matrix. Array processors differ from FPGAs, in that each element is capable of performing complex mathematical functions, instead of just the simple 4 input logic functions of the FPGA. Although the architecture of an Array processor leads to highly efficient floating point implementations, it is not as efficient in performing bitwise and integer operations as the FPGA. Like FPGAs, array based processors can also suffer from a lack of backwards compatibility.

All the architectures currently available have disadvantages, and many exist due to their superior performance in certain, specialized areas. It is, however, possible to create a system that will provide the optimal performance for every type of application by combining, CISC, VLIW, and Array processors, together with FPGAs, in a single platform. Since CISC processors have high levels of abstraction, and can thus maintain backwards compatibility, they are the best choice for the primary processor in the system, and therefore the architecture that all applications would be written for. Runtime monitoring software would analyze computationally intensive sections of code, to determine which type of computational resource is best suited to performing each task. The corresponding dynamic translation layer could then convert the code to the appropriate assembly language or hardware configuration. This approach would provide the performance advantages of a specialized system, whilst still maintaining compatibility with existing software applications.

For a heterogeneous computing system to provide significant increases in performance, a viable hardware platform is required. Some multi-core, multi-architecture devices have recently been introduced (e.g. the Cell processor [118]), however, because in these devices, the number and type of the processing elements is fixed, this category of system will not be optimal for every class of application. One possible solution, is to base the hardware platform on a blade architecture with a very high bandwidth between the compute blades (e.g. Cray XD1 [67]). Blades with different types of computational resources could then be created. This approach would allow system administrators to optimize systems for the specific needs of their applications, by altering the proportions of the various computational resources present in the system. Blade architectures also provide an open, flexible platform to ease the introduction of new types of processor and/or new reconfigurable hardware.

Software to software translation is commonplace (e.g. Java [49] and Transmeta [50]), and hardware to software translation is addressed throughout this thesis. This com-

bined with the fact that the hardware platforms required, either already exist or are relatively easy to develop, indicate that the concept of a heterogeneous computing environment is achievable.

7.5 Future Research

In addition to implementing the additional optimizations and concepts outlined in this chapter, the following should be investigated during any future work:-

Cache configuration A lot of work has be conducted to determine the optimal cache size and organization for CPUs with various type of memory subsystems. Although a reconfigurable computing system will be used to execute the same algorithms as conventional CPUs, the memory bandwidth RC systems require is significantly higher due to the increase in computation rate. In addition any bandwidth required for instruction fetches is eliminated. A detailed analysis of the effects of different cache architectures and hierarchies should be conducted.

Clock speed to bandwidth ratio Because RC systems consume high amounts of memory bandwidth in investigation into the relationship between RC area clock speed and required bandwidth should be performed. Although some work in this field has already been performed [17] a good understanding of the levels of bandwidth required is lacking, particularly in the context of a large scale RC system with multiple high performance CPUs (e.g. the Cray XD1)

High level optimizations The possible use of high level optimization to restructure algorithms should be investigated. Typically code is optimized for software execution unfortunately this does not provide the best solution for reconfigurable computing platforms. In the future it might be possible for compilers

to include some form of meta data in the final program binary that could be processed by the hardware conversion tools at runtime. This could provide the additional information required to make many high level optimizations possible.

Virtual memory Since the RC area is usually connected directly to the memory sub-system it exists in the physical address space. However the instructions that the RC hardware will be based on operate in a virtual address space. One solution to this problem is to implement a full MMU between the RC area and the memory sub-system. Since the core code loops that are converted to hardware usually only access a small sub-set of the data present within a large application, it may be possible to create a more effective memory mapping system.

7.6 Summary

By extracting parallelism in an additional orthogonal direction, and combining this with the new optimisation techniques described in section 7.1, it is possible to considerably increase the performance achieved by the hardware generation tools described in chapter 4. Several of these optimisations will also reduce the hardware resources required, and consequently allow a greater proportion of an application to be converted to hardware, thus further increasing performance.

A detailed design of an idealised computing platform is presented, which allows the hardware conversion tools to fully exploit the potential offered by the reconfigurable computing concept. In addition, a heterogeneous computing system is proposed, which combines reconfigurable hardware with a variety of other computational resources. This would enable runtime translation of code to the most appropriate computational resource, producing dramatic increases in performance over a wide

range of applications, whilst minimizing power consumption and maintaining backwards compatibility.

Chapter 8

Conclusion

Over the past forty years, the performance of conventional computing platforms has increased exponentially, in line with "Moore's Law" [2]. Although this has resulted in enormous levels of computing power being available, some applications such as computer games, simulations, multimedia programs and others, require even greater levels of performance. In addition to this need for increased performance, power consumption is becoming a critical factor in the design of modern microprocessors. At up to 130Watts per processor [119], not only are the power requirements for super computers and large clusters a major cost issue, but the heat generated, causes thermal design problems, for all modern computing systems.

The use of dedicated hardware (e.g. PC graphics cards [5, 6]) can dramatically increase performance and reduce power consumption, compared to a conventional software solution. However, dedicated hardware is only capable of performing the specific task for which it was designed, leaving it idle for significant amounts of time when not required. This has limited its usage to frequently performed computational tasks. The use of reconfigurable hardware (e.g. FPGAs), has long been seen as a potential solution to this problem, as it can be used to provide the increased performance and also the reduced power consumption that is associated with dedicated

hardware. In addition, the reconfigurable nature provides the flexibility associated with software solutions. Historically there have been two major barriers to the widespread adoption of this technology:-

Hardware platforms The high performance of hardware based solutions, leads to high bandwidth requirements, which necessitates the reconfigurable hardware to be closely integrated with both the CPU and the rest of the system. However in the past, reconfigurable hardware, has usually been added to existing platforms as an afterthought. Consequently the RC area was usually connected to one of the peripheral buses, which resulted in low bandwidth and poor performance.

Abstraction and backwards compatibility Due to its low level nature, reconfigurable hardware (unlike CPUs) has a very low level of abstraction between the configuration data and the underlying hardware. Consequently it is not possible, at the present time, to produce scalable systems that allow backwards compatibility.

In this thesis it has been shown that by using JIT techniques, it is possible to convert computationally intensive sections of software to hardware at runtime, and thereby fully abstract the RC area from the application. Not only does this allow the performance of the system to be scaled, as new, faster FPGAs become available, but it also enables applications that were not written for reconfigurable computing, to be accelerated by this technology.

To evaluate the performance improvements that could, potentially, be provided by runtime JIT techniques, a complete tool flow has been developed and described, which together with various test algorithms, was applied to two hardware platforms:-

MIPS platform The MIPS platform contains a small, embedded class CPU and has an FPGA connected directly to the instruction pipeline. As the CPU

core and the FPGA are in the same clock domain, they run at the same clock speed of 16 MHz. This single clock domain approach, significantly reduces the complexity of the hardware.

Cray XD1 platform The Cray XD1 is one of a new generation of supercomputers that integrate FPGAs into the core of the system, giving the FPGA high bandwidth and low latency access to the main memory. The CPU in the test system was a 2.2 GHz AMD Opteron, which was paired with a Virtex II Pro 50 FPGA, having a maximum clock speed of 200 MHz.

The automatic hardware acceleration of programs on the MIPS test platform, produced performance increases ranging from 20% to a factor of 54.9, with many of the test algorithms being over 10x faster than in the original software case. The results obtained from the Cray XD1 platform, ranged from a 7x decrease in performance to an 18x increase in performance. The reduction in performance, in some of the test cases, was due to the low levels of parallelism in the original software, combined with the fact that the FPGA ran at clock speeds up to 15 times slower than the CPU. It is worth noting, that in the majority of cases, the performance of the FPGA was limited by the available bandwidth, and not by the computational speed of the FPGA itself. Although significant increases in performance, in excess of an order of magnitude, have been obtained, it is clear from the results presented that reconfigurable computing cannot accelerate every test case. However, it is possible for the JIT hardware generation software to easily identify and screen out non-ideal kernel loops, and thus prevent reductions in performance.

Although significant increases in performance are produced by the JIT tool flow, higher levels of performance can usually be obtained by handwriting the hardware in an HDL (e.g. VHDL, SystemC [40]). However, this requires a time consuming and expensive process, that needs to be performed for every target platform. In comparison, the JIT based automatic conversion system outlined in this thesis

provides the advantages of scalability and total abstraction. As a result, the die area being utilized in the latest generation of CPUs to instantiate multiple cores, [119, 118] would produce much higher performance gains, if it were used instead, to instantiate an RC area. This solution would also help address the power consumption and thermal issues that are currently being faced by the industry. The proposed JIT based tool flow does present some additional problems and drawbacks:-

Verification A runtime tool flow that dynamically changes the nature of an executable presents some verification problems. This is because it is almost impossible for the developer to verify the correctness of the system as a whole, however this level of verification is only usually required for life support, aviation, and other critical applications. As such the vast majority of applications can still benefit from the increased performance provided by the use of JIT techniques. In these cases the application developer must verify that there application functions correctly on the platform without hardware acceleration, and the JIT tool flow vendor must verify the conversion process itself. It can therefore be assumed that in the majority of cases any application will still function correctly once it has been hardware accelerated. The concept of relying on abstraction in the verification model is common place (e.g. software vendors do not verify there applications against every possible combination of PC hardware components). It is worth noting that this approach is also used in the verification of many other JIT based tool flows (e.g. Java [49], FX32 [48], Crusoe [50]).

Time consuming tool flow Although the time required to perform the place and route operations, and therefore the overall conversion time is quite high, this will be reduced by the introduction designed for PAR (see section 7.3.2.5). In most cases the high conversion time does not pose a significant problem as the majority of programs fall into one of the following two categories; applications

that are run repeatedly (e.g. word processing) where the conversion can be perform during system idle time and the results stored for future runs, or programs that have very long run-times (e.g. complex simulations) where the time required to perform the conversion becomes insignificant.

Debugging Many threading related bugs are sensitive to the timing on the execution process. As such these bugs could be exposed and cause crashes on platforms that implement dynamic hardware acceleration. As this issue is common to any increase in system performance (e.g. increasing CPU clock speeds) care should be taken when writing multi-threaded applications for any platforms. In addition software development tools exist that can help identify these problems before the software is released.

Application specific performance increases The use of reconfigurable hardware can produce significant increases in performance. However this increase is very dependant on the application being accelerated, in some cases the performance might not increase at all. This drawback is common to virtually every method of increases performance (e.g. increasing cache sizes, number of execution units, memory bandwidth, etc).

Many different types of computational resource currently exist (e.g. RC areas [115, 114], floating point array processors [124], conventional superscalar CPUs [122, 121]), each with their own inherent strengths and weaknesses. In the future, it should be possible to create a system that harnesses the combined power of all of these types of computation resource, whilst still providing scalability, abstraction and backwards compatibility. This could be achieved by combining existing software to software conversion techniques [50, 48, 49], with the software to hardware translation outlined in this thesis. The conversion software would locate intensive sections of code, identify the most suitable computational resource, and automatically migrate the code accordingly. This ability to dynamically translate and move sections of code

from one resource to another, would produce a dramatic increase in performance and would also significantly reduce power consumption and operating costs.

Bibliography

[1] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, Apr 1965. [Online]. Available: ftp://download.intel.com/research/silicon/moorespaper.pdf

[3] A. Peleg and U. Weiser, "MMX technology extension to the intel architecture," *IEEE Micro*, vol. 16, no. 4, Aug 1996. [Online]. Available: http://www.eecs.lehigh.edu/~mschulte/ece401/papers/mmx.ps

[4] L. Gwennap, "AltiVec vectorizes PowerPC," *Microprocessor Report*, vol. 12, no. 6, May 1998. [Online]. Available: http://docencia.ac.upc.edu/ETSETB/SEGPAR/microprocessors/altivec %20(mpr).pdf

[5] (2005, May) GeForce 6 series. nVidia Corporation. [Online]. Available: http://www.nvidia.com/page/geforce6.html

[6] (2005, May) Radeon X850 graphics technology. ATI Technologies Inc. [Online]. Available: http://www.ati.com/products/radeonx850/index.html

[7] (2005, May) Nexperia PNX8550. Philips Semiconductors. [Online]. Available: http://www.semiconductors.philips.com/acrobat/literature/9397/75012469.pdf

[8] (2005, May) OMAP 2 architecture: OMAP2420 processor. Texas Instruments. [Online]. Available: http://focus.ti.com/pdfs/wtbu/TI_omap2420.pdf

[9] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, 2004, pp. 162–170. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/fpga04_anal.pdf

[10] G. Lu, H. Singh, M.-H. Lee, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho, "The MorphoSys parallel reconfigurable system," in *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, 1999, pp. 727–734.

[11] M. Sima, S. Cotofana, S. Vassiliadis, J. T. van Eijndhoven, and K. Vissers, "MPEG-compliant entropy decoding on FPGA-augmented TriMedia/CPU64," in *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2002, pp. 261–270.

[12] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Apr 1997, pp. 12–21.

[13] M. J. Wirthlin, "A dynamic instruction set computer," in *IEEE Symposium on FPGA's for Custom Computing Machines.* IEEE Computer Society, 1995, pp. 99–107.

[14] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Liu, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," *IEEE Transactions On Computers*, vol. 53, no. 11, pp. 1436–1448, 2004.

[15] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The raw benchmark suite: computation

structures for general purpose computing," in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 1997, p. 134. [Online]. Available: http://www.crhc.uiuc.edu/ mfrank/pubs/Babb-1997-FCCM.pdf

[16] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, and M. Szydlo, "Security analysis of a cryptographically-enabled," in *14th USENIX Security Symposium*, 2006, pp. 1–16. [Online]. Available: https://www.usenix.org/events/sec05/tech/bono.html

[17] S. Derrien and S. Rajopadhye, "FCCMs and the memory wall," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2000, pp. 329–330.

[18] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 322–354, 1997.

[19] (2005, May) Intel dual-core processors. Intel Corporation. [Online]. Available: http://www.intel.com/technology/computing/dual-core/?iid=search&

[20] (2005, May) Introducing multi-core technology. Advanced Micro Devices, Inc. [Online]. Available: http://multicore.amd.com/Technology/

[21] T. Takayanagi, J. L. Shin, B. Petrick, J. Y. Su, H. Levy, J. H. P. Son, N. Moon, D. B. D, U. Nair, M. Singh, V. Mathur, and A. S. Leon, "A dual-core 64-bit ultraSPARC microprocessor for dense server applications," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 1, Jan 2005.

[22] A. Ansari, P. Ryser, and D. Isaacs, "Accelerated system performance with APU-enhanced processing," *Xcell*, no. 52, pp. 36–39, 2005. [Online].

Available: http://www.xilinx.com/publications/xcellonline/xcell_52/xc_pdf/xc_xcell52.pdf

[23] (2005, May) Custom instructions. Altera Corporation. [Online]. Available: http://www.altera.com/products/ip/processors/nios2/features/ni2-cust_instructions.html#custom_instructions

[24] R. Laufer, R. R. Taylor, and H. Schmit, "PCI-PipeRench and the SWORDAPI: A system for stream-based reconfigurable computing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 1999, pp. 200–208.

[25] S. D. Haynes, P. Y. K. Cheung, W. Luk, and J. Stone, "SONIC-a plug-in architecture for video processing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 1999, pp. 280–281. [Online]. Available: http://www.ee.ic.ac.uk/pcheung/publications/fpl99_sonic.pdf

[26] (2005, May) Nallatech FPGA computing solutions. Nallatech Inc. [Online]. Available: http://www.nallatech.com/?node_id=1.2.1&id=1

[27] C. Plessl and M. Platzner, "TKDM A reconfigurable co-processor in a PCs memory slot," in *IEEE International Conference on Field-Programmable Technology*, Dec 2003, pp. 252–259. [Online]. Available: http://www.tik.ee.ethz.ch/~plessl/publications/fpt03/fpt03.pdf

[28] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee, "Pilchard A reconfigurable computing platform with memory slot interface," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2001, pp. 170–179.

[29] (2005, Jun) SRC Computers Inc. - Hardware elements. SRC Computers Inc. [Online]. Available: http://www.srccomp.com/HardwareElements.htm

[30] (2005, Jun) Application acceleration with FPGA-based reconfigurable computing. Cray Inc. [Online]. Available: http://www.cray.com/products/xd1/acceleration.html

[31] (2004, Nov) Extraordinary acceleration of workflows with reconfigurable application-specific computing from SGI. Silicon Graphics Inc. [Online]. Available: http://www.sgi.com/pdfs/3721.pdf

[32] (2005, May) The hypercomputer product line. Star Bridge Systems Inc. [Online]. Available: http://www.starbridgesystems.com/products/hardware.html

[33] M. Verderber, A. Zemva, and D. Lampret, "HW/SW partitioned optimization and VLSI-FPGA implementation of the MPEG-2 video decoder," in *Proceedings of the conference on Design, Automation and Test in Europe.* IEEE Computer Society, 2003, pp. 238–243.

[34] (2005, May) Case study: Oil & gas - seismic exploration. Nallatech Ltd. [Online]. Available: http://www.nallatech.com/mediaLibrary/images/english/970.pdf

[35] P. Waldeck and N. Bergmann, "Dynamic hardware-software partitioning on reconfigurable system-on-chip," in *IEEE International Workshop on System-on-Chip for Real-Time Applications*, Jun 2003, pp. 102–105.

[36] T. Maruyama and T. Hoshino, "A C to HDL compiler for pipeline processing on FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 101–110.

[37] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and compiling applications for RaPiD," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 116–125.

[38] (2005, May) Complete design environment for C-based algorithmic design entry, simulation and synthesis. Celoxica Ltd. [Online]. Available: http://www.celoxica.com/products/dk/default.asp

[39] (2004, Oct) A true software approach to FPGA programming. Mitrionics AB. [Online]. Available: http://www.mitrion.com/press/Mitrion_whitepaper_041030.pdf

[40] G. Arnout, "SystemC standard," in *Asia South Pacific Design Automation Conference*, 2000, pp. 573–578.

[41] J. Hopf, G. S. Itzstein, and D. Kearney, "Hardware join java: A high level language for reconfigurable hardware development," in *IEEE International Conference On Field Programmable Technology*, Dec 2002, pp. 311–347.

[42] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[43] R. Lysecky, F. Vahid, and S. Tan, "Dynamic FPGA routing for just-in-time FPGA compilation," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, 2004, pp. 954–959. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/dac04_jitfpgaroute.pdf

[44] ——, "A study of the scalability of on-chip routing for Just-in-Time FPGA compilation," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/fccm05_jitroute.pdf

[45] J. M. P. Cardoso and H. C. Neto, "Macro-based hardware compilation of Java(tm) bytecodes into a dynamic reconfigurable computing system," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999, pp. 2–11.

[46] J. L. Schilling, "The simplest heuristics may be the best in Java JIT compilers," *ACM SIGPLAN Notices*, vol. 38, no. 2, pp. 36–46, Feb 2003.

[47] A. Gordon-Ross and F. Vahid, "Frequent loop detection using efficient non-intrusive on-chip hardware," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 117–124. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/cases03_profile.pdf

[48] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: a profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, Mar 1998.

[49] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. mei W Hwu, "Java bytecode to native code translation: the caffeine prototype and preliminary results," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, 1996, pp. 90–99.

[50] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*, 2003, pp. 15–24. [Online]. Available: http://www.cs.aau.dk/~fleury/bug_cms/CMS_Reverse/Papers/DGB03.pdf

[51] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, Apr 1967.

[52] H. Walder and M. Platzner, "Online scheduling for block-partitioned reconfigurable devices," *Design Automation and*

Test in Europe, pp. 290–295, 2003. [Online]. Available: http://www.tik.ee.ethz.ch/~walder/HomePage/XFORCES/DATE03.pdf

[53] A. Ahmadinia, C. Bobda, and J. Teich, "A dynamic scheduling and placement algorithm for reconfigurable hardware," *Lecture Notes In Computer Science*, vol. 2981, pp. 125–139, Mar 2004. [Online]. Available: http://www12.informatik.uni-erlangen.de/publications/pub2004/ABT04.pdf

[54] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 22–36. [Online]. Available: http://www.ece.wisc.edu/~kati/Publications/Li_FCCM00.pdf

[55] M. Sánchez-Élez, M. Férnandez, R. Maestre, F. Kurdahi, R. Hermida, and N. Bagherzadeh, "A complete data scheduler for multi-context reconfigurable architectures," in *Design Automation And Test In Europe*, 2002, pp. 547–552. [Online]. Available: http://www.eng.uci.edu/comp.arch/new_pubs/c84.pdf

[56] (2005, June) Floating point cores. Nallatech. [Online]. Available: http://www.nallatech.com/mediaLibrary/images/english/2432.pdf

[57] (2005, June) Double precision floating point cores. Nallatech. [Online]. Available: http://www.nallatech.com/mediaLibrary/images/english/3269.pdf

[58] (2005, June) Quixilica® floating point cores. QinetiQ. [Online]. Available: http://www.qinetiq.com/home_rtes/quixilica_products/firmware_cores/quixillica_fp.SupportingPar.0001.File.pdf

[59] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, pp. 185–194. [Online]. Available: http://www.doc.ic.ac.uk/~oskar/pubs/fccm03.pdf

[60] B. Catanzaro and B. Nelson, "Higher radix floating-point representations for FPGA-based arithmetic," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[61] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. S. Hemmert, and K. Underwood, "A comparison of floating point and logarithmic number systems for FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[62] F. Vahid. (2005, Jul) Warp processors. [Online]. Available: http://www.cs.ucr.edu/~vahid/warp/

[63] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *DAC '03: Proceedings of the 40th conference on Design automation*, 2003, pp. 250–255. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/dac03_dhs.pdf

[64] R. Lysecky and F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, Mar 2005, pp. 18–23. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/date05_warp_microblaze.pdf

[65] G. Stitt, Z. Guo, W. Najjar, and F. Vahid, "Techniques for synthesizing binaries to an advanced register/memory structure," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, pp. 118–124. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/fpga05_binsyn.pdf

[66] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *Proceedings of the 2004 ACM*

SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, 2004, pp. 249–256.

[67] (2005, Jun) Cray XD1 supercomputer. Cray Inc. [Online]. Available: http://www.cray.com/products/xd1

[68] R. Lysecky and F. Vahid, "A configurable logic architecture for dynamic hardware/software partitioning," in *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, vol. 1, no. 1, Feb 2004, pp. 480–485. [Online]. Available: http://www.cs.ucr.edu/~vahid/pubs/date04_clf.pdf

[69] G. Stitt and F. Vahid, "Hardware/software partitioning of software binaries," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002, pp. 164–170.

[70] (2005, May) Highly integrated, programmable system-on-chip (SoC). Philips Electronics. [Online]. Available: http://www.semiconductors.philips.com/products/nexperia/

[71] P. Lieverse, P. V. D. Wolf, K. Vissers, and E. Deprettere, "A methodology for architecture exploration of heterogeneous signal processing systems," *Journal of VLSI Signal Processing Systems*, vol. 29, no. 3, pp. 197–207, Nov 2001.

[72] D. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, and P. Fung, "System design methodologies: aiming at the 100 h design cycle," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp. 70–82, Mar 1996.

[73] (2005, May) Virtio - Virtual platforms for embedded system design. Virtio. [Online]. Available: http://www.virtio.com

[74] (2005, May) TriMedia SDE. Philips Semiconductors. [Online]. Available: http://www.alacron.com/downloads/vncl98076xz/sde_2_75006255.pdf

[75] P. R. Panda, "SystemC: a modeling platform supporting multiple design abstractions," in *Proceedings of the 14th international symposium on Systems synthesis*, 2001, pp. 75–80.

[76] J. Gosling, B. Joy, G. L. S. Jr, and G. Bracha, *The Java^{TM} Language Specification*, 3rd ed. Addison-Wesley Professional, Jun 2005. [Online]. Available: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[77] R. K. Gupta and S. Y. Liao, "Using a programming language for digital system design," *IEEE Design & Test*, vol. 14, no. 2, pp. 72–80, Apr 1997.

[78] J. Gong, D. D. Gajski, and A. Nicolau, "Performance evaluation for application-specific architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 4, pp. 483–490, Dec 1995.

[79] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: a system-level exploration platform for network processors," *IEEE Design & Test*, vol. 19, no. 6, pp. 17–26, Nov 2002.

[80] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *Computer*, vol. 33, no. 2, pp. 37–45, Feb 2000.

[81] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, pp. 24–43, Sep 2000. [Online]. Available: http://courses.ece.uiuc.edu/ece512/Papers/itaniumarchitecture.pdf

[82] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 44–55, Mar 2003.

[83] A. Settle, D. A. Connors, G. Hoflehner, and D. Lavery, "Optimization for the Intel® Itanium® architecture register stack," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*, 2003, pp. 115–124. [Online]. Available: http://rogue.colorado.edu/draco/papers/cgo-03-register.pdf

[84] M. Chrobak and J. Noga, "LRU is better than FIFO," in *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, 1998, pp. 78–81. [Online]. Available: http://www.cs.ucr.edu/~marek/pubs/lru_vs_fifo.ps

[85] G. Slavenburg and M. Janssens, *DataBook: PNX1300 Series Media Processors.* Philips Electronics North America Corporation, Feb 2002, ch. Appendix A. [Online]. Available: http://www.semiconductors.philips.com/acrobat_download/literature/9397/75010145.pdf

[86] *IA-32 Intel® Architecture Software Developer's Manual*, Intel Corporation, Apr 2005. [Online]. Available: ftp://download.intel.com/design/Pentium4/manuals/25366615.pdf, ftp://download.intel.com/design/Pentium4/manuals/25366715.pdf

[87] R. B. Lee, "Multimedia extensions for general-purpose processors," in *IEEE Workshop on Signal Processing Systems*, Nov 1997, pp. 9–23. [Online]. Available: http://www.ee.princeton.edu/~rblee/HPpapers/sips15go.ps

[88] M. L. Anido, A. Paar, and N. Bagherzadeh, "Improving the operation autonomy of SIMD processing elements by using guarded instructions and pseudo branches," in *DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design*, 2002, pp. 148–156.

[89] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, 1994, pp. 120–129. [Online]. Available: http://www.mhl.tuc.gr/research/publications/ISCA1994-GuardedExecution.pdf

[90] *MIPS32™ Architecture For Programmers*, MIPS Technologies Inc, Jun 2003.

[91] (2006, Apr) Transitive corporation: Technology overview. Transitive Corporation. [Online]. Available: http://www.transitive.com/technology.htm

[92] D. Sima, "The design space of register renaming techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70–83, 2000. [Online]. Available: http://www.dc.uba.ar/people/materias/ap/Articulos/The Design Space of Register Renaming Techniques.pdf

[93] G. H. Gonnet, "Balancing binary trees by internal path reduction," *Communications of the ACM*, vol. 26, no. 12, pp. 1074–1081, 1983.

[94] V. Betz and J. Rose, "How much logic should go in an FPGA logic block?" *IEEE Design & Test*, vol. 15, no. 1, pp. 10–15, 1998. [Online]. Available: http://www.eecg.utoronto.ca/~jayar/pubs/betz/design98.pdf

[95] K. Eckl and C. Legl, "Retiming sequential circuits with multiple register classes," in *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, Mar 1999, pp. 650–656.

[96] C. Leiserson and J. Saxe, "Optimizing synchronous systems," in *Journal of VLSI and Computer Systems*, vol. 1, no. 1, 1983, pp. 41–67.

[97] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 473–489, 2004. [Online]. Available: http://athos.rutgers.edu/pub/sigplan92-landi-ryder.ps

[98] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1467–1471, 1994.

[99] D. R. Ditzel, "Transmeta's crusoe: Cool chips for mobile computing," *Hot Chips Symposium*, Aug 2000.

[100] (2005, May) Quartus II software. Altera. [Online]. Available: http://www.altera.com/products/software/products/quartus2/qts-index.html

[101] L. Hansen, "Design performance leaps forward with ISE 7.1i software," *Xcell*, no. 53, pp. 64–66, 2005. [Online]. Available: http://www.xilinx.com/publications/xcellonline/xcell_53/xc_pdf/xc_xcell53.pdf

[102] G. Kane and J. Heinrich, *MIPS RISC architecture*, 2nd ed. Prentice-Hall, Inc., 1992.

[103] S. Rhoads. (2005, June) Plasma - most MIPS I™ opcodes. Opencores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/mips/overview

[104] T. Balph, "LFSR counters implement binary polynomial generators," *EDN*, May 1998. [Online]. Available: http://www.pldworld.com/html/technote/11df_06.pdf

[105] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.

[106] P. J. M. Laarhoven and E. H. L. Aarts, Eds., *Simulated annealing: theory and applications.* Norwell, MA, USA: Kluwer Academic Publishers, 1987.

[107] B. Preneel, A. Biryukov, E. Oswald, B. V. Rompay, L. Granboulan, E. Dottax, S. Murphy, A. Dent, J. White, M. Dichtl, S. Pyka, M. Schafheutle, P. Serf, E. Biham, E. Barkan, O. Dunkelman, J. J. Quisquater, M. Ciet, F. Sica, L. Knudsen, M. Parker, and H. Raddum, *NESSIE security report.* NESSIE, Feb 2003, ch. 3.

[108] E. Çetin, R. C. S. Morling, and L. Kale, "An integrated 256-point complex FFT processor for real-time spectrum analysis and

measurement," *IEEE Proceedings of Instrumentation and Measurement-Technology Conference*, vol. 1, pp. 96–101, May 1997. [Online]. Available: http://dolphin.wmin.ac.uk/~cetine/IMTC97.pdf

[109] L. C. Ludeman, *Fundamentals of digital signal processing.* Wiley, 1996, ch. 6, pp. 272–286.

[110] ——, *Fundamentals of digital signal processing.* Wiley, 1996, ch. 5, pp. 246–252.

[111] D. L. Gall, "MPEG: a video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[112] G. de Haan, "IC for motion-compensated de-interlacing, noise reduction, and picture-rate conversion," *IEEE Transactions Consumer Electronics*, vol. 45, no. 3, pp. 617–624, 1999. [Online]. Available: http://www.semiconductors.philips.com/acrobat_download/other/cms/99_225.pdf

[113] W. A. Martin, "Sorting," *ACM Computing Surveys*, vol. 3, no. 4, pp. 147–174, Dec 1971.

[114] (2005, June) Virtex-4 family overview. Xilinx. [Online]. Available: http://www.xilinx.com/bvdocs/publications/ds112.pdf

[115] (2005, June) Stratix II device family data sheet. Altera. [Online]. Available: http://www.altera.com/literature/hb/stx2/stx2_sii5v1_01.pdf

[116] (2005, Jul) HyperTransport Consortium. [Online]. Available: http://www.hypertransport.org/

[117] (1997) Using the RDTSC instruction for performance monitoring. Intel Corporation. [Online]. Available: http://www.math.uwaterloo.ca/~jamuir/rdtscpm1.pdf

[118] (2005, Aug) The CELL project at IBM research. IBM Corporation. [Online]. Available: http://www.research.ibm.com/cell/

[119] *Intel® Pentium® D Processor 840, 830 and 820 Datasheet*, Intel Corporation, May 2005. [Online]. Available: http://download.intel.com/design/Pentiumd/datashts/30750601.pdf

[120] B. Davis, T. Mudge, B. Jacob, and V. Cuppu, "DDR2 and low latency variants," in *The 27th Annual International Symposium on Computer Architecture*, May 2000.

[121] (2005, Aug) AMD Athlon™ 64. Advanced Micro Devices, Inc. [Online]. Available: http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_9484,00.html

[122] (2005, Aug) Intel® Pentium® 4 processor. Intel Corporation. [Online]. Available: http://www.intel.com/products/processor/pentium4/index.htm

[123] (2005, Aug) Transmeta™ Efficeon™ TM880 processor. Transmeta Corporation. [Online]. Available: http://www.transmeta.com/pdfs/brochures/tmta_efficeon_tm8800.pdf

[124] (2005, Aug) CSX600 application accelerator. ClearSpeed Technology plc. [Online]. Available: http://www.clearspeed.com/products/si.php

Appendix A

MIPS Test Algorithms Source Code

This appendix contains only the kernels of the test algorithms used to evaluate the MIPS platform. The source code for the console application (described in section 5.2.1) is not included.

A.1 PRBS Generator (Standard)

```
void randomFill( unsigned long data[], unsigned long length,
                 unsigned long seed )
{
    int            i, curIndex;
    unsigned long bit31, bit28;
    unsigned long shiftReg;

    shiftReg = (seed == 0) ? DEFAULT_PRBS_SEED : seed;
    // fill the array
    for ( curIndex = 0; curIndex < length; curIndex++ )
    {
        // generate 32 bits of random data
        for ( i = 0; i < 32; i++ )
        {
```

```
            // get the last 2 bits of the register
            bit31 = (shiftReg >> 31) & 0x1;
            bit28 = (shiftReg >> 28) & 0x1;
            // shift the reg up and or in the new bit
            shiftReg <<= 1;
            shiftReg  |= bit31 ^ bit28;
        }
        // store the random number
        data[ curIndex ] = shiftReg;
    }
}
```

A.2 PRBS Generator (Unrolled)

```
void randomFillOpt( unsigned long data[],
                    unsigned long length,
                    unsigned long seed )
{
    int           curIndex;
    unsigned int bit31, bit30, bit29, bit28, bit27, bit26;
    unsigned int bit25, bit24, bit23, bit22, bit21, bit20;
    unsigned int bit19, bit18, bit17, bit16, bit15, bit14;
    unsigned int bit13, bit12, bit11, bit10, bit9,  bit8;
    unsigned int bit7,  bit6,  bit5,  bit4,  bit3,  bit2;
    unsigned int bit1,  bit0;
    unsigned int temp0, temp1, temp2, shiftReg, newShiftReg;

    shiftReg = (seed == 0) ? DEFAULT_PRBS_SEED : seed;
    // fill the array
    for ( curIndex = 0; curIndex < length; curIndex++ )
    {
        // Get the bits of the shift reg into the registers
        bit31 = (shiftReg >> 31) & 0x1;
        bit30 = (shiftReg >> 30) & 0x1;
        bit29 = (shiftReg >> 29) & 0x1;
        bit28 = (shiftReg >> 28) & 0x1;
        bit27 = (shiftReg >> 27) & 0x1;
        bit26 = (shiftReg >> 26) & 0x1;
        bit25 = (shiftReg >> 25) & 0x1;
        bit24 = (shiftReg >> 24) & 0x1;
        bit23 = (shiftReg >> 23) & 0x1;
        bit22 = (shiftReg >> 22) & 0x1;
        bit21 = (shiftReg >> 21) & 0x1;
        bit20 = (shiftReg >> 20) & 0x1;
```

```
bit19 = (shiftReg >> 19) & 0x1;
bit18 = (shiftReg >> 18) & 0x1;
bit17 = (shiftReg >> 17) & 0x1;
bit16 = (shiftReg >> 16) & 0x1;
bit15 = (shiftReg >> 15) & 0x1;
bit14 = (shiftReg >> 14) & 0x1;
bit13 = (shiftReg >> 13) & 0x1;
bit12 = (shiftReg >> 12) & 0x1;
bit11 = (shiftReg >> 11) & 0x1;
bit10 = (shiftReg >> 10) & 0x1;
bit9  = (shiftReg >> 9)  & 0x1;
bit8  = (shiftReg >> 8)  & 0x1;
bit7  = (shiftReg >> 7)  & 0x1;
bit6  = (shiftReg >> 6)  & 0x1;
bit5  = (shiftReg >> 5)  & 0x1;
bit4  = (shiftReg >> 4)  & 0x1;
bit3  = (shiftReg >> 3)  & 0x1;
bit2  = (shiftReg >> 2)  & 0x1;
bit1  = (shiftReg >> 1)  & 0x1;
bit0  = (shiftReg >> 0)  & 0x1;
// Calc the new shift register value
temp2 = bit31 ^ bit28;
temp1 = bit30 ^ bit27;
temp0 = bit29 ^ bit26;
newShiftReg  = (bit0  ^ temp0);
newShiftReg |= (bit1  ^ temp1) << 1;
newShiftReg |= (bit2  ^ temp2) << 2;
newShiftReg |= (bit3  ^ bit0)  << 3;
newShiftReg |= (bit4  ^ bit1)  << 4;
newShiftReg |= (bit5  ^ bit2)  << 5;
newShiftReg |= (bit6  ^ bit3)  << 6;
newShiftReg |= (bit7  ^ bit4)  << 7;
newShiftReg |= (bit8  ^ bit5)  << 8;
newShiftReg |= (bit9  ^ bit6)  << 9;
newShiftReg |= (bit10 ^ bit7)  << 10;
newShiftReg |= (bit11 ^ bit8)  << 11;
newShiftReg |= (bit12 ^ bit9)  << 12;
newShiftReg |= (bit13 ^ bit10) << 13;
newShiftReg |= (bit14 ^ bit11) << 14;
newShiftReg |= (bit15 ^ bit12) << 15;
newShiftReg |= (bit16 ^ bit13) << 16;
newShiftReg |= (bit17 ^ bit14) << 17;
newShiftReg |= (bit18 ^ bit15) << 18;
newShiftReg |= (bit19 ^ bit16) << 19;
newShiftReg |= (bit20 ^ bit17) << 20;
newShiftReg |= (bit21 ^ bit18) << 21;
newShiftReg |= (bit22 ^ bit19) << 22;
```

```
        newShiftReg |= (bit23 ^ bit20) << 23;
        newShiftReg |= (bit24 ^ bit21) << 24;
        newShiftReg |= (bit25 ^ bit22) << 25;
        newShiftReg |= (bit26 ^ bit23) << 26;
        newShiftReg |= (bit27 ^ bit24) << 27;
        newShiftReg |= (bit28 ^ bit25) << 28;
        newShiftReg |= temp0 << 29;
        newShiftReg |= temp1 << 30;
        newShiftReg |= temp2 << 31;
        // rotate vars and store the random number
        shiftReg = newShiftReg;
        data[ curIndex ] = shiftReg;
    }
}
```

A.3 FFT

```
/* Written by:          Tom Roberts     11/8/89
 * Made portable:       Malcolm Slaney  12/15/94
 *                      malcolm@interval.com
 * Embedded MIPS port:  Thomas Grocutt  10/1/2005
 */
int fixedPointFft( volatile unsigned long data[],
                   unsigned long m, bool inverse )
{
    int mr,nn,i,j,l,k, n, scale, temp;
    bool shift;
    short qr,qi,tr,ti,wr,wi;

    mr    = 0;
    scale = 0;
    n     = 1 << m;
    nn    = n - 1;
    // check that the fft isn't to bit to process
    if ( n > N_WAVE ) return -1;

    // decimation in time - re-order data
    for ( m = 1; m <= nn; m++ )
    {
        l = n;
        do
        {
            l >>= 1;
        } while ( (mr + l) > nn );
```

```
        mr = ( mr & (l - 1) ) + l;
        // make sure elements are in the right order
        if ( mr > m )
        {
            temp        = data[ m ];
            data[ m ]   = data[ mr ];
            data[ mr ]  = temp;
        }
    }

    k = LOG2_N_WAVE - 1;
    for ( l = 1; l < n; l <<= 1 )
    {
        if ( inverse )
        {
            // variable scaling, depending upon data
            shift = false;
            for ( i = 0; i < n; i++ )
            {
                temp = data[ i ];
                tr   = temp & 0xFFFF;
                ti   = temp >> 16;
                if ( tr < 0 ) tr = -tr;
                if ( ti < 0 ) ti = -ti;
                if ( (tr > 16383) || (ti > 16383) )
                {
                    shift = true;
                    i     = n;
                }
            }
            if ( shift ) scale++;
        }
        else
        {
            /* fixed scaling, for proper normalization.
             * There will be log2(n) passes, so this results
             * in an overall factor of 1/n, distributed to
             * maximize arithmetic accuracy.
             */
            shift = true;
        }

        /* are we on final butteffly where there is only 1
         * pass of inner loop
         */
        if ( (l << 1) < n)
        {
```

```
        /* it may not be obvious, but the shift will be
         * performed on each data point exactly once,
         * during this pass.
         */
        for ( m = 0; m < l; m++ )
        {
            j  = m << k;
            wr =  sinLut[ j + (N_WAVE / 4) ];
            wi = -sinLut[ j ];
            if ( inverse ) wi = -wi;
            if ( shift   )
            {
                wr >>= 1;
                wi >>= 1;
            }

            for ( i = m; i < n; i += l << 1 )
            {
                j = i + l;
                temp = data[ j ];
                qr   = temp & 0xFFFF;
                qi   = temp >> 16;
                tr   = FIX_MUL( wr, qr )
                     - FIX_MUL( wi, qi );
                ti   = FIX_MUL( wr, qi )
                     + FIX_MUL( wi, qr );
                temp = data[ i ];
                qr   = temp & 0xFFFF;
                qi   = temp >> 16;
                if ( shift )
                {
                    qr >>= 1;
                    qi >>= 1;
                }
                data[ j ] = ((qi - ti) << 16)
                          | ((qr - tr) & 0xFFFF);
                data[ i ] = ((qi + ti) << 16)
                          | ((qr + tr) & 0xFFFF);
            }
        }
    }
    else
    {
        /* this is exactly the same as above, except its
         * the special case on the last butterfly when
         * the inner loop only does 1 iteration.
         * Removing this loop reducing the overhead and
```

```
            * speeds things up.
            */
            for ( m = 0; m < l; m++ )
            {
                j  = m << k;
                wr =  sinLut[ j + (N_WAVE / 4) ];
                wi = -sinLut[ j ];
                if ( inverse ) wi = -wi;
                if ( shift   )
                {
                    wr >>= 1;
                    wi >>= 1;
                }

                j = m + l;
                temp = data[ j ];
                qr   = temp & 0xFFFF;
                qi   = temp >> 16;
                tr   = FIX_MUL( wr, qr )
                     - FIX_MUL( wi, qi );
                ti   = FIX_MUL( wr, qi )
                     + FIX_MUL( wi, qr );
                temp = data[ m ];
                qr   = temp & 0xFFFF;
                qi   = temp >> 16;
                if ( shift )
                {
                    qr >>= 1;
                    qi >>= 1;
                }
                data[ j ] = ((qi - ti) << 16)
                          | ((qr - tr) & 0xFFFF);
                data[ m ] = ((qi + ti) << 16)
                          | ((qr + tr) & 0xFFFF);
            }
        }
        k--;
    }
    return ( scale );
}
```

A.4 Low Pass Filter

```
void lpfAudio( volatile unsigned int buffer[],
```

```
                unsigned int length
            )
{
    int             i;
    unsigned int    temp;
    short           c0;
    short           c0Cur;
    short           c0Prev1;
    short           c0Prev2;
    int             c0Prev3_mul2;
    int             c0Prev4_mul2;
    int             c0Prev5_mul2;
    short           c1;
    short           c1Cur;
    short           c1Prev1;
    short           c1Prev2;
    int             c1Prev3_mul2;
    int             c1Prev4_mul2;
    int             c1Prev5_mul2;

    // pre calc some temp vars
    c0Cur           = buffer[ 0 ] & 0xFFFF;
    c1Cur           = buffer[ 0 ] >> 16;
    c0Prev1         = c0Cur;
    c0Prev2         = c0Cur;
    c0Prev3_mul2    = c0Cur * 2;
    c0Prev4_mul2    = c0Cur * 2;
    c0Prev5_mul2    = c0Cur * 2;
    c1Prev1         = c1Cur;
    c1Prev2         = c1Cur;
    c1Prev3_mul2    = c1Cur * 2;
    c1Prev4_mul2    = c1Cur * 2;
    c1Prev5_mul2    = c1Cur * 2;
    // go through the buffer low pass filtering each channel
    for ( i = 0; i < length; i++ )
    {
        // get the value of the 2 channels
        temp  = buffer[ i ];
        c0Cur = temp & 0xFFFF;
        c1Cur = temp >> 16;
        // calc the lpf output
        c0 = ( (c0Cur   * 4) + (c0Prev1 * 3) +
               (c0Prev2 * 3) +  c0Prev3_mul2 +
                c0Prev4_mul2 +  c0Prev5_mul2 ) / 16;
        c1 = ( (c1Cur   * 4) + (c1Prev1 * 3) +
               (c1Prev2 * 3) +  c1Prev3_mul2 +
                c1Prev4_mul2 +  c1Prev5_mul2 ) / 16;
```

```
        // rotate valus
        c0Prev5_mul2 = c0Prev4_mul2;
        c0Prev4_mul2 = c0Prev3_mul2;
        c0Prev3_mul2 = c0Prev2 * 2;
        c0Prev2      = c0Prev1;
        c0Prev1      = c0Cur;
        c1Prev5_mul2 = c1Prev4_mul2;
        c1Prev4_mul2 = c1Prev3_mul2;
        c1Prev3_mul2 = c1Prev2 * 2;
        c1Prev2      = c1Prev1;
        c1Prev1      = c1Cur;
        // store lpf result
        buffer[ i ] = (c1 << 16) | (c0 & 0xFFFF);
    }
}
```

A.5 Normalization

```
void normaliseAudio( volatile unsigned int buffer[],
                             unsigned int length )
{
    int          i;
    short        c0Val;
    short        c0Min;
    short        c0Max;
    short        c1Val;
    short        c1Min;
    short        c1Max;
    short        absMax;
    int          upScale;
    unsigned int temp;

    /* go through the buffer getting the min and max
     *  values of each channel
     */
    c0Max = buffer[ 0 ] & 0xFFFF;
    c0Min = c0Max;
    c1Max = buffer[ 0 ] >> 16;
    c1Min = c1Max;
    for ( i = 0; i < length; i++ )
    {
        // get the value of the 2 channels
        temp  = buffer[ i ];
        c0Val = temp &  0xFFFF;
```

```
        c1Val = temp >> 16;
        // update min and max vals
        if ( c0Val > c0Max ) c0Max = c0Val;
        if ( c0Val < c0Min ) c0Min = c0Val;
        if ( c1Val > c1Max ) c1Max = c1Val;
        if ( c1Val < c1Min ) c1Min = c1Val;
    }

    // find the max absolute value and calc the scale factor
    absMax = (c0Min < 0) ? (-1 * c0Min) : c0Min;
    i      = (c0Max < 0) ? (-1 * c0Max) : c0Max;
    if ( i > absMax ) absMax = i;
    i      = (c1Min < 0) ? (-1 * c1Min) : c1Min;
    if ( i > absMax ) absMax = i;
    i      = (c1Max < 0) ? (-1 * c1Max) : c1Max;
    if ( i > absMax ) absMax = i;
    upScale = (MAX_VALUE * MAX_VALUE) / absMax;

    // now apply the correction
    for ( i = 0; i < length; i++ )
    {
        temp  = buffer[ i ];
        c0Val = temp & 0xFFFF;
        c1Val = temp >> 16;
        // get the value of the 2 channels
        c0Val = ( c0Val * upScale ) >> 15;
        c1Val = ( c1Val * upScale ) >> 15;
        buffer[ i ] = (c1Val << 16) | (c0Val & 0xFFFF);
    }
}
```

A.6 Block Search (Planar)

```
static int findBlockPlanar( IMAGE_PROC_FrameBuffer *src,  IMAGE_PROC_Point *macroBlock,
                            IMAGE_PROC_FrameBuffer *dest, IMAGE_PROC_Point *location
                          )
{
    int           sad;
    int           curPix;
    int           curLine;
    int           curBlockLine;
    int           dif0, dif1, dif2,  dif3,  dif4,  dif5,  dif6,  dif7;
    int           dif8, dif9, dif10, dif11, dif12, dif13, dif14, dif15;
    unsigned int  vals0A, vals1A, vals2A, vals3A;
    unsigned int  vals0B, vals1B, vals2B, vals3B;
    unsigned int  line1;
    unsigned int  line2;
    unsigned char *curBlock;
    unsigned char *macroBlockAddr;
    int           lowestSad       = -1;
    unsigned char *closestBlock   = NULL;

    // calc the address of the macroblock to search
    macroBlockAddr = (unsigned char *) &src->baseAddr[ (macroBlock->y * src->stride)
                                                       + (src->width / 4) ];
    macroBlockAddr = &macroBlockAddr[ macroBlock->x ];
    // go through the lines and pixels in the image
    for ( curLine = 0; curLine < (dest->height - MACRO_BLOCK_HEIGHT); curLine++ )
    {
```

```
for ( curPix = 0; curPix < dest->width - MACRO_BLOCK_WIDTH; curPix++ )
{
    curBlock = (unsigned char *) &dest->baseAddr[ (curLine * dest->stride)
                                                  + (dest->width / 4) ];
    curBlock = &curBlock[ curPix ];
    // go through the 2 blocks calculating the sum of absolute differences
    sad      = 0;
    for ( curBlockLine = 0; curBlockLine < MACRO_BLOCK_HEIGHT; curBlockLine += 1 )
    {
        line1 = (unsigned int) &curBlock[        curBlockLine * (dest->stride * 4) ];
        line2 = (unsigned int) &macroBlockAddr[ curBlockLine * (dest->stride * 4) ];
        // get the values
        MISS_LOAD( line1,       vals0A );
        MISS_LOAD( line2,       vals0B );
        MISS_LOAD( line1 + 4,   vals1A );
        MISS_LOAD( line2 + 4,   vals1B );
        MISS_LOAD( line1 + 8,   vals2A );
        MISS_LOAD( line2 + 8,   vals2B );
        MISS_LOAD( line1 + 12,  vals3A );
        MISS_LOAD( line2 + 12,  vals3B );
        // calc the differences
        dif0  = ( (vals0A & 0xFF000000) - (vals0B & 0xFF000000) ) >> 24;
        dif1  = ( (vals0A & 0x00FF0000) - (vals0B & 0x00FF0000) ) >> 16;
        dif2  = ( (vals0A & 0x0000FF00) - (vals0B & 0x0000FF00) ) >> 8;
        dif3  = ( (vals0A & 0x000000FF) - (vals0B & 0x000000FF) );
        dif4  = ( (vals1A & 0xFF000000) - (vals1B & 0xFF000000) ) >> 24;
        dif5  = ( (vals1A & 0x00FF0000) - (vals1B & 0x00FF0000) ) >> 16;
        dif6  = ( (vals1A & 0x0000FF00) - (vals1B & 0x0000FF00) ) >> 8;
        dif7  = ( (vals1A & 0x000000FF) - (vals1B & 0x000000FF) );
```

```
            dif8  = ( (vals2A & 0xFF000000) - (vals2B & 0xFF000000) ) >> 24;
            dif9  = ( (vals2A & 0x00FF0000) - (vals2B & 0x00FF0000) ) >> 16;
            dif10 = ( (vals2A & 0x0000FF00) - (vals2B & 0x0000FF00) ) >> 8;
            dif11 = ( (vals2A & 0x000000FF) - (vals2B & 0x000000FF) );
            dif12 = ( (vals3A & 0xFF000000) - (vals3B & 0xFF000000) ) >> 24;
            dif13 = ( (vals3A & 0x00FF0000) - (vals3B & 0x00FF0000) ) >> 16;
            dif14 = ( (vals3A & 0x0000FF00) - (vals3B & 0x0000FF00) ) >> 8;
            dif15 = ( (vals3A & 0x000000FF) - (vals3B & 0x000000FF) );
            // now sum the absolute differences
            sad  += ( (dif0  >= 0) ? dif0  : -dif0  ) + ( (dif1  >= 0) ? dif1  : -dif1  ) +
                    ( (dif2  >= 0) ? dif2  : -dif2  ) + ( (dif3  >= 0) ? dif3  : -dif3  ) +
                    ( (dif4  >= 0) ? dif4  : -dif4  ) + ( (dif5  >= 0) ? dif5  : -dif5  ) +
                    ( (dif6  >= 0) ? dif6  : -dif6  ) + ( (dif7  >= 0) ? dif7  : -dif7  ) +
                    ( (dif8  >= 0) ? dif8  : -dif8  ) + ( (dif9  >= 0) ? dif9  : -dif9  ) +
                    ( (dif10 >= 0) ? dif10 : -dif10 ) + ( (dif11 >= 0) ? dif11 : -dif11 ) +
                    ( (dif12 >= 0) ? dif12 : -dif12 ) + ( (dif13 >= 0) ? dif13 : -dif13 ) +
                    ( (dif14 >= 0) ? dif14 : -dif14 ) + ( (dif15 >= 0) ? dif15 : -dif15 );

        }
        // is this block a better match than the previous best match
        if ( (sad <= lowestSad) || (closestBlock == NULL) )
        {
            lowestSad    = sad;
            closestBlock = curBlock;
        }
    }
}
// calc the location of the block from its address
curPix      = ( (int) closestBlock ) - ( (int) dest->baseAddr ) - dest->width;
```

```
    location->x = curPix % (dest->stride * 4);
    location->y = curPix / (dest->stride * 4);
    return ( lowestSad );
}
```

A.7 Block Search (Packed)

```
static int findBlockPacked( IMAGE_PROC_FrameBuffer *src,  IMAGE_PROC_Point *macroBlock,
                            IMAGE_PROC_FrameBuffer *dest, IMAGE_PROC_Point *location
                          )
{
    int           sad;
    int           dif0, dif1, dif2,  dif3,  dif4,  dif5,  dif6,  dif7;
    int           dif8, dif9, dif10, dif11, dif12, dif13, dif14, dif15;
    int           curPix;
    int           curLine;
    int           curBlockLine;
    unsigned int  *curBlock;
    unsigned char *line1;
    unsigned char *line2;
    unsigned int  *macroBlockAddr;
    int           lowestSad       = -1;
    unsigned int  *closestBlock   = NULL;

    // calc the address of the macroblock to search
    macroBlockAddr = &( src->baseAddr[ (macroBlock->y * src->stride) + macroBlock->x ] );
```

```
// go through the lines and pixels in the image
for ( curLine = 0; curLine < (dest->height - MACRO_BLOCK_HEIGHT); curLine++ )
{
    for ( curPix = 0; curPix < (dest->width - MACRO_BLOCK_WIDTH); curPix++ )
    {
        curBlock = &( dest->baseAddr[ (curLine * dest->stride) + curPix ] );
        // go through the 2 blocks calculating the sum of absolute differences
        sad      = 0;
        for ( curBlockLine = 0; curBlockLine < MACRO_BLOCK_HEIGHT; curBlockLine += 1 )
        {
            line1 = (unsigned char *) &curBlock[       curBlockLine * dest->stride ];
            line2 = (unsigned char *) &macroBlockAddr[ curBlockLine * dest->stride ];
            // calc sum of the abs difs for all the pixels in the line in the block
            dif0  = line1[ 1 + (4 *  0) ] - line2[ 1 + (4 *  0) ];
            dif1  = line1[ 1 + (4 *  1) ] - line2[ 1 + (4 *  1) ];
            dif2  = line1[ 1 + (4 *  2) ] - line2[ 1 + (4 *  2) ];
            dif3  = line1[ 1 + (4 *  3) ] - line2[ 1 + (4 *  3) ];
            dif4  = line1[ 1 + (4 *  4) ] - line2[ 1 + (4 *  4) ];
            dif5  = line1[ 1 + (4 *  5) ] - line2[ 1 + (4 *  5) ];
            dif6  = line1[ 1 + (4 *  6) ] - line2[ 1 + (4 *  6) ];
            dif7  = line1[ 1 + (4 *  7) ] - line2[ 1 + (4 *  7) ];
            dif8  = line1[ 1 + (4 *  8) ] - line2[ 1 + (4 *  8) ];
            dif9  = line1[ 1 + (4 *  9) ] - line2[ 1 + (4 *  9) ];
            dif10 = line1[ 1 + (4 * 10) ] - line2[ 1 + (4 * 10) ];
            dif11 = line1[ 1 + (4 * 11) ] - line2[ 1 + (4 * 11) ];
            dif12 = line1[ 1 + (4 * 12) ] - line2[ 1 + (4 * 12) ];
            dif13 = line1[ 1 + (4 * 13) ] - line2[ 1 + (4 * 13) ];
            dif14 = line1[ 1 + (4 * 14) ] - line2[ 1 + (4 * 14) ];
            dif15 = line1[ 1 + (4 * 15) ] - line2[ 1 + (4 * 15) ];
```

```
                sad  += ( (dif0  >= 0) ? dif0  : -dif0  ) + ( (dif1  >= 0) ? dif1  : -dif1  ) +
                        ( (dif2  >= 0) ? dif2  : -dif2  ) + ( (dif3  >= 0) ? dif3  : -dif3  ) +
                        ( (dif4  >= 0) ? dif4  : -dif4  ) + ( (dif5  >= 0) ? dif5  : -dif5  ) +
                        ( (dif6  >= 0) ? dif6  : -dif6  ) + ( (dif7  >= 0) ? dif7  : -dif7  ) +
                        ( (dif8  >= 0) ? dif8  : -dif8  ) + ( (dif9  >= 0) ? dif9  : -dif9  ) +
                        ( (dif10 >= 0) ? dif10 : -dif10 ) + ( (dif11 >= 0) ? dif11 : -dif11 ) +
                        ( (dif12 >= 0) ? dif12 : -dif12 ) + ( (dif13 >= 0) ? dif13 : -dif13 ) +
                        ( (dif14 >= 0) ? dif14 : -dif14 ) + ( (dif15 >= 0) ? dif15 : -dif15 );

            }
            // is this block a better match than the previous best match
            if ( (sad <= lowestSad) || (closestBlock == NULL) )
            {
                lowestSad    = sad;
                closestBlock = curBlock;
            }
        }
    }
    // calc the location of the block from its address
    curPix      = ( ( (int) closestBlock ) - ( (int) dest->baseAddr ) ) >> 2;
    location->x = curPix % dest->stride;
    location->y = curPix / dest->stride;
    return ( lowestSad );
}
```

A.8 Mandelbrot

```
void mandelbrot( IMAGE_PROC_FrameBuffer *fb )
{
   int Zr, Zi;
   int Cr, Ci;
   int x, y;
   int modulous;
   int i;
   int tempR, tempI;

   // iterate over the lines and the pixels on the lines
   for ( y = 0; y < 240; y++ )
   {
      for ( x = 0; x < 320; x++ )
      {
         // scale pixels so they are in the complex plane
         Cr = ((x - X_OFFSET) << FIX_POINT_SCALE) / X_SCALE;
         Ci = ((y - Y_OFFSET) << FIX_POINT_SCALE) / Y_SCALE;
         // iterate over the z value
         Zr = 0;
         Zi = 0;
         modulous = 0;
         for ( i = 0; (i < MAX_ITERATIONS) &&
                      (modulous < MOD_THRESHOLD); i++ )
         {
            // calc the mod and for early exit
            modulous = Zr * Zr;
            R_DIV( modulous, FIX_POINT_SCALE );
            modulous += Zi + Zi;
            // calc the square of Z
            tempR = (Zr * Zr) - (Zi * Zi);
            R_DIV( tempR, FIX_POINT_SCALE );
            tempI = 2 * Zr * Zi;
            R_DIV( tempI, FIX_POINT_SCALE );
            // add C to get new Z value
            Zr = tempR + Cr;
            Zi = tempI + Ci;
         }
         // calc the modulous of z |(a+ib)|=(a2+b2)1/2
         i = (modulous < MOD_THRESHOLD) ?
             0xFFFFFF : (i * 15);
         fb->baseAddr[ (y * fb->stride) + x ] = i;
      }
   }
}
```

A.9 Half Brightness

```
void halfBrightness( IMAGE_PROC_FrameBuffer *fb )
{
   int r, g, b, y, u, v;
   int curLine, curPixel, width;
   unsigned int value, *line;

   width = fb->width;
   // go through the lines in the picture
   for ( curLine = 0; curLine < fb->height; curLine++ )
   {
      line = fb->baseAddr + (curLine * fb->stride);
      // go through the pixels on the line
      for ( curPixel = 0; curPixel < width; curPixel++ )
      {
         // unpack the RGB value and convent to YUV
         value = line[ curPixel ];
         b =  value & 0xFF;
         g = (value >>  8) & 0xFF;
         r = (value >> 16) & 0xFF;
         y = ( (Y_R_CONST * r) + (Y_G_CONST * g) +
               (Y_B_CONST * b) ) >> 15;
         u = ( U_CONST * (b - y) ) >> 15;
         v = ( V_CONST * (r - y) ) >> 15;
         // half the brightness
         y >>= 1;
         // convert back to RGB, clip to range and repack
         r = ( (R_CONST * v) >> 15 ) + y;
         g = y - ( ( (G_V_CONST * v) +
                     (G_U_CONST * u) ) >> 15 );
         b = ( (B_CONST * u) >> 15 ) + y;
         if ( r > 255 ) value = 255;
         if ( r <   0 ) value = 0;
         if ( g > 255 ) value = 255;
         if ( g <   0 ) value = 0;
         if ( b > 255 ) value = 255;
         if ( b <   0 ) value = 0;
         value  = r << 16;
         value |= g << 8;
         value |= b;
         line[ curPixel ] = value;
      }
   }
}
```

A.10 Factorial and Series Sum

```
static void fact( const CONSOLE_GeneralCmdType *cmd,
                  void *params[] )
{
    int curVal;
    int fact;

    curVal = (int) params[0];
    fact   = (curVal == 0) ? 0 : 1;
    // calc the factorial itself
    for ( ; curVal > 0; curVal-- ) fact *= curVal;
    // print out the results
    printNum( fact );
    putString( "\n" );
}

static void seriesSum( const CONSOLE_GeneralCmdType *cmd,
                       void *params[] )
{
    int curVal;
    int sum    = 0;

    // calc the factorial itself
    for ( curVal = (int) params[0]; curVal > 0; curVal-- )
    {
        sum += curVal;
    }
    // print out the results
    printNum( sum );
    putString( "\n" );
}
```

A.11 Copy

```
static void copyFunc( const CONSOLE_GeneralCmdType *cmd,
                      void *params[] )
{
    unsigned long *srcAddr;
    unsigned long *destAddr;
    unsigned long len;
    unsigned long i;
```

```
    /* extract the data from the params array
     * and mask the addresses
     */
    srcAddr  = (unsigned long *) (( (unsigned long)
                 params[0] ) & WORD_ADDRESS_MASK);
    destAddr = (unsigned long *) (( (unsigned long)
                 params[1] ) & WORD_ADDRESS_MASK);
    len      = ( (unsigned long) params[2] ) >> 2;
    // do the memory copy
    for ( i = 0; i < len; i++ )
    {
        destAddr[ i ] = srcAddr[ i ];
    }
}
```

A.12 Sort

```
/*
 * @(#)BubbleSortAlgorithm.java 1.6 95/01/31 James Gosling
 *
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights
 * Reserved
 *
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for NON-COMMERCIAL
 * purposes and without fee is hereby granted provided that
 * this copyright notice appears in all copies. Please refer
 * to the file "copyright.html" for further important
 * copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE
 * SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED,
 * INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
/**
 * A bubble sort demonstration algorithm
 * SortAlgorithm.java, Thu Oct 27 10:32:35 1994
 *
 * @author James Gosling
 * @version      1.6, 31 Jan 1995
```

```
 *
 * Modified 23 Jun 1995 by Jason Harrison@cs.ubc.ca:
 *   Algorithm completes early when no items have been
 *   swapped in the last pass.
 * Modified 9-9-2004 by Thomas Grocutt
 *   Minor optimizations and ported to C
 */
static void bubbleSort( unsigned long data[],
                        unsigned long length
                      )
{
    int           i;
    int           j;
    unsigned long dataTemp;
    bool          complete  = false;

    // keep sweep until reach end or we completed early
    for ( i = length - 1; (i >= 0) && !complete; i-- )
    {
        complete = true;
        dataTemp = data[ 0 ];
        // sweep the array swapping out of order pairs
        for ( j = 0; j < i; j++ )
        {
            // should the 2 elements be swapped
            if ( dataTemp <= data[ j + 1 ] )
            {
                dataTemp = data[ j + 1 ];
            }
            else
            {
                data[ j ]     = data[ j + 1 ];
                data[ j + 1 ] = dataTemp;
                complete      = false;
            }
        }
    }
}

/*
 * @(#)HeapSortAlgorithm.java    1.0 95/06/23 Jason Harrison
 *
 * Copyright (c) 1995 University of British Columbia
 *
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for NON-COMMERCIAL
 * purposes and without fee is hereby granted provided that
```

```
 * this copyright notice appears in all copies. Please refer
 * to the file "copyright.html" for further important
 * copyright and licensing information.
 *
 * UBC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE
 * SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED,
 * INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT. UBC SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
/**
 * A heap sort demonstration algorithm
 * SortAlgorithm.java, Thu Oct 27 10:32:35 1994
 *
 * @author Jason Harrison@cs.ubc.ca
 * @version     1.0, 23 Jun 1995
 *
 * Modified 9-9-2004 by Thomas Grocutt
 *   Minor optimizations and ported to C
 */
static void heapSort( unsigned long data[],
                      unsigned long length
                    )
{
    unsigned long i;
    unsigned long temp;
    unsigned long curElement;

    for ( i = length / 2; i > 0; i-- )
    {
        heapSortDownHeap( data, i, length );
    }

    for ( curElement = length - 1; curElement > 0;
          curElement--
        )
    {
        temp                = data[ 0 ];
        data[ 0 ]           = data[ curElement ];
        data[ curElement ] = temp;
        heapSortDownHeap( data, 1, curElement );
    }
}

static void heapSortDownHeap( unsigned long data[],
```

```
                                    unsigned long k,
                                    unsigned long curElement
                                )
{
    unsigned long temp;
    unsigned long j;
    bool          done = false;

    temp = data[ k - 1 ];
    while ( (k <= curElement/2) && !done )
    {
        j = k + k;
        if ( (j < curElement) && (data[j-1] < data[j]) )
        {
            j++;
        }
        if ( temp >= data[j-1] ) done = true;
        else
        {
            data[k-1] = data[j-1];
            k = j;
        }
    }
    data[ k - 1 ] = temp;
}

/*
 * @(#)QSortAlgorithm.java  1.6f 95/01/31 James Gosling
 *
 * Copyright (c) 1994-1995 Sun Microsystems, Inc. All Rights
 * Reserved.
 *
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for NON-COMMERCIAL or
 * COMMERCIAL purposes and without fee is hereby granted.
 * Please refer to the file
 * http://java.sun.com/copy_trademarks.html for further
 * important copyright and trademark information and to
 * http://java.sun.com/licensing.html for further important
 * licensing information for the Java (tm) Technology.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE
 * SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED,
 * INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
 * NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
```

```
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 *
 * THIS SOFTWARE IS NOT DESIGNED OR INTENDED FOR USE OR
 * RESALE AS ON-LINE CONTROL EQUIPMENT IN HAZARDOUS
 * ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN
 * THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION
 * OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT
 * LIFE SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE
 * FAILURE OF THE SOFTWARE COULD LEAD DIRECTLY TO DEATH,
 * PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL
 * DAMAGE ("HIGH RISK ACTIVITIES"). SUN SPECIFICALLY
 * DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR
 * HIGH RISK ACTIVITIES.
 */
/**
 * A quick sort demonstration algorithm
 * SortAlgorithm.java, Thu Oct 27 10:32:35 1994
 *
 * @author James Gosling
 * @version      1.6f, 31 Jan 1995
 *
 * 19 Feb 1996: Fixed to avoid infinite loop discoved by
 *              Paul Haeberli. Misbehaviour expressed when
 *              the pivot element was not unique.
 *              -Jason Harrison
 *
 * 21 Jun 1996: Modified code based on comments from Paul
 *              Haeberli, and Peter Schweizer
 *              (Peter.Schweizer@mni.fh-giessen.de). Used
 *              Daeron Meyer's (daeron@geom.umn.edu) code
 *              for the new pivoting code.
 *              - Jason Harrison
 *
 * 09 Jan 1998: Another set of bug fixes by Thomas Everth
 *              (everth@wave.co.nz) and John Brzustowski
 *              (jbrzusto@gpu.srv.ualberta.ca).
 * 9-9-2004     Modified  by Thomas Grocutt
 *              Minor optimizations and ported to C
 */
static void quickSort( unsigned long data[],
                       unsigned long lo0,
                       unsigned long hi0 )
{
    unsigned long lo;
    unsigned long hi;
    unsigned long temp;
    unsigned long pivot;
```

```
// set initial values
lo = lo0;
hi = hi0;

if      ( lo >= hi      ) return;
else if ( lo == hi - 1 )
{
    // sort a two element list by swapping if necessary
    if ( data[lo] > data[hi] )
    {
        temp        = data[ lo ];
        data[ lo ] = data[ hi ];
        data[ hi ] = temp;
    }
    return;
}

// Pick a pivot and move it out of the way
pivot                   = data[ (lo + hi) / 2 ];
data[ (lo + hi) / 2 ] = data[ hi ];
data[ hi ]              = pivot;
while ( lo < hi )
{
    /*
     * Search forward from a[lo] until an element is
     * found that is greater than the pivot or lo >= hi
     */
    while ( (data[ lo ] <= pivot) && (lo < hi) ) lo++;
    /*
     * Search backward from a[hi] until element is
     * found that is less than the pivot, or lo >= hi
     */
    while ( (pivot <= data[ hi ]) && (lo < hi) ) hi--;
    // Swap elements a[lo] and a[hi]
    if ( lo < hi )
    {
        temp        = data[ lo ];
        data[ lo ] = data[ hi ];
        data[ hi ] = temp;
    }
}

// Put the median in the "center" of the list
data[ hi0 ] = data[ hi ];
data[ hi ]  = pivot;
```

```
    /*
     * Recursive calls, elements a[lo0] to a[lo-1] are
     * less than or equal to pivot, elements a[hi+1]
     * to a[hi0] are greater than pivot.
     */
    quickSort( data, lo0, lo-1 );
    quickSort( data, hi+1, hi0 );
}
```