Micro-Threading And FPGA Implementation Of A RISC Microprocessor

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Science

in

Computer Science

at Massey University, Palmerston North,

New Zealand.

Firas Al-Ali

2007

In loving memory of my mother, the late Zahra Ridha Witwit

MICRO-THREADING AND FPGA IMPLEMENTATION OF A RISC MICROPROCESSOR

Firas Al-Ali

Institute of Information Sciences and Technology

Massey University, Palmerston North

NEW ZEALAND

Submitted:

June 2007

Abstract

This thesis is the outcome of research in two areas of computer technology: microprocessor and multi-processor architectures (specifically from the perspective of how differently they tolerate highly-latent and non-deterministic events), and hardware design of complex digital systems containing both datapath and control (particularly microprocessors).

This thesis starts by pointing out that in order to achieve high processing speeds, current popular superscalar microprocessors (e.g. Intel Pentiums, Digital Alpha, etc) rely heavily on the technique of speculating the outcome of instruction flow in order to predict the behaviour of non-deterministic computing operations (as in loading operands from high-latency memory into the processor). This is fine only if the speculation is correct. But, what if it isn't? If the speculation fails, this would mean that the processor has to abandon its current decision (which now proved to be the wrong one) for the instruction flow path taken and to start all over again with the other path (the actual correct one). This is a waste of valuable processing time and hardware resources and a reduction of performance when speculation fails. Therefore, these processors can achieve high performance only when the majority of speculations are successful (being able to predict the right path).

In an attempt to overcome the above shortcomings, the first part of this thesis is an investigation of the novel vector micro-threading architecture as an alternative approach to the current superscalar-based speculative microprocessor designs. Micro-threading is based on the not-so-novel multithreading technique, which avoids speculation altogether and instead, starts running a different thread of instructions while waiting for the non-determinism to be resolved. This utilizes the chip resources more efficiently without waste of any processing power.

The rest of this thesis focuses on the baseline RISC processor platform, the MIPS R2000, which is reviewed first then partially synthesized from the RTL (Register Transfer Level) description using VHDL and then simulated and tested. This is conducted in order for future research to build upon and add the micro-threading architectural add-ons and modifications.

Keywords:

Micro-threading, Latency Tolerance, FPGA Synthesis, RISC Architecture, MIPS R2000 processor, VHDL.

Acknowledgements

This thesis is the final result of the research efforts and considerable time investment of seemingly only one person; that would be myself! However, this thesis (and its preceding background research work) would not have been completed (or even started) without the help of the following very special people in my research career and my life:

- First and foremost, my enormously wonderful supervisor, Dr. Roger Browne. His technical knowledge, sound advice, and endless patience were all paramount factors in helping me successfully complete this project. However, his role in this research and thesis was far more than just being a supervisor to me, but more of a father figure too with his warm kindness and sincere sympathy towards the personal circumstances I endured during the course of this project.

- My co-supervisor, Mr. Chris Norrie, has been extremely helpful with valuable applied and practical hints and tips from his years of experience in the microprocessor design industry. His thorough insight into this exciting field along with his relentless passion in his career designing microprocessors and VLSI chips, have both spilled over into my research and thereby positively triggering my inspiration, excitement, and technical mastery over the obstacles I encountered during the course of my research.

- During the course of this project, the Institute of Information Sciences and Technology, where I work, lived through the eras of two consecutive heads of institute. Prof. Bob Hodgson was head of institute when I started this research and also started my academic career at Massey University. He provided me with the needed momentum to start this project and continued to support me throughout. Also, his adorable wit and memorable light heartedness and enjoyable sense of humour have all helped me through many a dark day. Now, Prof. Janina Mazierska is head of institute and she provided me with considerable support and help to finish it! I owe it to both of these extremely incredible role models that I had the inspiration to start, and then successfully complete this undertaking!

- The support and guidance I received from A/P Ray Kemp and A/P Elizabeth Kemp was phenomenal. They both had a huge impact on the successful completion of this research and thesis.

- Completing a research thesis is also a personal effort involving one's family as well. Special thanks, with love, goes out to all of them: my loving father Dr. Muneer Al-Ali and my beloved late mother Zahra Witwit, without both of whom, this thesis never would've started in the first place. However, now that my mom is no longer with us, her loving and caring sister Widad Witwit, now my step mom (and second mom as I like to call her), alongside my father, are both witnessing this successful completion at the end.

- The support and love from my sweet brother Wisam Al-Ali, my cute sister Asil Al-Ali, and my loving step-brother Ammar Al-Eid, were catalyst factors in helping me through this period.

- And last but not least, not to forget all those wonderful people in my life who are my very good and close friends. They were all of great support and help during this period.

To all those special and wonderful people I mentioned above, I say from the bottom of my heart: Thank you. Thank you very much. I'm very proud to dedicate this thesis to all of you, with appreciation and love.

Table Of Contents

❑ Companion CD ----------

Publications

Publications prepared during the course of the research for this thesis are as follows:

1. F M Al-Ali and C R Jesshope, 2000, *Survey of High-latency Tolerance in Contemporary Microprocessor Architectures*, Proc. 7th Annual New Zealand Engineering and Technology Postgraduate Conference, pp339-346, ISBN 0-473-07224-6, Massey University, Palmerston North, New Zealand, 23rd & 24th Nov.

2. F M Al-Ali and C R Jesshope, 2001, *Survey of High-latency Tolerance in Future Microprocessor Architectures*, Proc. New Zealand Computer Science research Students' Conference (NZCSRSC 2001), pp86-97, TR-COSC 02/01, University of Canterbury, Christchurch, New Zealand, 19th & 20th April.

3. F M Al-Ali and R F Browne, *Behavioural VHDL Model of a Vector Micro-threading Chip-multiprocessor,* Proceedings of the 6th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC Asia 2002), Bangalore, 16-19 December 2002, Vol.2, pp 518-521, Tata McGraw-Hill Publishing Company Ltd., New Delhi, India, ISBN 0-07-049992-6.

4. F. M. Al-Ali and R. F. Browne, *An FPGA Implementation of a RISC Microprocessor*, Proceedings of the 11th Electronics New Zealand Conference (ENZCon'04), 15-16 Nov, 2004, p 106-111, ISBN 0-476-01106-X, Massey University, Palmerston North, New Zealand.

5. F. M. Al-Ali and R. F. Browne, *VHDL Modelling of a RISC Microprocessor: Synthesis, Assembler, Loader, and Testing*, Proceedings of the 12th Electronics New Zealand Conference (ENZCon'05), 14-15 Nov, 2005, pp 63-68, ISBN 0-473-10634-5, Manukau Institute of Technology, Manukau City, Auckland, New Zealand.

CHAPTER ONE

INTRODUCTION

This chapter outlines the motivation behind this thesis, followed by the scope and logical structure of the research work undertaken. The key ideas to be presented in the body of the thesis are introduced and the contents of the chapters and appendices are briefly outlined.

1.1 How It All Started

So, why did I decide to do my MSc? And, why did I choose this particularly challenging yet equally inspiring subject of Microprocessor Design?

It all started back in Christmas of 1991; only a few months after I graduated with a B.Sc. degree in Electronics Engineering and Communications. It was the subject of microprocessor design, which really mesmerized my imagination! It was the idea of how powerful the human spirit is, how creative the minds of scientists are, and how today's technology is so developed to the point of conceiving such powerful inventions while keeping them tamed to the point of advancing our lives.

So, I wanted to design microprocessors! I wanted to sit in front of the screen and design the internal interconnections, components, and wiring diagram of the microprocessor. Although back then in 1991 it seemed to me no more than a dream, but the following 15 years led to the successful fruition of this MSc thesis in the FPGA implementation of a RISC microprocessor.

Embarking on this 15-year quest towards achieving my dream, my first job after graduation was as a PC Repair Technician. I did not know what exactly I had to do in the short term (with PCs), but I knew exactly where I am headed to in the long run (with designing microprocessors). Coming from an engineering background was the main drive behind my involvement in the hardware aspects of PCs as the starting point. Being involved with PCs, I learned of Intel's line of the then-popular microprocessors (8086/8088, 80286, 80386, 80486…).

The main events behind turning my dream of designing microprocessors into an actual MSc dissertation took place between 1991 and 1996. That was the time when I started to take notice of the fact that a great percentage of Intel's microprocessors were actually fabricated in Malaysia, where Intel has the largest chip manufacturing facility in the world, located on the industrial island of Penang in the northern part of Malaysia. Therefore, my dream of designing microprocessors became then synonymous with another one: working as a chip design engineer at Intel Malaysia. So, it was time to pay Malaysia a visit!

So, I arrived in Malaysia in October 1996. A few days later, I paid Intel Penang's office a visit. That was a historic event! I could not believe that I was finally walking into the offices of the Holy Grail of the microprocessor industry and the largest chip designer and manufacturer in the world! I asked the receptionist for a job application form. Instead, she got the human resources manager to come down and see me. He told me that Intel would not hire me unless I have, at the least, a Masters in VLSI/Chip/Microprocessor Design.

THAT was the moment when my dream became even more focused: to have a Masters in Microprocessor Design!

Of course, I did not end up getting any job at Intel back then but I continued to look out for any opportunities to pursue my MSc studies in this field. This opportunity did finally arise in 2000 at Massey University. It brings me so much pride now that this MSc dissertation is the end of that 15-year quest.

Now that the story behind this research has been told, the next section elaborates on the scope of this research and thesis overview.

1.2 Scope of This Research: Thesis Overview

This thesis is the outcome of research in two areas of the computer technology: microprocessor and multi-processor architectures (specifically from the perspective of how differently they tolerate highly-latent and non-deterministic events), and hardware design of complex digital systems containing both datapath and control (particularly microprocessors).

As a result, the key achievements of this work are based on the three key areas of research investigated and covered in this thesis. These are:

- The problems associated with tolerating highly latent and non-deterministic events in existing microprocessor and multi-processor architectures.

- The high level behavioural VHDL (Very High Speed Integrated Circuit Hardware Description Language) description of the novel vector micro-threading chip multi-processor architecture, which is proposed to efficiently tolerate such high latency and non-determinism. The starting point for the design of this micro-threading architecture is the popular MIPS RISC (Reduced Instruction Set Computing) processor architecture.

- The hardware implementation involving the VHDL description, synthesis and simulation of the MIPS R2000 RISC microprocessor onto an FPGA (Field Programmable Gate Array) chip. The MIPS microprocessor is an existing architecture and is implemented in this research to provide the baseline processor platform for the future micro-threading architectural add-ons and modifications.

This thesis shows that in order to achieve high processing speeds, current popular superscalar microprocessors (e.g. Intel Pentiums, Digital Alpha, etc) rely heavily on the technique of speculating the outcome of instruction flow in order to predict the behaviour of non-deterministic computing operations (as in loading operands from high-latency memory into the processor). This is fine only if the speculation is

correct. But, what if it isn't? If the speculation fails, this would mean that the processor has to abandon its current decision (which now proved to be the wrong one) for the instruction flow path taken and to start all over again with the other path (the actual correct one). This is a waste of valuable processing time and hardware resources and a reduction of performance when speculation fails. Therefore, these processors can achieve high performance only when the majority of speculations are successful (being able to predict the right path).

A part of the focus of this research is an investigation of the novel vector micro-threading architecture as an alternative approach to the current superscalar-based speculative microprocessor designs. Micro-threading is based on the not-so-novel multithreading technique, which avoids speculation altogether and instead, starts running a different thread of instructions while waiting for the non-deterministic outcome of the instruction execution to be resolved. This utilises the chip resources more efficiently without waste of any processing power.

As this research progressed, the baseline RISC processor platform, the MIPS R2000, was reviewed first then synthesized from the RTL (Register Transfer Level) description using VHDL and then simulated and tested. This was conducted in order for future research to build upon and add the micro-threading architectural add-ons and modifications.

One outcome of this research is the publication of a total of five papers (refereed and non-refereed) in five different conference proceedings within New Zealand [2, 4, 5, 6] and abroad [3]. It is worth mentioning here that [3] was a refereed publication in the conference proceedings of an international conference of high standing.

The next section briefly looks at the contents of the rest of the chapters in this thesis.

1.3 Contents of the Chapters

Chapter Two:

Survey of High-Latency Tolerance in Contemporary and Future Processor Architectures

This chapter provides the necessary background and motivation for this research work by addressing the first of the three key areas of research investigated and covered in this thesis. Therefore, in this chapter, existing material and literature is surveyed in order to shed the necessary light on the problem at hand: the shortcomings of existing and future processor architectures in terms of their tolerance for high-latency and non-determinism. The architectures surveyed are the Superscalar, VLIW (Very Long Instruction Word), EPIC (Explicitly Parallel Instruction Computing), Dataflow, and the different Multi-threading variants.

Consequently, this sets the scene for the introduction of the micro-threading architecture (which will be introduced in chapter three).

Chapter Three:

Introducing Micro-threading as a Solution to the Problem of High-Latency

In this chapter, the Micro-threading architecture as a proposed solution to the problems of high-latency and non-determinism, is formally introduced and described. The material presented here is based mainly on research work carried out by Jesshope [13, 32] and Jesshope and Luo [34, 39, 33] and then surveyed by the author [5, 6].

Chapter Four:

Hardware Design Methodology and EDA Design Tools

This chapter outlines the hardware design methodology, processes, challenges, CAD/EDA design tools, and lessons learnt from synthesizing a MIPS R2000 RISC microprocessor onto an FPGA VLSI chip. The chapter starts with an overview of the design process and hierarchical partitioning. Then, the issues of implementing the datapath (combinational logic) and memory (sequential logic) components onto the chosen Xilinx Virtex-II FPGA, are discussed. This determines the efficiency with which a design can be implemented on an FPGA chip.

Chapter Five:

Review of MIPS R2000 Architecture

This chapter presents a brief review of the basics of the MIPS R2000 microprocessor Instruction Set Architecture (ISA), or simply, Architecture. This is the interface between the highest layer of the microprocessor hardware and the lowest layer of the software. The basics outlined in this chapter constitute the foundation on top of which the rest of the chapters are based. This chapter is extracted mainly from excerpts from [47]. Wherever necessary and possible, reference to the relevant page numbers will also be made. This chapter is annotated with the author's comments and tailored adaptation for the context of this research.

Chapter Six:

VHDL Description and Synthesis of MIPS R2000 Microprocessor

This chapter presents a brief review of the Register Transfer Level (RTL) description of the MIPS R2000 microprocessor followed by the author's own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of the MIPS R2000 microprocessor includes synthesis onto the target Xilinx Virtex-II FPGA chip followed by simulating a machine language code running on this microprocessor. Again, this chapter is based on and complements the material presented in [47] and [48] and is annotated with the author's comments and tailored adaptation for the context of this research. The details are covered in Appendices A to C (on the *Companion CD*).

Chapter Seven:

Assembler/Loader for the Synthesised MIPS R2000 Microprocessor

This chapter presents a novel and unconventional way of writing an assembler/loader for the MIPS R2000 microprocessor synthesized in chapter six, using the VHDL language. This was simulated in Model Technology Inc. (MTI) ModelSim XE.

Chapter Eight:

VHDL Description of the Micro-threading Chip Multi-processor

This chapter briefly describes how the micro-threading architectural add-ons and components are added to the standard MIPS architecture to build the micro-threading microprocessor and also the chip multiprocessor. The micro-threading VHDL description presented in this chapter is at a high level of abstraction as it is a behavioural description augmented with algorithms. Some VHDL pseudo-code is also included. As elaborated in chapter four, this is the first step of the hardware design process for the micro-threading microprocessor/multiprocessor and, therefore, paves the way for future research in which these algorithms and high level descriptions are utilized in designing the final micro-threading microprocessor and/or chip multiprocessor.

Chapter Nine:

Conclusions and Future Work

This chapter concludes this thesis by reviewing the summaries of the key points from the previous chapters along with the important areas of research covered by the thesis. Conclusions are drawn and further areas of enhancement and future research work are listed.

A glossary is also provided following the list of references. The next section briefly looks at the contents of the appendices found on the accompanying *Companion CD*.

1.4 Contents of the Appendices (on the Companion CD)

Appendix A:

VHDL Description and Synthesis of MIPS R2000 Datapath Basic Building Blocks

This appendix presents a brief review of the Register Transfer Level (RTL) description of the basic building blocks for the datapath of the MIPS R2000 microprocessor followed by the author's own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of these datapath basic building blocks includes simulation and synthesis onto the target Xilinx Virtex-II FPGA chip. Again, this appendix is based on and complements the material presented in [47] and [48] and is annotated with the author's comments and tailored adaptation for the context of this research. This appendix is the basis on which Appendix B builds upon to create the VHDL description and synthesis of the finalized full MIPS R2000 microprocessor in chapter 6.

Appendix B:

VHDL Description and Synthesis of MIPS R2000 Complete Datapath

This appendix presents the development of the Register Transfer Level (RTL) description of the complete datapath (without the control unit yet) of the MIPS R2000 microprocessor. The datapath concepts are first reviewed and then followed by the author's own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of this complete datapath includes simulation and synthesis onto the target Xilinx Virtex-II FPGA chip. Again, this appendix is based on and complements the material presented in [47] and [48] and is annotated with the author's comments and tailored adaptation for the context of this research. This appendix is the basis on which Appendix C builds upon to create the VHDL description and synthesis of the finalized full MIPS R2000 microprocessor in chapter 6.

Appendix C:

VHDL Description and Synthesis of MIPS R2000 Control Unit

This appendix presents the development of the Register Transfer Level (RTL) description of the control unit of the MIPS R2000 microprocessor. The control unit concepts are first reviewed and then followed by the author's own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of this control unit includes simulation and synthesis onto the target Xilinx Virtex-II FPGA chip. Again, this appendix is based on and complements the material presented in [47] and [48] and is annotated with the author's comments and tailored adaptation for the context of this research. This appendix is the last

piece of the big picture used to create the VHDL description and synthesis of the finalised full MIPS R2000 microprocessor in chapter 6.

Appendix D:

Supplementary Material for Chapter Six

This appendix covers the supplementary material for Chapter Six. This includes, higher resolution figures, diagrams, and detailed VHDL code.

Appendix E:

Published Conference Proceedings

This appendix is a collection of the five papers generated by this research and published in conference proceedings. These papers are:

- *Survey of High-latency Tolerance in Contemporary Microprocessor Architectures [5]*
- *Survey of High-latency Tolerance in Future Microprocessor Architectures [6]*
- *An FPGA Implementation of a RISC Microprocessor [2]*
- *VHDL Modelling of a RISC Microprocessor: Synthesis, Assembler, Loader, and Testing [4]*
- *Behavioural VHDL Model of a Vector Micro-threading Chip Multi-processor [3]*

1.5 Summary and Conclusions

This chapter outlined the motivation behind this thesis, along with the scope and structure of the research work undertaken. The key ideas to be presented in the body of the thesis were introduced and the content of each chapter was briefly outlined.

The next chapter surveys high-latency tolerance in contemporary and future microprocessor architectures and the problems associated with that.

CHAPTER TWO

SURVEY OF HIGH-LATENCY TOLERANCE IN CONTEMPORARY AND FUTURE PROCESSOR ARCHITECTURES

This chapter provides the necessary background and motivation for this research work. This is achieved by surveying existing material and literature to shed the necessary light on the problem at hand, i.e., the shortcomings of existing and future processor architectures in terms of their tolerance for high-latency and non-determinism. The architectures surveyed are the Superscalar, VLIW (Very Long Instruction Word), EPIC (Explicitly Parallel Instruction Computing), Dataflow, and the different Multi-threading variants. Consequently, this sets the scene for the introduction of the micro-threading architecture, which is introduced in chapter three.

2.1 Introduction

This chapter presents a survey of contemporary and future microprocessor architectures from the viewpoint of their different techniques used in tolerating highly latent and non-deterministic events. This is a key factor in determining the microprocessor's performance. Each architecture is presented with a brief review and an example commercial implementation or at least a proposed one.

This survey starts with section 2.2 where the problem of high-latency tolerance is identified first. Then, a brief review of how the different microprocessor architectures are classified base on the von Neumann computing model is covered in section 2.3. The sections that follow review the different architectures and how they tolerate high-latency. The *Dataflow* architecture is reviewed in section 2.4. This exploits the finest-grained parallelism available in a program, using dataflow graphs. The *Superscalar* approach is reviewed in section 2.5, where one sequence of instructions is issued out of order on multiple datapaths. The *VLIW* and *EPIC* architectures are reviewed in section 2.6. They utilise sequential streams of wide instruction words, again executed on multiple datapaths. Finally, *Multithreaded* processors are reviewed in section 2.7. Multithreading provides coarse-grained parallelism through context switching between multiple threads of instructions. This chapter ends with a summary and conclusions in section 2.8.

2.2 The Problem of High Latency Tolerance

Pushing microprocessor designs to the extreme limits of high performance has led to different approaches in this field of design, in order to meet evolving and increasingly high, computational demands. This trend is reinforced by Jesshope and Luo [34] in the following extract:

> The transition from the earlier CISC, to RISC, and now to the post-RISC era has been driven by the demand to achieve good efficiency, optimising the common case to make it as fast as possible, obtaining a larger *IPC (Instructions Per Cycle)*, and extracting the best of *ILP (Instruction-Level Parallelism)* while executing sequential legacy code [34].

One main obstacle facing microprocessor designers in their quest towards achieving higher performance is that of tolerating high latency and non-determinism in instruction execution, such as the latency in the access to main or remote memory, responding to branches in control, or performing a floating-point division. Another example of non-determinism is the result of statically *scheduling* concurrent operations and the need for synchronisation between these operations. It is worth mentioning here that *scheduling* is the process of assigning specific instructions and their operand values to designated hardware resources at designated times [17].

In the following sections of this chapter, current and future micro-architectures (microprocessor architectures) are reviewed along with a description of how they differently address the above issues.

2.3 Review of Microprocessor Architectures Classification

At least four possible classes of micro-architectures can be recognized [53, 54] some of which continue in the ongoing evolution path of the von Neumann computer [53, 54] while others follow a totally different path:

- Microprocessors that retain the von Neumann architectural principle of *Result Serialisation* (where the order in which the instruction flows as observed from the outside by the compiler still retains the original sequential program order), despite the inherent use of out-of-order execution technique within the microprocessor [53]. Micro-architectures that belong to this class are today's *Superscalar* architectures. There is still a considerable effort being directed towards improving such architectures, e.g. *Superspeculative* [36, 35, 37], *Multiscalar* [18, 57, 58], *Trace* [50, 56, 68], *Datascalar* [14], and *Superthreaded* [53, 66]. All these approaches fall into the same category because the result serialisation must be preserved [53]. A reordering of results is performed in a *Retirement* or *Commitment* stage in order to fulfil this requirement. These architectures are reviewed in this paper.

- Microprocessors that modestly deviate from the von Neumann architecture principle while allowing the use of sequential von Neumann languages by compiling programs to the new instruction set principles [53]. Examples of such approach are the *Very Long Instruction Word (VLIW), Single Instruction Multiple Data (SIMD)*, and *Vector* architectures. VLIW is reviewed in this chapter. While the SIMD and vector architectures are not covered in this architecture, the vector instruction set is implemented in the proposed *Micro-threading* architecture discussed later (chapters 3 and 9).

- Microprocessors that optimise the throughput of a multi-programming workload by executing multiple threads of control at the same time [53]. In this case, each thread of control is a sequential thread of instructions executable on a separate von Neumann computer [53]. Two example architectures are the *Multi-threaded* approach and the *Chip Multi-Processor (CMP)*. Multi-threaded architectures are discussed in this chapter. CMP is not discussed explicitly, but rather implicitly as it is tightly coupled with multi-threading and micro-threading as discussed in chapters 3 and 9.

- Microprocessor architectures that deviate totally from the standard von Neumann architecture and that need to use new languages, such as *Dataflow* with *Dataflow Single-Assignment Languages (SALs)* [53]. These are discussed in this chapter.

One of the many motivations for this diversity of approaches is that of tolerating high latency and non-determinism, as explained earlier. In the remaining sections of this chapter, these micro-architectures are reviewed along with a description of how they differently address the above issues of latency tolerance.

2.4 Dataflow Architectures

The dataflow architecture exploits all the finest-grained parallelism available in a program [53]. In this architecture, there is no need at all for a program counter since program execution is driven only by the availability of operands at the inputs to the functional units. This is why this architecture is also known as being *data driven*, and hence the name "dataflow", as opposed to the standard von Neumann *control-flow* principle. Thus, in dataflow the parallelism is limited only by the actual data dependencies in the application program. Dataflow architectures are not classified as RISC and they represent a truly radical alternative to the von Neumann control-flow architecture because they use *dataflow graphs* as their machine language. Dataflow graphs specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instructions. A program for a dataflow architecture is usually written in a *SAL (Single-Assignment Language),* then compiled into a dataflow graph which is a directed graph consisting of named *nodes*, which represent instructions, and *arcs*, which represent data dependencies among instructions. When a program executes, data "propagates" along the arcs as data packets, called *tokens*. This flow of tokens enables an instruction when all its input arcs have been traversed. The arcs in the graph are represented dynamically as unique tags in the tokens and the firing mechanism uses special memories called *matching stores* to match operands to instructions.

While a single thread of control in other microprocessor architectures often does not incorporate enough fine-grained parallelism to feed multiple functional units of today's microprocessors, the dataflow approach resolves any threads of control into separate instructions that are ready to execute as soon as all required operands become available. Therefore, the fine-grained parallelism potentially utilised by a dataflow computer is much greater than the parallelism available for today's conventional microprocessors [53]. The massive parallelism generated in a dataflow computer is controllable through the implementation of techniques such as *K-bounded loops*, which are used to introduce false dependencies into the dataflow graph to limit and throttle concurrency [12, 5]. Thus, dataflow processors not only support a dynamic schedule but also dynamic parallelism, which is not required in compiling most imperative programming languages [13]. Consequently, code must be recompiled for a dataflow processor, and single-assignment languages produce much more parallelism than is usually required for good performance.

The dataflow concept offers the potential of high performance and was thought by many to provide the answer to the scheduling problem, but the solution it provides is far too general and the overheads are high. This is because the performance of an actual dataflow implementation can be restricted by two main

limitations; firstly, the need for deep pipelines and a resulting high ratio of non-productive instructions, secondly, the need for expensive matching logic required for matching pending operations with operands generated by other instructions.

Since the early 1970s, there has been significant research and practical realisations of dataflow computers [53]. These dataflow computers can be based on either a *Pure Dataflow* [64, 59, 19, 52] or an *Augmented Dataflow with Control-flow*. Examples of the pure dataflow model are the *MIT Static Dataflow Machine* (based on static dataflow) [15], *MIT Tagged-Token Dataflow Architecture* (based on dynamic dataflow) [9, 10, 21], and the *Monsoon Multiprocessor* (based on explicit token-store dataflow) [43, 45, 46, 44].

On the other hand, there has been active research and development in the field of augmenting dataflow with control-flow [53]. Examples are:

- *Threaded Dataflow* (multithreaded architectures which have evolved from the dataflow model), eg. *Epsilon-1 and Epsilon-2* processors [22] and, again, the *Monsoon Multiprocessor* [43, 45, 46, 44].
- *Large-Grain Dataflow*, an example of which is the MIT and Motorola joint venture *StarT* processor [41].
- *Dataflow with Complex Machine Operations*, an example of which is the *ASTOR* architecture [76].

These developments have also had a certain impact on the conception of high-performance processor architectures in the "post-RISC" era. For example, dataflow is used in *Tomasulo's Algorithm* (an example of *Dataflow Scheduling*), which is a hardware-dependent resolution scheme that allows for dynamic scheduling out-of-order execution of instructions in the presence of hazards. This technique was first introduced in the *IBM 360 Model 91* [62] and is now used in today's popular superscalar microprocessors, as we will be covered in the next section.

As noted above, the Monsoon multiprocessor features both explicit token-store dataflow and threaded dataflow. This is one solution in dataflow research to address the problems associated with dataflow architectures, as raised earlier (the need for deep pipelines, ...etc.). Two main features are introduced in the Monsoon Multiprocessor which are of direct relation to this research and are discussed below in a little more detail:

- ***Explicit Token-Store (ETS) Dataflow***

  This is an evolution from the tagged-token dynamic dataflow principles [28]. ETS was the result of Papadopoulos' work [46, 44] and was later incorporated in the Monsoon multiprocessor, the latter being the product of a joint effort of the MIT Computation Structures Group, and the Motorola company [45, 46] after ETS was developed. The earlier tagged-token dataflow architectures used an associative matching store to determine when instructions are ready for execution. For a two-operand instruction to

become enabled, tokens carrying the two-operand values must be received. The first token to arrive is held in the matching store until its associate companion has arrived. Two tokens provide a pair of operands to the instruction if they have the same tag. In ETS, the associative search for matching tokens is replaced by establishing a memory location where each synchronization takes place. This location is within an *Activation Frame* associated with each function activation. Therefore, the memory where the token is waiting is directly addressed, without the need for associative matching which is slow and expensive [27]. Every location that corresponds to an activation of a two-input operation is augmented with a *Presence Bit* that is initially in the "empty" state. When the first token arrives it notices the "empty" state and writes its value into the location. Then, the presence bit is set into the "present" state. The second token notices the "present" state and reads the location, and then resets the presence bit back to "empty". The operation defined by the instruction (from the tag) is performed on the incoming token's value and the value read, and a new token is generated with the resulting value. Details of how ETS works are found in [43, 44]. This ETS model is what is implemented in Monsoon. The benefit of eliminating the associative search is obtained, generally, by reducing the cost of dynamically allocating storage for an activation frame for each function initiation (pre-allocation is can occur when a static analysis of the program demonstrates that this is feasible [28]). Another innovation in the Monsoon is its support for I-structures, which refer to arrays of data in which reads of an element are made to wait until the element is defined by a write operation. Actually, as will be described later, the registers in the Micro-threaded architecture are in fact I-structures.

- ***Threaded Dataflow (Augmenting Dataflow with Control-flow)***

Evolving from the dataflow model, is the Monsoon multithreaded architecture [27]. The maximum configuration build consists of 8 processors and 8 I-structure memory modules using an 8x8 crossbar network, and became operational in 1991. The Monsoon is a *cycle-by-cycle-interleaving*, multithreaded computer due to its capability of *Direct Token Recycling* [46]. Direct token recycling allows a particular thread to occupy only one pipeline frame slot in the 8-stage pipeline, which implies that at least 8 threads must be active to achieve full pipeline utilisation [53]. Figure 2.1 shows the Monsoon execution pipeline. The question that arises here is how are threads tracked and scheduled in this threaded dataflow architecture since there are no program counters involved to refer to? The answer is that massively parallel processors operate in an asynchronous manner in a network environment, where asynchrony is used to solve the two fundamental latency problems: *Remote Loads* and *Synchronization Loads* [27]. One solution to this problem is Threaded dataflow, by multiplexing between many threads (when one thread issues a remote-load request, the processor switches to another thread) where "full/empty" bits present in memory words are used to synchronize remote loads associated with different threads [27]. In threaded dataflow, the threads are tracked and scheduled by associating each remote load and response with a *thread identifier* (referred to as a "continuation on a message") for the appropriate thread, so it could be re-enabled upon arrival of a response. A large hardware-implemented *Continuation Name Space* is provided to store an adequate number of threads for remote responses [27].

Threads in Monsoon are short instruction sequences that access the local variables of its function initiation from its activation frame, and pass intermediate results using a small register file, thereby eliminating the need for dataflow synchronization during each instruction of a program [28].

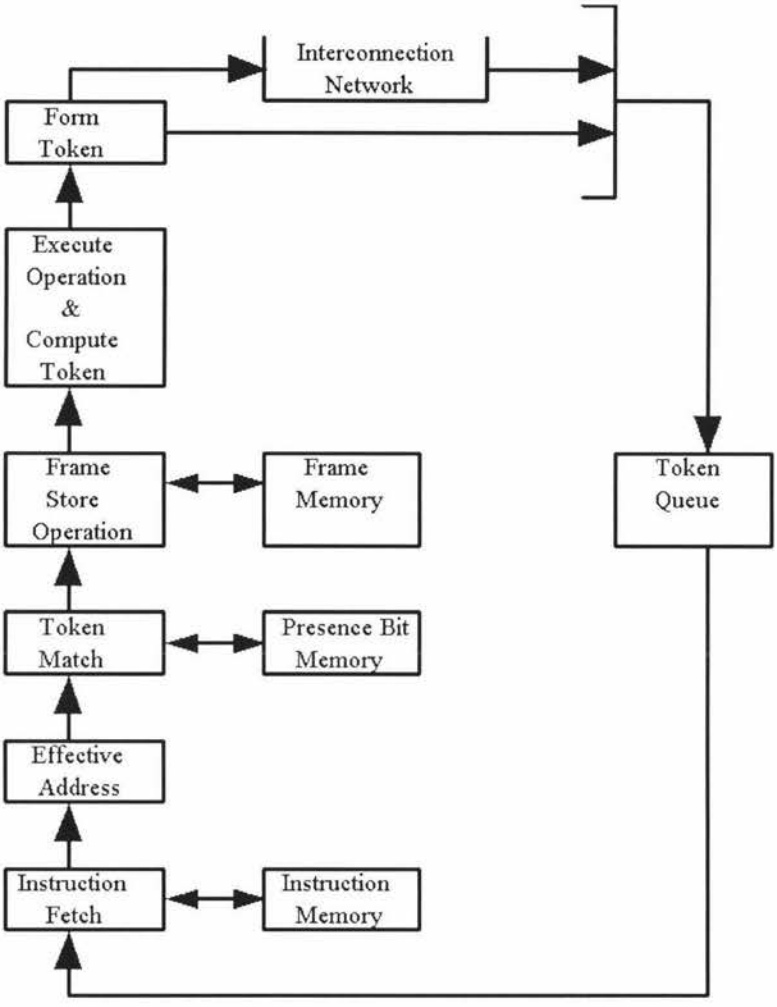

Figure 2.1 Monsoon multiprocessor execution pipeline [53].

2.5 Superscalar Architectures

Superscalar microprocessors are implicit multiple-issue processors [53]. The principal motivation behind the superscalar architecture was to overcome the single-issue of uni-scalar (single pipeline) RISC processors by providing the facility to fetch, decode, issue, execute, retire, and write back results of more than one instruction per clock cycle. The instructions are scheduled dynamically by the hardware. In other words, a conventional serial instruction stream is split dynamically into concurrent instruction sequences by an instruction window during execute time inside the processor [17].

The superscalar trend is based on replicating the internal datapath components of the microprocessor so that it can launch multiple instructions in every pipeline stage. This allows for the instruction execution rate to exceed the clock rate and for the *CPI (Cycles Per Instruction)* to be less than 1 [47]. The superscalar pipeline (as shown in Figure 2.2) features several independent *Functional Units (FUs)* that execute instructions independently.

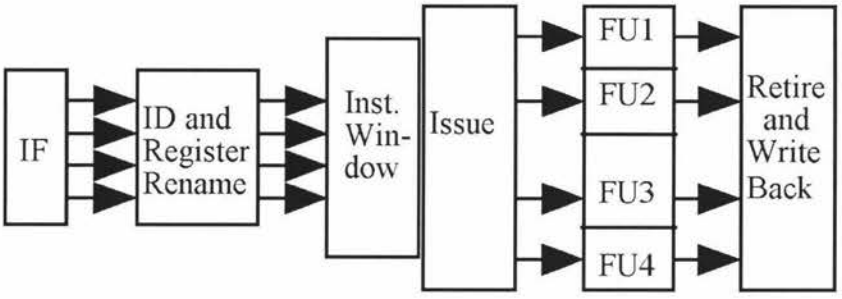

Figure 2.2 A quad-issue superscalar pipeline [53].

The problem of tolerating high latency in superscalar processors is addressed with the use of *Out-of-Order Execution* hardware that invokes different techniques such as *Dynamic Branch Prediction*, and *Control Speculation* to solve the problems of control and data conflicts and hazards [53, 5]. A point worth mentioning here is that out-of-order execution is an example of dynamic scheduling which implements Tomasulo's algorithm mentioned earlier in section 2.4. The performance of branch prediction depends on the prediction accuracy and the cost of misprediction. Although static branch prediction techniques can be used here, implementing the alternative dynamic branch prediction usually delivers better performance. The high penalty of misprediction, on the other hand, could be no less than 2 cycles, and sometimes up to 11 or more cycles in the Pentium II or the Alpha 21264 processors [53]. This is a high misprediction penalty, especially as these processors execute many instructions in each cycle. Another severe problem with the superscalar approach is keeping its pipelines full. This problem is well understood and solved using speculation. But speculative instruction issue generates write-after-read and write-after-write hazards, which would severely reduce instruction issue rates when not using register renaming [34]. That, again, adds complexity and cost to the design.

By now, it is clear that superscalar execution increases instruction throughput. Wider superscalar issue puts even more pressure on the compiler to deliver on the performance potential of the hardware. But data and control dependencies in programs, together with instruction latencies, offer an upper limit on delivered performance because the processor must sometimes wait for a dependency to be resolved, such as with a mispredicted branch. Furthermore, the techniques of out-of-order execution and dynamic branch prediction, as shown above, attempt to predict the non-determinism in the areas of cache accesses and branching, or try to counteract the effects of misprediction [13]. These techniques do increase the processing power of superscalar processors, but rarely by a factor proportional to the width of the pipeline used. Even with 4- or 8-way superscalar pipelines, it is difficult to obtain an IPC count of much more than 2 [34]. Moreover,

speculation makes a computer's performance dependent on the application and large penalties are paid for misprediction both in terms of execution time and, perhaps more importantly, in silicon chip area as much digital logic is used for the prediction mechanisms and for misprediction clean-up [34].

It is, thus, concluded that some tolerance of high latency is achieved in superscalar architectures. With wider issue pipelines, the performance depends critically on branch prediction accuracy and dynamic scheduling, as there is always a limit on the window in the instruction stream.

Most of current microprocessors utilize *ILP (Instruction Level parallelism)* by implementing a deep processor pipeline (more than five pipeline stages) and superscalar instruction issue techniques [5]. Future VLSI technologies will allow future generations of microprocessors to exploit aggressively ILP of up to 16 or even 32 IPC. Due to technological advances, gate-delay will be replaced by an on-chip wire-delay as the main obstacle to increase chip complexity and cycle rate [53, 54].

Superscalar processors began to dominate the microprocessor market at the beginning of the 1990s with dual-issue processors [53]. Today's superscalar microprocessors try to find six or more instructions to execute in every pipeline stage [47]. Examples of the most successful and popular commercial superscalar processors are:

- Intel i960 RISC processor, Pentium, Pentium Pro, Pentium II, Celeron, Klamath, and Pentium III [53, 47, 25].
- DEC Alpha 21064, 21164, and 21264 [53, 47, 25].
- IBM PowerPC 601, 603, 604, 620, and 750 (the G3) [53, 47, 25].
- MIPS R10000 and R12000 [53, 47, 25].

Following is a review of the current trends in superscalar architecture developments [53]:

2.5.1 Advanced Superscalar Processor Architecture

Although this architecture still focuses on using ILP together with speculation, these processors are wide-issue superscalars with an IPC of up to 32 [42]. This is achieved through the use of features such as; a large sophisticated trace cache for providing a contiguous instruction stream (more details on this approach are covered in subsection 2.5.3 shortly), a multi-hybrid branch predictor, a large number of reservation stations to accommodate approximately 2000 instructions, and 24 to 48 pipelined functional units. Figure 2.3 shows the internal architecture for such a processor [53, 54].

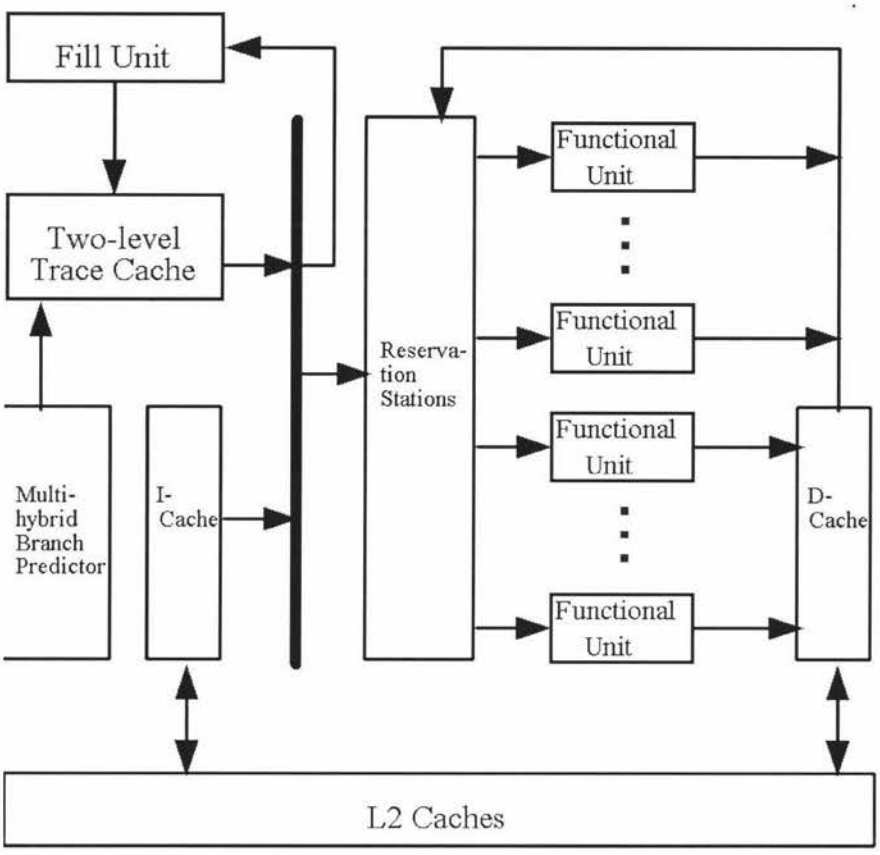

Figure 2.3 An advanced superscalar architecture [53].

2.5.2 Superspeculative Processor Architecture

These are also wide-issue superscalar microprocessors that use speculation techniques 53, 36, 35, 37]. This approach is based on the observation that in real programs, instructions generate many highly predictable result values. Therefore, consumer instructions can frequently and successfully speculate on their source operand values and begin execution without actual results from their producer instructions, thus removing the serialization constraints between producer and consumer instructions. As a result, it is claimed that the performance of a superspeculative program can exceed the classical dataflow limit which where even with unlimited machine resources, a program counter cannot execute any faster than the execution of the longest dependent instruction chain introduced by the program's actual data dependencies [53, 54]. It is further claimed that the dataflow limit is huge and unmatchable by any other architecture, and that the problems are in managing the dependencies. Superspeculative processors speculate on data dependencies, instruction flow, register dataflow, and memory dataflow in addition to branch prediction [25]. This is all possible by using the *Weak-Dependency Model* [47,43, 35], which states that dependencies can be temporarily violated during instruction execution as along as recovery can be guaranteed before affecting the permanent machine state. If a significant percentage of speculations are correct, the machine can exceed the performance limit imposed by the traditional, Strong-Dependency Model.

2.5.3 Trace Processor Architecture

The *Trace* processor is derived from the *Multiscalar* processor. The main features of this processor architecture are presented in [53, 50, 56, 68]. The trace processor extends the instruction window size to a trace, where traces are dynamic instruction sequences constructed and cached by the hardware. Thus, the *trace cache* stores dynamic instruction traces contiguously and fetches instructions from the trace cache rather than from the instruction cache. Since a dynamic trace of instructions may contain multiple taken branches, there is no need to fetch from multiple targets, as would be necessary when predicting multiple branches and fetching 16 or 32 instructions from the instruction cache. Trace processors also distribute the instruction window and register file to solve the instruction issue and register complexity problems (found in other designs such as *Simultaneous Multithreading*) by breaking up the processor into several *Processing Elements - PEs* (similar to Multiscalar – see next subsection 2.5.4) and the program into several traces so that the current trace is executed on one PE while the future traces are speculatively executed on other PEs. Because traces are neither scheduled by the compiler, nor guaranteed to be parallel, they still rely on control speculation and memory dependency speculation. The main difference between the trace processor and the multiscalar processor is that the traces in a trace processor are built as the program is executed, whereas the tasks in the multiscalar processor require explicit compiler support.

2.5.4 Multiscalar Processor Architecture

The *Multiscalar* model [53, 18, 57, 58] represents another architecture in which large amounts of inherent parallelism are extracted from a sequential instruction flow. Multiscalar and trace processors define several parallel processing cores, or PEs, that speculatively execute different parts of a sequential program in parallel. Multiscalar uses a compiler to partition the program segments, whereas a trace processor uses a trace cache to generate dynamically trace segments for the processing cores.

2.5.5 Datascalar Processor Architecture

Datascalar processors run the same sequential instruction stream redundantly across multiple processors using distributed datasets [53, 14]. Loads and stores are only performed locally by the processor that owns the data, but a local load broadcasts the loaded value to all other processors. Figure 2.4 demonstrates the execution of load and store operations for replicated and communicated memory. Assume that both processors execute a sequence of *load-1, store-1, load-2,* and *store-2.* Operations *load-1* and *store-1* are issued to the replicated memory and can therefore complete locally on both processors. Operations *load-2* and *store-2* are issued to the communicated memory of the first processor. The *load-2* of this processor is deferred until the value is broadcast from it. Since all processors are running the same program, they all generate the same store value, which is stored only in the communicated memory of the processor that owns the address. Therefore, *store-2* is completed by the first processor, but is aborted on the second processor. It is quite clear here that the datascalar approach emphasises redundancy rather than performance. The increase

in speed here is not from the increased ILP, but rather from increased data locality that is hiding the latency to some degree [53].

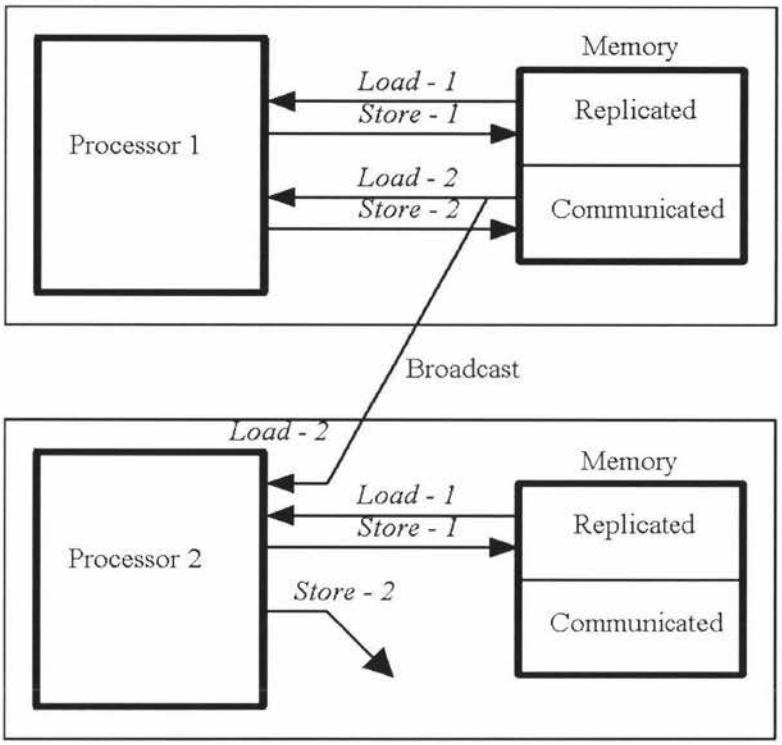

Figure 2.4 Datascalar processors access to replicated and communicated memory [53].

2.5.6 Superthreaded Processor Architecture

This is a concurrent multiple-threaded architecture for exploiting thread-level parallelism (TLP) on a processor [53, 66]. This architectural model adopts a thread pipelining execution model that allows threads with data dependencies and control dependencies to be executed in parallel, thereby enforcing data dependencies between concurrent threads. The basic idea of thread pipelining is to compute and forward recurrence data and possible dependent store addresses to the next thread as soon as possible, so the next thread can start execution and perform run-time data dependence checking on its own thread processing unit. Thread pipelining also forces contiguous threads to perform their memory write-backs in order, which enables the compiler to fork threads with control speculation.

The superthreaded architectural model can exploit loop-level parallelism from a broad range of applications through run-time support for data dependence checking and control speculation [53, 66]. The memory buffering and the in-order thread completion schemes allow control dependent threads to be executed concurrently with control speculation. Unlike the instruction pipelining mechanism in a superscalar processor, where instruction sequencing and data dependence checking and forwarding are performed by the processor hardware automatically, the superthreaded architecture performs thread initiation and data forwarding through explicit thread management and communication instructions. The execution of a thread

is partitioned into several pipeline stages, each of them performing a specific function. The first pipeline stage is the *Continuation Stage* where a thread starts after being initiated by its predecessor thread. The next stage is the *Target-Store-address-Generation (TSAG)* stage, which performs the address computation for *Target Stores (TSs)*. Target stores are performed by a thread, and are store operations on which later concurrent threads could be data-dependent. To facilitate run-time data dependence checking, the addresses of these target stores are calculated as soon as possible in the TSAG stage. The following computation stage performs the main computation of a thread. The last stage is the *Write-Back (WB)*. Because all of the stores are committed, thread-by-thread, write-after-read (anti-dependence) and write-after-write (output-dependence) hazards cannot occur during run time [53, 66]. Figure 2.5 depicts the organisation of the superthreaded processor.

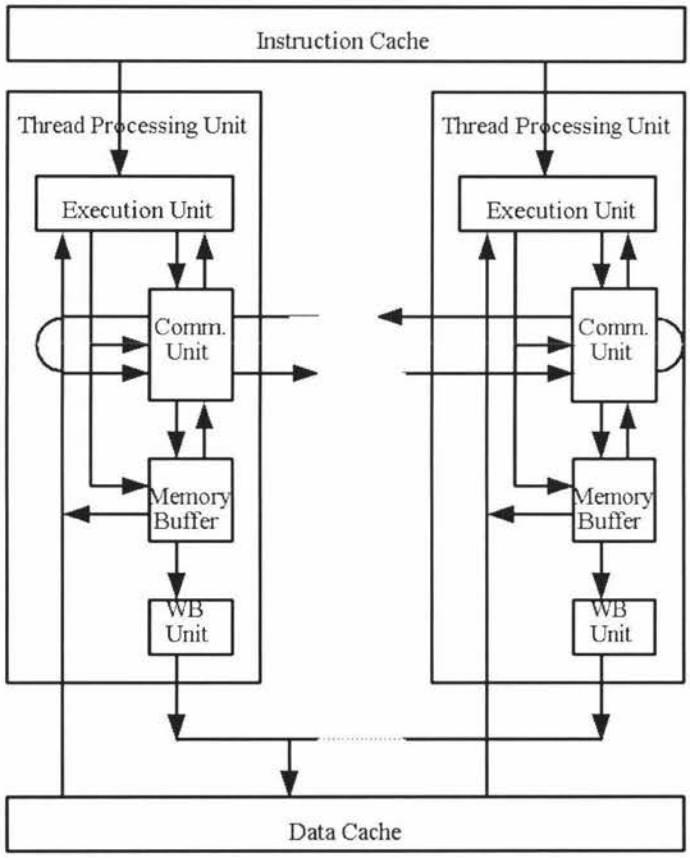

Figure 2.5 Superthreaded processor architecture [53].

2.6 VLIW and EPIC Architectures

2.6.1 VLIW Processor Architecture

VLIW (Very Long Instruction Word) microprocessors are, essentially, programmed multiple-issue processors [53]. They are predecessors to their superscalar counterparts but not as flexible. Therefore, they have been confined to signal processors during the last two decades, due to their fixed static schedule, which can easily be destroyed by non-determinism. This previously-not-very-successful VLIW technique has come into focus again recently with the introduction of the *EPIC* design style (covered in the next subsection), and its adoption by Intel for its 64-bit architecture.

VLIW processors use a sequential stream of long instruction words (called *instruction tuples*) that normally contain a fixed number of operations that are fetched, decoded, issued, and executed synchronously. All operations specified within a VLIW instruction must be independent of one another and, also, independent of previous instructions that may still be executing in the pipeline. Thus, VLIW is characterised by a *static issue* whereby a fixed number of instructions are issued each cycle, which are statically scheduled by the compiler. These instructions, as previously mentioned, can be organized as one large instruction or as a fixed instruction packet (instruction tuple) [53]. To summarise, dynamic (superscalar) issue utilising dynamic (out-of-order) scheduling is in contrast to static (VLIW) issue utilising static (in-order) scheduling [25].

VLIW microprocessors, including the Intel Merced architecture [29, 31] use concurrency detected by the compiler to perform *Aggressive Static Code Scheduling*. However, this solution also requires speculation, as static schedules do not work in the presence of non-determinism (conditional branch direction and memory access latency). The result is that misprediction may now require an interrupt and software intervention, and although non-computational hardware overhead is reduced, performance in the event of speculation failure is significantly impaired. VLIW usually implements speculation through the use of *Predication* or *Guard Bits*, and usually executes both branches of a condition until it is resolved. This still does not solve the basic problem however, which is that some operations are inherently non-deterministic. This situation performs more work than is strictly necessary in order to maintain a schedule in the presence of non-determinism. Only with hardware-based, asynchronous scheduling can the use of explicit concurrency overcome non-deterministic instruction execution. This approach is addressed in chapter 3, where *Micro-threading* is proposed as a solution. To conclude this discussion of VLIW, some commercial implementations are listed below:

- TI TMS320C6x family [61].
- Sun MAJC-5200 Chip Multi-processor [65, 60] (which also features multithreading).

- Transmeta Crusoe TM3200, 5400 and 5600 processors [63, 20].

2.6.2 EPIC Processor Architecture

Generally, the VLIW style of architecture addresses the issue of achieving high levels of ILP with reduced hardware complexity. However, it is specialised for numerical computing and has shortcomings when executing branch-intensive and pointer-based scalar applications. These were some of the reasons behind the evolution of VLIW, which led to the introduction of its successor, *EPIC (Explicitly Parallel Instruction Computing)*. The EPIC architecture is the result of the joint R&D project between Hewlett-Packard and Intel announced back in June 1994, aimed at developing the 64-bit instruction set and compiler optimisation, leading to the EPIC design style. This design philosophy seeks to further simplify hardware complexity while still extracting even more ILP from programs than either VLIW or superscalar strategies.

Actually, EPIC is a broad concept, as defined by *HPL-PD (Hewlett-Packard Laboratories – Play Doh)* architecture, which defines a large number of possible EPIC instruction set architectures [51]. The first commercially available EPIC ISA is Intel's *IA-64 (64-bit Intel Architecture)* [29, 31], also jointly developed by HP and Intel and introduced in 1999.

One of the main goals for EPIC is to retain VLIW's philosophy of statically constructing the *POE (Plan Of Execution)*. This is how the processor will execute the code. However, augmenting it with features (akin to those in a superscalar processor) would permit it to better cope with dynamic factors, which traditionally limited VLIW-style parallelism. To accomplish these goals, the EPIC philosophy is based on three main principles. Firstly, designing the desired POE at compile time, and the architecture should provide successful support for it. Secondly, the architecture should provide features that assist the compiler in exploiting static ILP. Thirdly, the POE must be conveyed to the hardware [51]. The architectural techniques that EPIC uses to support the above philosophy are *Static Scheduling, Branch Unbundling, Predicated execution, Control Speculation, Predicated Code Motion* [51, 29], and *Scalability* [53].

The EPIC architecture addresses the problem of memory latency with two techniques. The first is the use of *Cache Specifiers* [51], where load instructions are provided with a source cache specifier that the compiler utilises to inform the hardware of the cache location it can expect to find the referenced datum and, implicitly, the assumed latency. The second technique is *Data Speculation*, or sometimes called *Speculative Loading* [53, 51], where the processor will load data from memory well before the program needs it, and thus to efficiently minimise the impact of memory latency. This technique, just like predication, also is a combination of compile-time and run-time optimisations.

EPIC benefits from advanced compiler techniques that are closely coupled with the micro-architecture. EPIC exploits compiler ability, and enhances interactivity between the compiler and architecture. EPIC also uses *double branch execution, code movement*, and other techniques, but still relies heavily on speculation to overcome non-determinism. Finally, An example commercial processor of the EPIC architecture, and the IA-64 instruction set, is the Intel Itanium Processor [31].

2.7 Multithreading Architectures

We can conclude now that the superscalar, VLIW and EPIC architectures discussed so far, although they seem different in the way they address the scheduling and latency-tolerance problems, all introduce the potential for severely impaired performance when speculation fails. We have seen how superscalar, VLIW and EPIC either attempt to predict the non-determinism in the areas of cache accesses and branching, or try to mitigate against the effects of misprediction (by executing both branches of control concurrently), prefetching, and speculative loading. Often however, these techniques introduce further speculation, which can have an even more detrimental effect on performance in the event of misprediction.

According to Bolchevsky, Jesshope, and Muchnick [13], microprocessor architects are looking in the wrong direction. They are designing processors that try to prejudge and predict a program's data accesses or branches, when instead; they should simply look at tolerating the latencies involved. The proposed alternative approach is *Multithreading* generally, with *Micro-threading* being one of its numerous variants.

The multithreaded architecture is used as a solution to the problem of limited ILP in a conventional instruction stream. A multithreaded processor is based on the concept of additional utilisation of more *fine-grained* to *medium-grained* parallelism [6]. It optimises the throughput of a *multiprogramming workload* rather than just single-thread performance. This is done by executing multiple *threads* of control simultaneously. Each thread of control is a sequential program, and exists within a *context* of its own. Therefore the terms thread and context can be used interchangeably. Actually, a thread is a lightweight process (a few instructions) complete with minimal context [13], such as stack and registers. Thus, multithreaded processors are also known as *Multiple-context* processors, as they are based on the idea of switching the processor to another context (or thread) when a long-latency event occurs. This is possible because the multithreaded processor generally has several register files and maintains several PCs (Program Counters) along with related program states. Each register file and PC holds the program state for a separate parallel thread (or context). The functional units (FUs) are multiplexed between the threads in the register sets [6].

When long-latency or non-deterministic events are encountered, such as branches and loads, the processor switches to another thread, executing instructions from this new thread while the non-deterministic event is being handled [6]. This is called *Context Switching* and must be very fast for this to be effective [6].

Still, one problem with multithreaded architectures, in general, is that context switching might cause problems of loss of cache locality [6]. One proposed solution is the *Micro-threading* approach, introduced in the next chapter.

The different multithreading architectural approaches are discussed below.

2.7.1 Cycle-by-Cycle Interleaving

In this model, the processor switches to a different thread after each instruction fetch [53]. Some example implementations of this multithreaded architecture are:

- **Burton Smith's Delencor HEP**

  The *Heterogeneous Element Processor (HEP)* [53, 55, 28 , 27, 26] is a *MIMD (Multiple Instruction Multiple Data)* shared-memory multiprocessor system. Switching occurs between two queues of processes: one queue controls program memory, register memory, and the functional memory while the other queue controls data memory. The main processor pipeline has eight stages. Consequently, at least eight threads must be in cycle-by-cycle interleaving execution concurrently within a single HEP processor to give maximum performance. All threads within a HEP processor share the same register set. Multiple processors and data memories are interconnected using a pipelined switch. Any register-memory or data-memory location could be used to synchronize two processes on a producer-consumer basis via a "full/empty" bit synchronization on a data memory word [53, 32, 26]. Figure 2.6 illustrates the control loop for a single HEP pipeline. The 8-stage execution pipeline is shown, where *IF* denotes *Instruction Fetch*, *DF* for *Data Fetch*, *INC* for *Increment*, *PSW* for *Process Status Word*, and *SFU* for the shared memory. The pipeline is controlled by a queue of *Process Tags* (one for each thread representing an instruction stream). These process tags (or threads) rotate around the control loop, which executes one instruction from each thread every clock cycle. When an instruction accesses memory, it is removed from this loop and waits on memory in another queue (SFU queue). This is similar to *Vertical Micro-threading (VT)* as will be discussed in chapter 3, but with one main difference, being that the threads in HEP are stored in the memory queue, while in micro-threading, the micro-threads are stored in the registers. The problem with HEP is that to tolerate long memory access latencies, a large number of threads and non-blocking memory accesses are necessary.

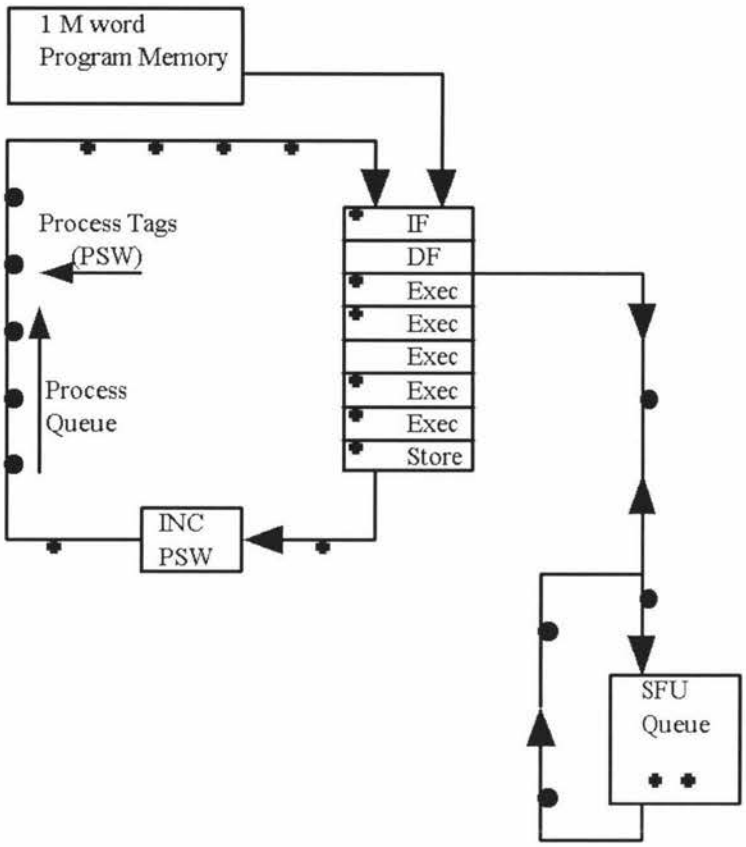

Figure 2.6 Control loop of a HEP pipeline [53].

- **Tera MTA Processor**

  The *Tera Multi-Threaded Architecture (MTA)* computer features a VLIW instruction set, a three-dimensional toroidal interconnection mesh network of pipelined packet-switching nodes, uniform access time from any processor to any memory location, and zero-cost synchronization and swapping between threads of control [53, 8, 28, 27, 7]. The uniform access time is accomplished through distributing the resources (i.e., processors, data memory units, I/O processors, and I/O cache units) uniformly throughout the network, instead of locating the processors on one side of the network and memories on the other. This allows data to be placed in memory units adjacent to the appropriate processor when that is possible and otherwise, generally, maximizes the distance between potentially interfering resources [8]. The Tera MTA exploits parallelism at all levels, from fine-grained ILP within a single processor to parallel programming across processors, to multi-programming among several applications simultaneously. As a consequence, processor scheduling occurs at multiple levels, and managing these levels introduces some unique and challenging scheduling problems [7]. The Tera MTA contains 128 thread contexts and register sets per processor node to mask remote memory access latencies effectively. This is considered too much overhead in order to tolerate such latencies, as a 128-register set is expensive to implement in logic. Figure 2.7 shows the Tera MTA 256 multiprocessor where the interconnection network is a 16x16x16 three-dimensional sparsely populated torus

architecture, with 4096 pipelined packet-switching nodes. Every processor posseses a clock register which synchronizes exactly with its counterparts in the other processors and counts up once per clock cycle [53]. The average latency in the Tera is about 70 clock cycles. This means that when a latency happens, this requires 70 different instruction streams to be running on each processor in order to tolerate such latency. The *Explicit-Dependence Lookahead* technique detailed in [8] allows streams to issue multiple instructions in parallel, which reduces the number of streams needed to achieve peak performance.

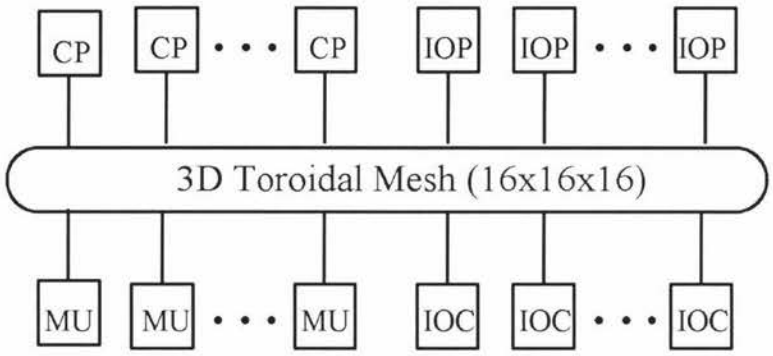

Figure 2.6 The Tera MTA 256 Computer System [53].

2.7.2 Block Interleaving

In this approach, a single thread continues to execute until it encounters a situation that triggers a context switch to another thread [53]. Such a situation could be a long-latency operation, which usually causes the pipeline to be flushed and a new register set is used. Example implementations of this multithreaded architecture are:

- **Sun MAJC-5200 Chip Multi-processor**

The *Microprocessor Architecture for Java Computing (MAJC)* processor architecture [65,60] from Sun Microsystems is based on a variable-length VLIW instruction set. Each *Processing Unit (PU)* contains 1 to 4 Functional Units (FUs). Each FU is viewed as a RISC processor in itself and is the basic building block of a PU. Individual instructions are issued to these FUs. Also, a new technique referred to as *Space Time Computing (STC)*, is used to enable speculative threads (future instruction streams) to execute across separate processor units, which substantially improves performance of many single-threaded and multithreaded applications. For example, if we have two processors on a chip, then two threads (Head and Speculative) execute on separate processors. They operate in a different space (speculative heap) and in a different time [60]. Also supported is *Vertical Multithreading (VMT)*, where the CPU switches to a new instruction stream (thread) whenever there is a cache miss. Each processor can switch between four different threads. The large register file maintains these four thread references.

➢ **Nanothreading**

Nanothreading [53, 24] proposed for the *Dansoft* processor breaks away from full multithreading by introducing a *nanothread* that executes in the same register set as the main thread. The DanSoft nanothread requires only a 9-bit PC, some simple control logic, and it resides in the same page as the main thread. Whenever the processor stalls on the main thread, it automatically begins fetching instructions from the nanothread. Only one register set is available, so the two threads must share that register set. Typically, the nanothread will focus on a simple task, such as prefetching data into a buffer, which can be done asynchronously to the main thread.

In the DanSoft processor, nanothreading is used to implement a new branch strategy that fetches both sides of a branch. A static branch prediction scheme is used, where branch instructions include 3 bits to direct the instruction stream. The bits specify eight levels of branch direction. For the middle four cases, denoting low confidence on the branch prediction, the processor fetches from both the branch target and the fall-through path. If the branch is miss-predicted in the main thread, the back-up path executed in the nanothread generates a misprediction penalty of only 1 to 2 cycles.

The Dansoft processor proposal is a dual-processor CMP, called *Dan 2433*, each processor featuring a VLIW instruction set and the nanothreading technique. Each processor is an integer processor, but the 2 processor cores share a floating point unit as well as the system interface.

However, the nanothread technique might also be used to fill the instruction issue slots of a wide superscalar approach as in simultaneous multithreading. Finally, nanothreading is proposed in the context of a block-interleaving multithreading technique.

2.7.3 Other Multithreading Architectures

➢ **Simultaneous Speculation Scheduling (S3)**

The architecture of *Simultaneous Speculation Scheduling (S3)* is a combined compiler and hardware technique to control multiple path execution [53, 67]. The S3 technique can be applied to dual path branch speculation in case of unpredictable branches and to multiple path speculative execution of loop iterations. In this approach, separate threads are generated by the compiler that harnesses thread-level speculation by speculating on the outcome of branches executing in parallel on a multithreaded microprocessor. Loop-carried memory dependencies that cannot be disproven by the compiler are handled by data dependence speculation. The architectural requirements are the ability to run two or more threads in parallel and three new instructions (fork, sync, wait) to control threads. This technique is targeted at simultaneous multithreaded, nanothreaded, and micro-threaded processors, but can also be modified for implementation in multiscalar, datascalar, and trace processors [6]. Applying the S3

technique to branches in kernel sections of SPECint95 benchmark programs shows a performance increase of up to 40% compared to purely static scheduling techniques [53].

- **Simultaneous Multi-threading (SMT)**

  The SMT processor [34, 16, 38] which is the result of combining the multithreading and superscalar architectures together. This leads to having all hardware contexts active simultaneously and competing each cycle for all available hardware resources [53]. This is why SMT is also known as the *Multithreaded Superscalar* approach. SMT architecture implements a rather large register file in which each thread can address 32 dedicated integer (and floating point) registers, and there are another additional 100 integer and floating-point renaming registers. Due to the longer access time of the larger register file, the SMT pipeline must be extended by using a two-cycle register read and a two-cycle register write [34]. The first commercial implementation of the SMT processor is the Intel Xeon implementing *Hyper-Threading*, which is the commercial name chosen by Intel for its SMT architecture [30].

- **Dynamic Multi-threading (DMT)**

  The DMT processor [34, 1] also uses an SMT pipeline to increase processor utilisation, except that the threads are created dynamically from the same program. The hardware breaks up a program automatically into loops and procedure threads that execute simultaneously on the superscalar processor.

2.8 Summary and Conclusions

This chapter presented a survey of different microprocessor architectures and how they tolerate high latency and non-determinism in instruction execution.

It was then shown that the solution to the latency-tolerance problem by each one of these architectures is both offset and compromised by the high overheads of the dataflow approach, the speculation involved in the superscalar approach, and the long context switch time introduced by the multithreading architecture.

This is where *Micro-threading* proposes a solution as a new approach towards highly efficient latency-tolerance and elimination of non-determinism through the use of *micro-threads* drawn from the same context. This is covered in the next chapter.

CHAPTER THREE

INTRODUCING MICRO-THREADING AS A SOLUTION TO THE PROBLEM OF HIGH-LATENCY

In this chapter, the Micro-threading architecture as a proposed solution to the problems of high-latency and non-determinism, is formally introduced and described. The material presented here is based mainly on research work carried out by Jesshope [13, 32] and Jesshope and Luo [34, 39, 33] and then surveyed by the author [5,6].

3.1 Introduction

It is was shown in the previous chapter (Ch.2) that the solution to the latency-tolerance problem by each one of the different microprocessor architecture families is offset by the high overheads of the dataflow approach, the speculation involved in the superscalar and VLIW approaches, and the context switch time introduced by the multithreading architecture. This is where *Micro-threading* is proposed as a solution to provide a new approach towards highly efficient latency-tolerance and elimination of non-determinism through the use of *micro-threads* drawn from within the same thread context.

This chapter starts with section 3.2 where micro-threading is first introduced. Then, a brief review of how micro-threading handles the issue of control transfer is covered in section 3.3. Section 3.4 introduces the basics of implementing micro-threading on a multi-CPU chip, while section 3.5 reviews the micro-threading vector instruction set architecture. This chapter ends with summary and conclusions in section 3.6.

3.2 What Is Micro-threading?

Micro-threading was introduced in [13, 34, 5, 6]. Simply put, micro-threading is multithreading within a single thread context [13]. In the micro-threading architecture, a thread (also known as *micro-thread*) is just a reference to a program counter. Non-deterministic events, such as branches and synchronisations, which may fail, will cause a new thread (program counter) to be executed, which may happen on every cycle. Thus, micro-threading combines the best of both block and cycle-by-cycle thread interleaving techniques. With the expectation that such threads will be rather small, maybe only a few instructions long, it is imperative that the overheads for fork, join and synchronisation are extremely low [13].

While existing multithreaded architectures are implicitly based on the assumption that latency tolerance requires massive parallelism, which must be found from diverse contexts, the quantitative analysis carried out for the efficiency of multithreaded execution as a function of the number of threads, shows that there are fundamental reasons for the efficiency to grow very rapidly with the number of threads [13]. This has been verified in [39] and, therefore, justifies the micro-threading approach, where the original goal of latency tolerance is achieved with only relatively few threads; these can easily be drawn from within the same referential context and do not, therefore, require the heavy weight hardware solutions of conventional multithreading [13]. This approach attempts to overcome the limitations of RISC instruction control (branch, loop, etc.) and data control (data miss, etc.) by providing such a low context switch time that it can be used not only to tolerate high latency memory, but also to avoid speculation in instruction execution [34]. It is, therefore, able to provide a more efficient approach to instruction pipelining [34].

3.3 Micro-threaded Control Transfer

Micro-threading performs true dynamic scheduling of several instruction streams by introducing the explicit notion of *independent points of control* (i.e. the manipulation of multiple program counters by the processor) **[13, 5, 6]**. Generally, a PC represents the minimum possible context information that can be kept for a given micro-thread, and it is the main reference to a micro-thread in the suggested micro-threading architecture **[13, 5, 6]**. Since several micro-threads can be active simultaneously, an explicit storage for their PCs, called the *Continuation Queue*, must be provided. This is associated with the instruction fetch logic at the entry point of the pipeline as shown in Figure 3.1 below **[13, 5, 6]**.

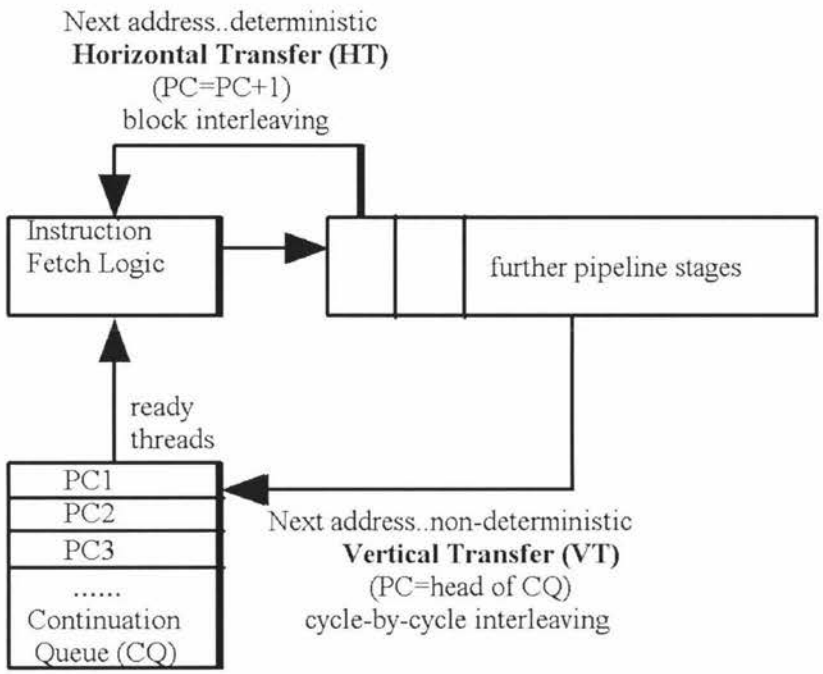

Figure 3.1 Micro-threaded Control Transfer [13, 5, 6].

In a normal RISC pipeline, the next address is transferred from the first stage of the pipeline in order to allow the next instruction to follow without delay. Branch instructions will normally involve speculation to predict the branch to be taken. If this prediction fails, any subsequent change of state must be "cleaned- up". This conventional mechanism of transferring control is called *Horizontal Transfer (HT)*, and the alternative mechanism proposed by micro-threading, which utilises the continuation queue, is called *Vertical Transfer (VT)*. In a vertical transfer, the next instruction is fetched from the PC at the head of the continuation queue. This is performed on non-deterministic operations **[13, 5, 6]**.

3.4 Micro-threading on a Multi-CPU

There are a number of different problems in designing a multi-threaded multi-CPU, with all CPUs sharing the same L2 cache. The major principle decision is whether a single or multiple register files should be implemented. Using a single register file means a potentially slow register access and possibly two pipeline cycles for an access, as demonstrated previously in the SMT architecture [34,5,16,38]. Additionally, there will be a large area overhead for multiple read ports to support all of the CPUs. On the other hand, there are also difficulties in the implementation of multiple register files, one per CPU. Firstly, there is a requirement for inter-CPU synchronisation and possibly data transfer. However, this is not so difficult to implement [5]. What poses the greater problem is the requirement to statically allocate resources, where *register identifiers* are allocated by the compiler for different threads that are executing a loop body, for example. This means that the compiler would effectively have to make a decision regarding thread allocation to each CPU, without any regard to the resolution of non-determinism and, hence, issues of load balancing. Clearly, this is not a good solution [5].

The proposed micro-threading solution to this problem is to effectively have multiple instances of exactly the same loop being executed (expecting the register resources not to be shared by the parallel threads). Ideally, one parametric loop body is written and then instanced as many times as the compiler thought necessary (i.e. depending on the number of CPUs) [5]. This situation is similar to register renaming, where additional registers are used to remove write-after-read and write-after-write hazards. Thus, a two-level *CQ (Continuation Queue)* is proposed. The first level holds ready threads that have not yet been allocated to any CPU. The register requirements for these ready threads are generic and subject to the limit of registers in any CPU's register file. This first level CQ is called the *Global Continuation Queue (GCQ)*. From the GCQ, threads are allocated to a CPU but only when that CPU has resources available. Once allocated, the thread runs to completion on that CPU and is held in the CPU's own CQ, the second level, called the *Local Continuation Queue (LCQ)*. Hence, there is a pool of unallocated threads in the GCQ, and a pool of allocated threads being held in the LCQ for each CPU. The allocation mechanism must ensure that register resources are available on the CPU where the threads are to be executed, and must rename the thread's local registers from their generic form to the actual registers allocated [5].

The proposed architecture for a single micro-threaded CPU is shown in figure 3.2, while figure 3.3 shows the multiple-CPU organization. For more details, see [32].

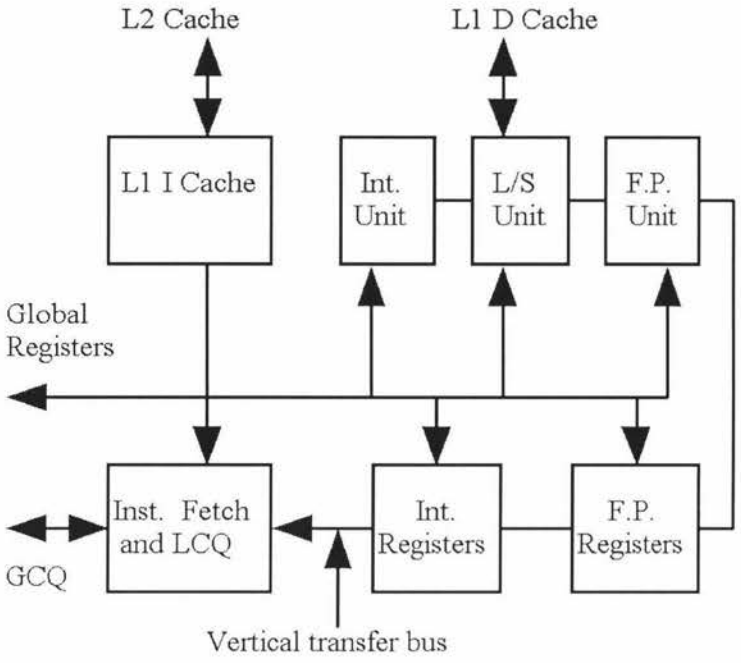

Figure 3.2 Micro-threaded Processing Unit [5, 6].

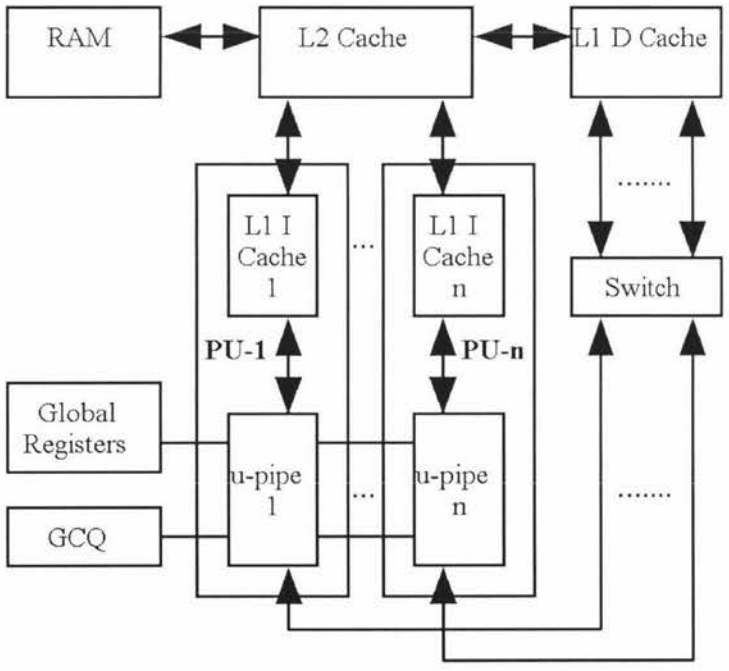

Figure 3.3 Micro-threaded Multiprocessing CPU [5, 6].

3.5 Micro-threaded Vector Instruction Set Architecture (ISA)

The following discussion is extracted from [32] where the micro-threaded vector instruction set architecture is described in detail. This is a combination of two different techniques from quite different eras in terms of computer architecture; one of which, using a vector instruction set has a long history dating back to pipelined vector supercomputers, such as the Cray 1 and its successors. The other technique, multi-threading, is also well understood. The combination can exploit both loop- and instruction-level parallelism without the need for speculation [32, 6]. This is important in the design of efficient chip-multi-processors, where large amounts of ILP are required. This novel approach proposed in [32, 6] combines both vertical and horizontal micro-threading with vector instruction descriptors, where it was shown that a family of threads can represent a vector instruction with dependencies between the instances of that family, the iterations. This technique gives a very low overhead in implementing an n-way loop and is able to tolerate high memory latency [32]. The use of micro-threading to handle dependencies between threads provides the ability to trade off between instruction level parallelism and loop parallelism [32, 6].

In a micro-threaded, vector architecture, threads are used to execute multiple loop bodies simultaneously, which provides parallelism to [32, 6]:

- Support multiple processors
- Keep the pipelines full in the presence of both data and control dependencies, and
- Tolerate high latency memory events.

To achieve parallelism on a large scale, it is imperative that just one instance of the loop body be used for all iterations. This is for reasons of code size and portability [32, 6].

In pipeline vector architectures, such as the Cray 1 [32, 6], vector instructions group single operations across the iterations in a loop. Thus a loop is transformed from a sequential execution model, where each instruction in the loop is executed for each iteration of the loop, to one where, each instruction in the loop is executed for all iterations of the loop before the next is executed. This grouping of the multiple scalar operations into vector operations allows the architecture to organise memory access and pipeline operation to achieve the optimal throughput of one cycle per operation, even for chained operations [6, 26]. Because of the parallel semantics of this execution methodology, there can be no dependencies between loop iterations [32, 6].

In a micro-threaded vector architecture, complete loop bodies can be executed in parallel for each loop index. This allows instruction level parallelism as well as loop parallelism to be exploited [32, 6]. Therefore any loop generates parallelism, even one containing a dependency between successive iterations. Code-

generation techniques normally used to maximise ILP at compile time, such as loop unrolling and software pipelining occur automatically at run time through dynamic thread creation. Thus the depth of unrolling is determined by the resources available on the target processor rather than by the compiler, giving more portable code [32, 6].

One problem faced, in devising a scheme to support the above execution model, is in the use of registers. This problem is addressed in [32, 39], as well as the means by which instruction classes may be instanced as independent parallel micro-threads along with an illustration of the speed-up that may be obtained compared to using a conventional loop.

3.6 Summary and Conclusions

This chapter presented micro-threading as a proposed solution to the problem of tolerating high-latency and non-determinism in existing and proposed microprocessor architecture.

The next chapter reviews the hardware design methodology and design tools used in designing and implementing the baseline MIPS RISC microprocessor onto an FPGA chip.

CHAPTER FOUR

HARDWARE DESIGN METHODOLOGY AND EDA DESIGN TOOLS

Good design doesn't just happen.

Good design is the end result of a search for inspiration [2].

This chapter outlines the hardware design methodology, processes, challenges, CAD/EDA design tools involved, and lessons learnt from synthesizing a MIPS R2000 RISC microprocessor onto an FPGA VLSI chip. The chapter starts with an overview of the design process and hierarchy partitioning. Then, the issues of implementing the datapath (combinational logic) and memory (sequential logic) components onto the chosen Xilinx Virtex-II FPGA are discussed. This determines the efficiency with which a design can be implemented on an FPGA chip.

4.1 Introduction

A micro-threading processor is based on the conventional MIPS R2000 RISC architecture, which is enhanced with the micro-threading architectural add-on components [13,34,32,39]. To be able to synthesize a hardware implementation of a micro-threading microprocessor, the baseline MIPS R2000 RISC processor has to be synthesized first [2].

In this chapter the author first addresses the issue of complex digital system design. This is covered in section 4.2. Then, in section 4.3, the design tools and target FPGA chip used in this research are outlined, along with my design methodology. Following on from this, section 4.4 illustrates the process of performing the design and synthesis. This is followed by the RTL description, synthesis, and simulation of a multiplexer as an example design in section 4.5. Finally, section 4.6 concludes this chapter with a summary and conclusions.

4.2 Complex Digital System Design

4.2.1 Design Project Workflow

A modular, layered approach is taken in designing complex digital systems. As outlined in figure 4.1, this process begins at the top with a specification of the requirements, and the final result is the description for manufacture and tape out of the system, usually implemented on a VLSI chip.

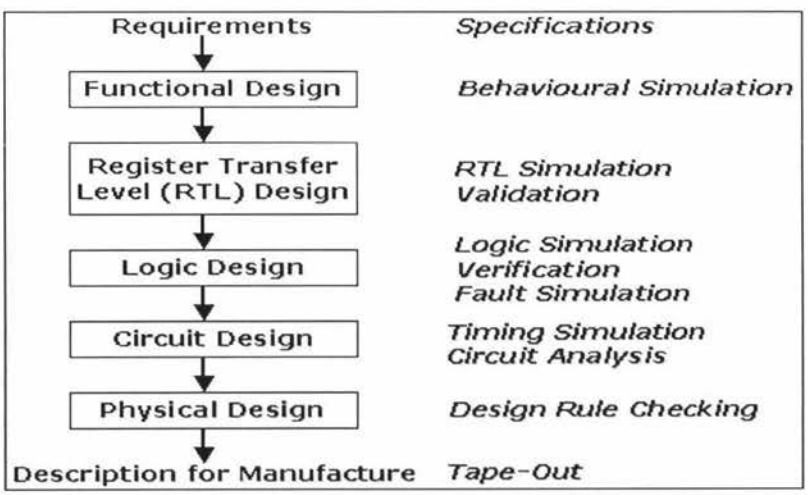

Figure 4.1 Design Project Workflow [75].

4.2.2 Design Views and Abstraction Levels

Any digital system can be described in one or all of the three different design views and abstraction levels represented by the Y-chart [75], which is outlined in figure 4.2 below. An example would be to describe a microprocessor in terms of its algorithm and instruction set architecture (behavioural), or its gate layout (physical), or in terms of an interconnection of its internal datapath and control units (structural).

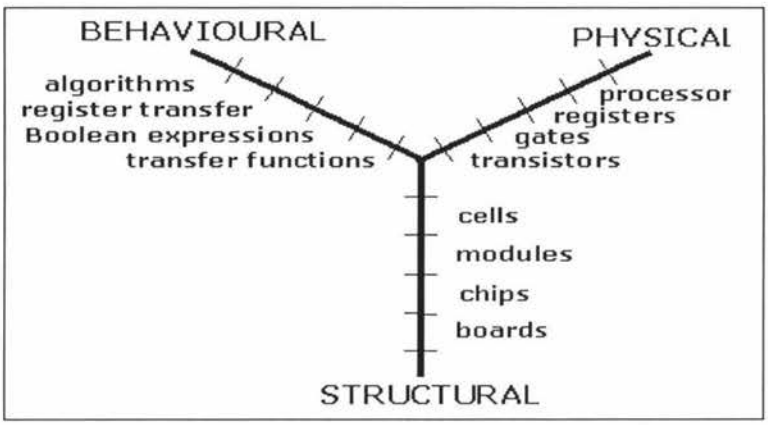

Figure 4.2 Y-Chart Representation for Design views and Abstraction Levels [75].

LESSONS LEARNT

The pros and cons for each of the above three design descriptions are listed below [2].

- ***Behavioural Description***
  - *Pros.* Ease of design at the highest level of abstraction without concern of the underlying hardware.
  - *Cons.* Requires tremendous optimization effort (and very long synthesis time) for the synthesis compiler to generate the hardware layout. This high dependence on the synthesis tools to ensure a consistent result is also a concern.
- ***Structural Description***
  - *Pros.* Designer has best control over exactly what hardware to synthesize. Thus, less effort is required from the synthesis compiler.
  - *Cons.* More manual effort (and time) required from designer.
- ***Physical Description***
  - *Pros.* Best representation for actual chip/board final layout/tape-out.
  - *Cons.* The most tedious and complicated style of design.

4.2.3 Hierarchical Design Approach

The starting point for a hierarchical design approach is the structural representation (as per figure 4.2). Figure 4.3 shows an example design hierarchy for combining logic gates into 32 reusable blocks of 1-bit ALUs (Arithmetic Logic Units), which in turn are combined into one 32-bit ALU at the top of the design hierarchy [40].

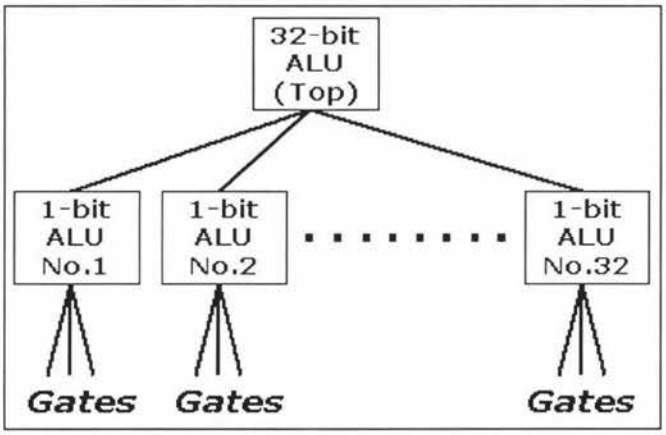

Figure 4.3 Design Hierarchy and Reusable Blocks [40].

LESSONS LEARNT

These are listed below [2].

- Tackle the design starting at the top of the hierarchy (*Top-Down* approach).
- Divide and conquer.
- The top design is repeatedly broken down (partitioned) into smaller blocks (modules, entities).
- Reusable blocks are instanced (used) again.
- Firstly, construct the smallest modules (at the bottom of the hierarchy), then combine them together working your way up the hierarchy (*Bottom-Up* approach).

SUMMARY

Break (partition) the design into smaller modules (Top-Down) then build it up from these smaller modules (Bottom-Up).

4.2.4 Module Design Entry

Most CAD/EDA (Computer-Aided Design/ Electronic Design Automation) tools would allow a module (the smallest designable and reusable block in your design hierarchy) to be described using one of the following methods [2]:

- ***Schematic Editor***

  Here, the module is constructed from a hierarchy of smaller components. These could be gates, library primitives, or smaller modules.

- ***Finite State Machine (FSM) Editor***

  The FSM editor is an HDL code generator used for creating, editing, and typically simulating FSMs. This option is best for modules represented in the form of an FSM. Example modules would be control units.

- ***HDL Editor***

  This is used when describing a module in a text-based Hardware Description Language (HDL). Examples of such languages are VHDL and Verilog.

4.2.5 Digital Design with VHDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a text-based industry standard (IEEE-1076) language for describing hardware and digital systems at multiple levels of abstraction (behavioural, RTL, structural, logic, ..) [40]. VHDL is the language of choice for this research due to its suitability for describing large complex digital systems like microprocessors [2].

LESSON LEARNT

In VHDL, any module (block) which is called *Entity,* must be associated with an *Architecture* describing its behaviour [2,75].

4.2.6 Best Design Practices: More Lessons Learnt

The final lessons and conclusions are summarized below [2].

- ***Hierarchy and Structure***

  The design must be structurally broken down into smaller modules (blocks), each of which can be comprised of other even smaller modules, and so on. This establishes a hierarchy of design modules.

- ***Behavioural Description***

  Never represent the whole design in one flat behavioural algorithm. Instead, break the design down into a hierarchy of smaller modules.

4.3 CAD/EDA Tools and Target Device

Since Field Programmable Gate Arrays (FPGAs) and their associated CAD/EDA software design tools are so popular nowadays for rapid prototyping, a very high-density FPGA has been chosen as the target device for this research [2].

4.3.1 Field Programmable Gate Arrays (FPGAs)

First introduced in 1985 by Xilinx Inc., an FPGA is a general-purpose SRAM-based programmable logic device (PLD) customised package. Figure 4.4 shows the structure of an FPGA, which comprises Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), interconnections and other resources [8]. The advantages of FPGAs include Non-recurring Engineering (NRE) costs, shorter time-to-market, low risk, and hardware prototyping (H/W Emulation) [2].

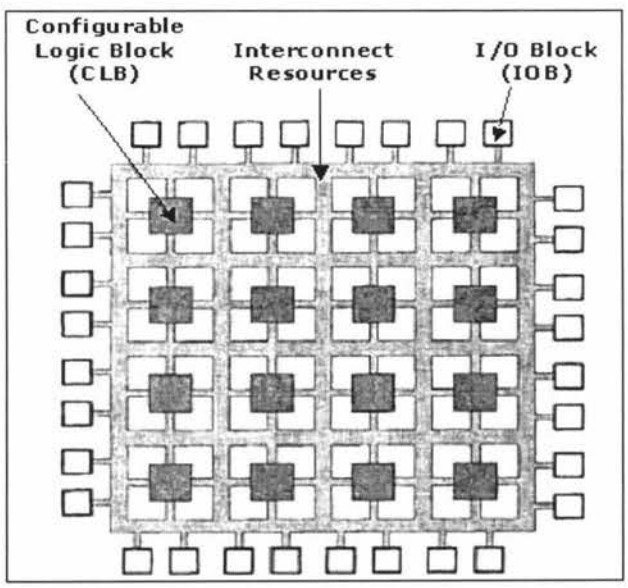

Figure 4.4 Structure of an FPGA chip [23].

4.3.2 Xilinx Virtex-II Platform FPGA

The Xilinx Virtex-II XC2V8000 is the chosen target device for prototyping this research design as it has 8 million re-configurable system gates. It also has a 420 MHz internal clock, 3 Mb of dual-port RAM in multiples of 18 Kb block SelectRAM, up to 1.5 Mb of distributed SelectRAM, dedicated 18-bit x 18-bit multiplier blocks, fast look-ahead carry logic chains, 12 DCM (Digital Clock Manager) modules, precise clock de-skew with GCLK (Global Clock), 93,184 internal registers/latches with CE (Clock Enable), and 93,184 Look-Up Tables (LUTs) [69]. Figure 4.5 shows the internal structure of this FPGA. The XC2V8000 has enough chip resources to implement the design outlined in this thesis.

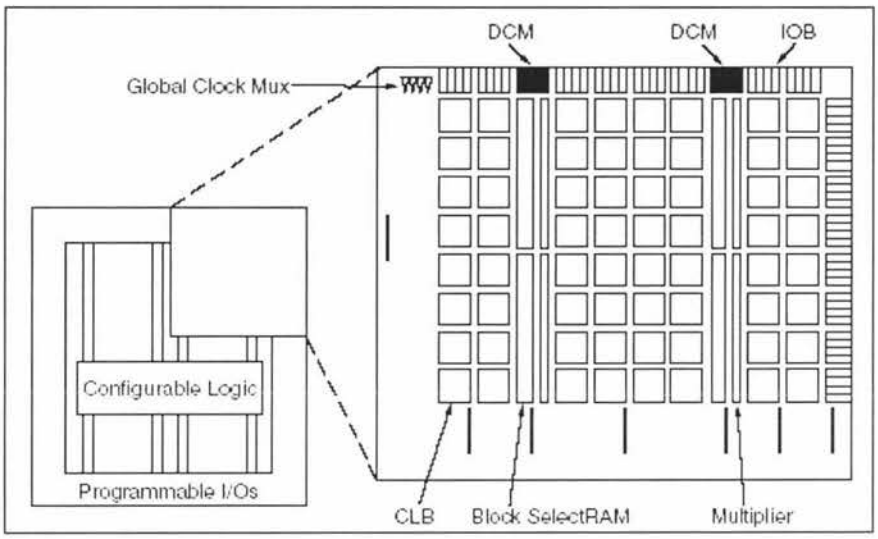

Figure 4.5 Xilinx Virtex-II Platform FPGA [69].

4.3.3 Xilinx ISE Design Tools for FPGAs

Figure 4.6 shows the design flow for FPGAs when using Xilinx ISE (Integrated Synthesis Environment) design tools. ISE version 5.1 was used for this research. Following are the four main stages/steps involved in the design flow process [72].

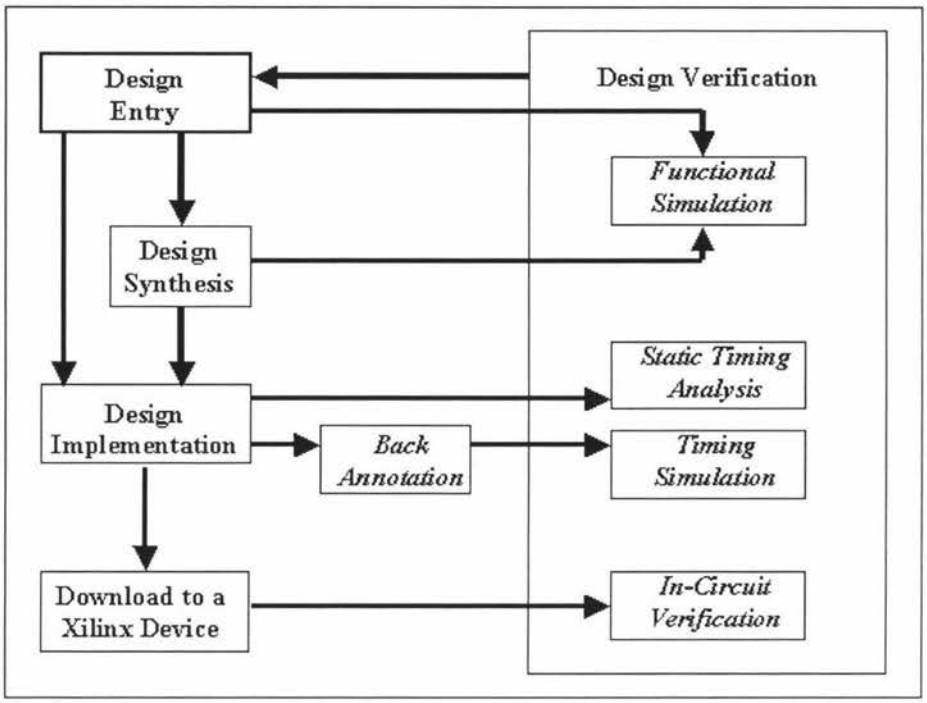

Figure 4.6 Xilinx ISE Design Flow for FPGAs [72].

- ***Design Entry***

  The design is created using a schematic editor, HDL editor, or state machine editor. This step is performed in this research.

- ***Design Synthesis***

  The synthesis compiler infers the hardware components. This step is performed in this research.

- ***Design Implementation***

  Implementing to a specific FPGA architecture (Spartan, Virtex,..). Optionally program a PROM or EPROM for subsequent programming of the FPGA chip. This step is beyond the scope of this research.

- ***Design Verification***

  Using a gate-level simulator or download cable, to test and ensure that the design meets the timing requirements and functions properly. This step is beyond the scope of this research.

4.4 Performing The Design and Synthesis

4.4.1 Design Methodology

Figure 4.7 outlines in more detail the author's design methodology adopted for this research. This design methodology is based on Xilinx technology [73].

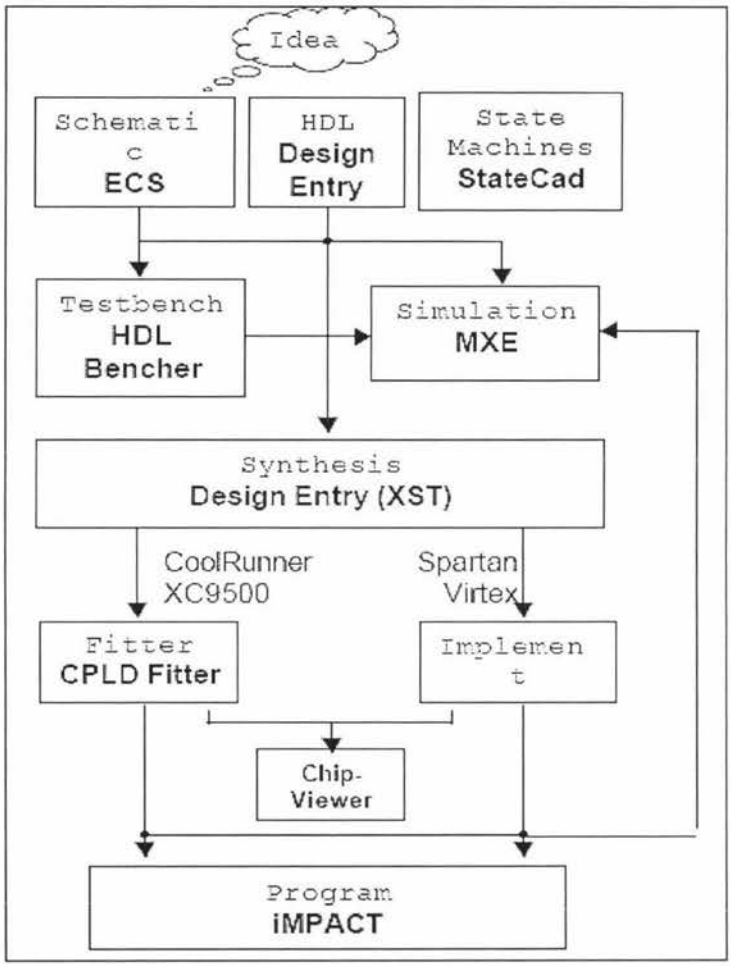

Figure 4.7 The author's adopted design methodology [73].

LESSONS LEARNT

These are listed below [2,72,73].

- Take each module separately, starting at the bottom of the hierarchy.
- Design it using VHDL, or schematics, or from an FSM diagram.

- Synthesize the logic hardware using XST (Xilinx Synthesis Technology).
- Check the synthesis report for percentage of FPGA resources utilized and any errors (if any).
- Check the RTL diagram generated by XST for the resulting schematic of lower level modules used.
- Create a testbench with the input test vectors (input signals).
- Run the simulation to check that the functionality of the design is correct.
- If not correct, then:
  - Repeat this process from the beginning by modifying the design.
  - Re-synthesize.
  - Re-simulate.
  - This iterative process continues until the functionality is satisfied.
- Move on to design the next module in the same level of the hierarchy, then in the next level up.

4.4.2 VHDL Simulation vs. Synthesis: Lessons Learnt

- ***VHDL for Simulation***

  A simulatable VHDL model of a block is usually used for describing the behaviour/functionality at the highest levels of abstraction. This is not necessarily synthesizable, as it is not necessarily bound to any device architecture (FPGA, ASIC,..). This is because only a subset rather than all of the VHDL language is synthesizable [2].

- ***VHDL for Synthesis***

  This is the best style for writing VHDL and is guaranteed to generate hardware logic as it must be bound to a specific device architecture. It is still not always optimal as the synthesis tools might generate much more logic than originally intended and take a long time in inferring (synthesizing) it [2].

- ***VHDL for Optimal Synthesis***

  Understanding the underlying device architecture allows for writing synthesizable VHDL code that generates the exact amount of hardware logic that you want in the least amount of time [2].

4.5 Example: Designing a Multiplexer

This section elaborates on the design process for a multiplexer, as an example of utilising the concepts presented so far in the previous sections. This design process is the methodology of choice followed throughout this research.

- *RTL Description*

One basic logic function that is used quite often in the MIPS hardware implementation is the *multiplexer*. A multiplexer is a combinational logic component. The multiplexer is described in detail on page B-9 of [47].

- *Design Entry and Synthesis*

Below is the VHDL code for synthesizing a 1-bit 2-to-1 multiplexer from the Xilinx ISE library using Schematic Editor:

```
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\m2_1e.sch

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Vhdl model created from schematic C:\Xilinx\virtex2\data\drawing\m2_1.sch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
-- synopsys translate_off
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
-- synopsys translate_on

ENTITY M2_1_MXILINX IS
   PORT ( D0   :       IN      STD_LOGIC;
          D1   :       IN      STD_LOGIC;
          S0   :       IN      STD_LOGIC;
          O    :       OUT     STD_LOGIC);

end M2_1_MXILINX;

ARCHITECTURE SCHEMATIC OF M2_1_MXILINX IS
   SIGNAL M0   :       STD_LOGIC;
   SIGNAL M1   :       STD_LOGIC;
```

```
   ATTRIBUTE BOX_TYPE : STRING;
   COMPONENT AND2
      PORT ( I0        :       IN      STD_LOGIC;
             I1        :       IN      STD_LOGIC;
             O         :       OUT     STD_LOGIC);
   END COMPONENT;

   ATTRIBUTE BOX_TYPE OF AND2 : COMPONENT IS "BLACK_BOX";
   COMPONENT AND2B1
      PORT ( I0        :       IN      STD_LOGIC;
             I1        :       IN      STD_LOGIC;
             O         :       OUT     STD_LOGIC);
   END COMPONENT;

   ATTRIBUTE BOX_TYPE OF AND2B1 : COMPONENT IS "BLACK_BOX";
   COMPONENT OR2
      PORT ( I0        :       IN      STD_LOGIC;
             I1        :       IN      STD_LOGIC;
             O         :       OUT     STD_LOGIC);
   END COMPONENT;

   ATTRIBUTE BOX_TYPE OF OR2 : COMPONENT IS "BLACK_BOX";
BEGIN

   I_36_9 : AND2
      PORT MAP (I0=>D1, I1=>S0, O=>M1);

   I_36_7 : AND2B1
      PORT MAP (I0=>S0, I1=>D0, O=>M0);

   I_36_8 : OR2
      PORT MAP (I0=>M1, I1=>M0, O=>O);

END SCHEMATIC;
```

The Xilinx ISE library schematic symbol for a 1-bit 2-to-1 multiplexor is shown in figure 4.8.

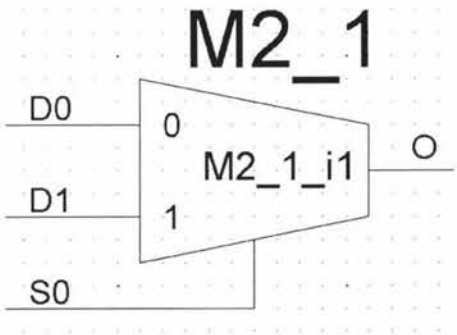

Figure 4.8 RTL Schematic symbol for a 1-bit 2-to-1 multiplexer in Xilinx ISE library.

➢ *Synthesis Results*

Using the Xilinx ISE synthesis tools, the hardware implementation for the above 1-bit 2-to-1 multiplexer, was generated. Figure 4.9 shows the resulting top level RTL symbol for the synthesized multiplexer while figure 4.10 shows the resulting top-level schematic diagram, which is also the gate level schematic.

Figure 4.9 Resulting top level RTL symbol for the synthesized 1-bit 2-to-1 multiplexer.

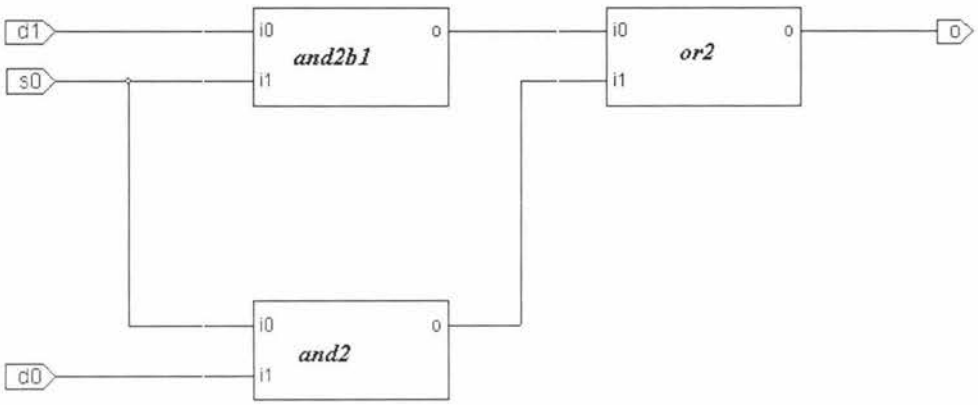

Figure 4.10 Resulting top level (is also gate level) schematic diagram for the synthesized 1-bit 2-to-1 multiplexer of figure 4.9.

➢ *FPGA Device Synthesis Summary*

After the hardware implementation for the above 1-bit 2-to-1 multiplexer using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```
Design Statistics:
# IOs                              : 4

Cell Usage:
# BELS                             : 3
#          and2                    : 1
#          and2b1                  : 1
```

```
#          or2                    : 1
# IO Buffers                     : 4
#          IBUF                   : 3
#          OBUF                   : 1

Device utilization summary:

Number of bonded IOBs:           4  out of   1108     0%
```

- *Place-and-Route onto the FPGA*

In figure 4.11, FPGA Editor shows the synthesized 1-bit 2-to-1 multiplexer after place-and-route onto the target Virtex-II FPGA chip. Notice that these are the small blue interconnections at the lower left corner in the figure.

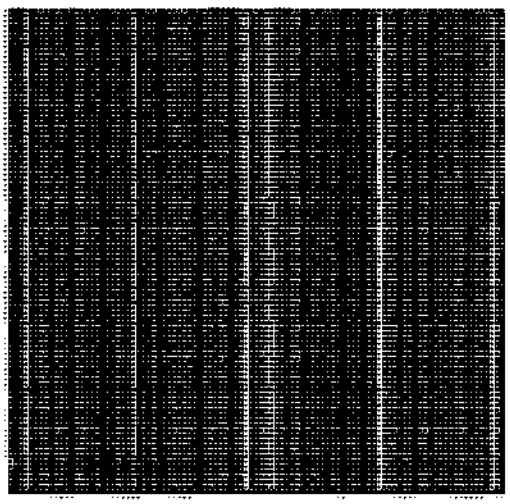

Figure 4.11 FPGA Editor showing the synthesized 1-bit 2-to-1 multiplexer after place-and-route onto the target Virtex-II FPGA chip.

- *Simulation Results*

Figure 4.12 shows the waveform results of simulating the 1-bit 2-to-1 multiplexer VHDL behavioural model in Mentor Graphics ModelSim by accepting input test vectors from a suitable VHDL testbench. All these waveforms are in binary format. It is clear that the resulting synthesized hardware functions according to the specified behavior of the multiplexer. This concludes the design cycle for this component.

Figure 4.12 Results of simulating the synthesized 1-bit 2-to-1 multiplexer using ModelSim.

4.6 Summary and Conclusions

This chapter presented the concepts and design methodology followed throughout this research in designing the hardware components and then the finalized MIPS R2000 microprocessor implementation on the Xilinx Virtex-II FPGA chip.

The next chapter reviews the MIPS R2000 instruction set architecture (also known as just "architecture").

CHAPTER FIVE

REVIEW OF MIPS R2000 ARCHITECTURE

This chapter presents a brief review of the basics of the MIPS R2000 microprocessor Instruction Set Architecture (ISA), or simply just known as Architecture. This is the interface between the top-most layer of the microprocessor hardware and the lower-most layer of the software. The basics outlined in this chapter constitute the foundation on top of which the rest of the chapters are based. This chapter is extracted mainly from excerpts from [47]. This chapter is annotated with the author's comments and tailored adaptation for the context of this research.

5.1 Introduction

In today's computer systems, both the hardware and software consist of hierarchical layers, with each lower layer hiding details from the layer above. This principle of *abstraction* is the way both hardware designers and software designers cope with the complexity of computer systems. One key interface between the levels of abstraction is the *instruction set architecture* (also known as *ISA* or simply just *architecture*): the interface between the hardware and the lowest-level software. This abstract interface enables many *implementations* of varying cost and performance to run the same identical software [47, p18].

This leads to the fact that the MIPS R2000 architecture described herein and in detail in [47], can be implemented either in a custom VLSI microprocessor chip, an embedded micro-controller, or in an FPGA chip; the latter being the scope of a major portion of this thesis work.

Any microprocessor can be programmed directly (explicitly) by writing programs (code) directly in *machine language*. The words of a machine's language are called *instructions*, and its vocabulary is called an *instruction set.* In this chapter, we will look at a subset of the MIPS R2000 instruction set, both in the form written and understood by humans (*mnemonics* form – *assembly language*) and in the form recognised and processed by the hardware (*binary* form – machine language) [47, pp106-107]. This instruction set subset is sufficient enough to implement a basic functioning MIPS R2000 microprocessor, as will be covered in forthcoming chapters.

This chapter starts the MIPS review with section 5.2 highlighting the underlying principles of MIPS hardware design, complemented by section 5.3 outlining the nomenclature implemented in this thesis. Section 5.4 follows with coverage of the basic MIPS instruction formats. Section 5.5 concludes the chapter with a summary.

5.2 Underlying Principles of MIPS Hardware Design

The MIPS R2000 is based on the RISC (Reduced Instruction Set Computer) principle, also known as the L/S *(Load/Store)* principle [17]. This is because the MIPS architecture does not work directly on operands that are found in the main memory, but rather these operands must be loaded from the main memory into the local register file (within the microprocessor) before operating on them, while the resulting operand from the operation can be stored back into the local register file, then into main memory.

This section covers the four underlying principles that were adhered to when originally designing the hardware for the MIPS microprocessor. These principles are [47]:

Design Principle 1: Simplicity favours regularity

The MIPS architecture is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three operands: two source operands (to be operated upon) and one destination operand [47, pp107-108]. Arithmetic operations are covered in section 5.4.

Requiring every arithmetic instruction to have exactly three operands conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number [47, p108]. This same philosophy applies to the instruction word length in the MIPS R2000 machine language; it's fixed to 1 word = 32 bits for all instructions.

Design Principle 2: Smaller is faster

An operand in a MIPS instruction can either be a constant/literal/immediate value supplied in the instruction itself, or a value stored in a register in the local register file. These registers are visible to the programmer. Each register in the MIPS architecture is 32 bits wide (= 1 word). The size of the register file in MIPS is 32 registers [47, p109].

The smaller-is-faster design principle may have had some historical basis and is the reason for the limit to 32 registers. A very large number of registers would increase the clock cycle time simply because it takes the electronic signals longer time when they must travel farther [47, p110]. Also, more registers simply means more complex instruction decoding and higher instruction latency.

Guidelines such as "smaller is faster" are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the programs demand for more registers with the need to keep the clock cycle fast [47, p110].

Design Principle 3: Good design demands good compromises

A problem occurs when an instruction needs to be longer than the fixed 32 bits. This is usually the case in instructions with a constant/literal/immediate operand value supplied in the instruction (refer to upcoming section 5.4), where the number of bits needed to represent the constant/literal/immediate is more than what can be accommodated [47, p118].

Hence, a conflict exists between the desire to keep all instructions the same length (design principle 1: simplicity favours regularity) and the desire to have a single unified instruction format. This leads us to the third hardware design principle: Good design demands good compromises.

The compromise chosen by MIPS designers is, as mentioned earlier, to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds/categories of instructions. These different instruction formats are described in section 5.4.

Design Principle 4: Make the common case fast

This design principle entails that common and frequently executed instructions are given more emphasis when designing the underlying hardware.

An example would be the fact that the use of constant/literal/immediate operands is quite common in almost any code (series of instructions). It is much faster to access a constant value if it is directly embedded within an instruction than if it is to be loaded from main memory [47, p146]. This is described in section 5.4.2.

5.3 Nomenclature

This thesis will adhere to the following nomenclature when referring to registers, memory locations, and their contents:

$Reg: Actual physical number of the specified register, also known as *register specifier*.

[$Reg]: Actual contents of register *$Reg*.

Memory: Address referring to memory location *Memory*.

[Memory]: Actual contents of memory location *Memory*.

Memory[$Reg]: Transfer program control to the memory location specified by the contents of register *$Reg*.

Memory[Mem]: Transfer program control to the memory location specified by the contents of memory location *Mem*.

Now that this nomenclature clarifies the conventions used in this thesis, the next section elaborates on the various MIPS instruction formats.

Another issue, which was decided upon within the context of this research, is to adopt a word-addressable implementation as opposed to the original MIPS architecture, which implements a byte-addressable policy.

5.4 MIPS Instruction Formats

Building upon the four underlying principles of MIPS hardware design (from section 5.2), this section describes the three main MIPS instruction formats. It is to be reiterated here that all instructions are of fixed length (32 bits). A MIPS instruction consists of distinct fields. MIPS instruction fields are given unique names to make them easier to discuss [47, p118]. Each field has a value assigned to it. This value is in binary format (in machine language). However, for human readability, these fields are usually represented also in decimal format, which is the format I'll be adopting throughout this chapter.

An issue worth mentioning here is that all the binary numeric values stored in registers are 2's complement signed binary numbers [47, ch.4].

Following are the three main instruction formats and their field layouts.

5.4.1 R-format Instructions

The R-format instruction layout is used in arithmetic, and logical instructions. Figure 5.1 shows the generic instruction encoding for such format.

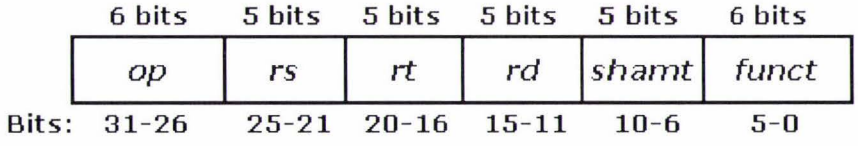

Figure 5.1 R-format Instruction Encoding[47, 4]

Here is the meaning of each name of the fields [1, p118]:

- *op:* Basic operation of the instruction, traditionally called the *opcode*. This has a unique value for each instruction as per the design of the ISA. Each instruction has a different set value for this field and is usually represented in decimal format.
- *rs:* The first source register specifier, usually represented in decimal format.
- *rt:* The second source register specifier, usually represented in decimal format.
- *rd:* The register destination specifier, usually represented in decimal format. It receives the result of the operation.

- *shamt:* Shift amount. This is reserved for use in shift instructions only. All other instructions will have this field set to the value zero (This is the case for all instructions implemented in this research).

- *funct:* Function. This field selects the specific variant of the operation in the *op* field, and is sometimes called the *function code*. This has a unique value for each instruction as per the design of the ISA. Each instruction has a different set value for this field and is usually represented in decimal format.

Following is a subset of the R-format instructions implemented in this research. The syntax and operation for each instruction is shown (the opcode is underlined for emphasis) along with a brief description and a figure illustrating its encoding layout.

- ***ADD** (Addition)*

This instruction adds the operand value found in the *first source register* $Src1Reg (=[$Src1Reg]) to the operand value found in the *second source register* $Src2Reg (=[$Src2Reg]) and stores the result operand value in the *destination register* $DestReg (=[$DestReg]). Figure 5.2 illustrates the encoding for this instruction.

Syntax: ADD $DestReg, $Src1Reg, $Src2Reg

Operation: [$DestReg] = [$Src1Reg] + [$Src2Reg]

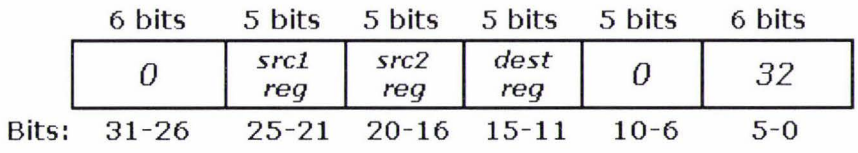

Figure 5.2 ADD Instruction format [47]

- ***SUB** (Subtraction)*

This instruction subtracts the operand value found in the *second source register* $Src2Reg (=[$Src2Reg]) from the operand value found in the *first source register* $Src1Reg (=[$Src1Reg]) and stores the result operand value in the *destination register* $DestReg (=[$DestReg]). Figure 5.3 illustrates the encoding for this instruction.

Syntax: SUB $DestReg, $Src1Reg, $Src2Reg

Operation: [$DestReg] = [$Src1Reg] – [$Src2Reg]

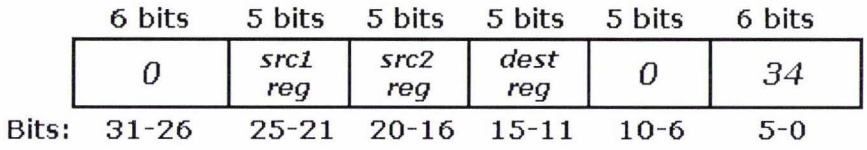

Figure 5.3 SUB Instruction format [47]

- ***SLT*** *(Set on Less Than)*

This instruction sets the value in the *destination register* $DestReg (=[$DestReg]) to 1 only if the operand value found in the *first source register* $Src1Reg (=[$Src1Reg]) is less than the operand value found in the *second source register* $Src2Reg (=[$Src2Reg]). Otherwise, the value in the *destination register* $DestReg (=[$DestReg]) is reset to 0 (zero). Figure 5.4 illustrates the encoding for this instruction.

Syntax: SLT $DestReg, $Src1Reg, $Src2Reg

Operation:
if [$Src1Reg] < [$Src2Reg]
then
[$DestReg] = 1
else
[$DestReg] = 0

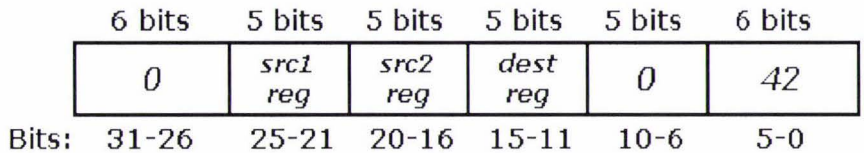

Figure 5.4 SLT Instruction format [47]

- ***JR*** *(Jump Register)*

This instruction causes the instruction execution flow of the program (specified by the contents of the *program counter register* PC) to start fetching the next instruction from the memory location specified by the value stored in the *register* $SrcReg (=[$SrcReg]). Figure 5.5 illustrates the encoding for this instruction.

Syntax: JR $SrcReg

Operation: go to Memory[$SrcReg]

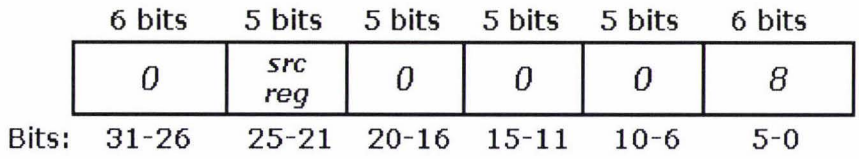

Figure 5.5 *JR* Instruction format [47]

- ***AND*** *(Logical AND)*

This instruction performs a logical/binary AND operation on the operand value found in the *first source register* $Src1Reg (=[$Src1Reg]) and the operand value found in the *second source register* $Src2Reg (=[$Src2Reg]) and stores the result operand value in the *destination register* $DestReg (=[$DestReg]). Figure 5.6 illustrates the encoding for this instruction.

Syntax: AND $DestReg, $Src1Reg, $Src2Reg

Operation: [$DestReg] = [$Src1Reg] *AND* [$Src2Reg]

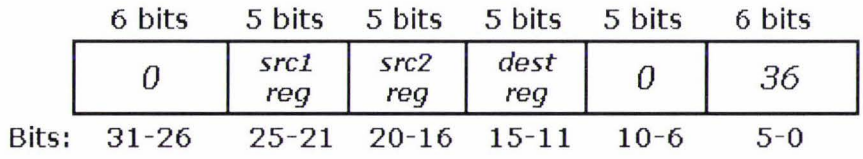

Figure 5.6 ***AND Instruction format [1]***

- ***OR*** *(Logical OR)*

This instruction performs a logical/binary OR operation on the operand value found in the *first source register* $Src1Reg (=[$Src1Reg]) and the operand value found in the *second source register* $Src2Reg (=[$Src2Reg]) and stores the result operand value in the *destination register* $DestReg (=[$DestReg]). Figure 5.7 illustrates the encoding for this instruction.

Syntax: OR $DestReg, $Src1Reg, $Src2Reg

Operation: [$DestReg] = [$Src1Reg] *OR* [$Src2Reg]

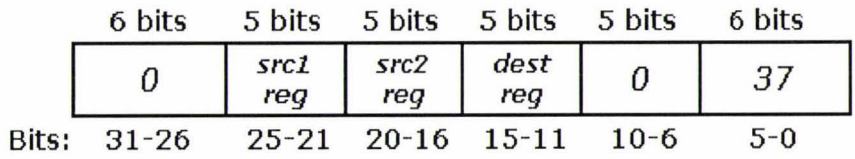

Figure 5.7 ***OR Instruction format [47]]***

5.4.2 *I-format Instructions*

The I-format instruction layout is used in control transfer, branching, and immediate value instructions. Figure 5.8 shows the generic instruction encoding for such format.

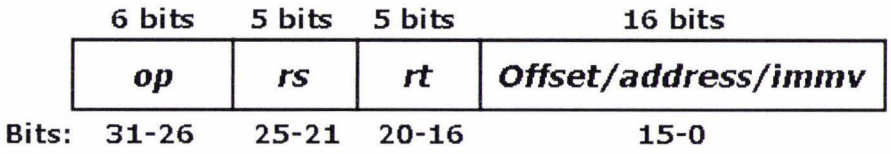

Figure 5.8 I-format Instruction Encoding [47, 4]

Here is the meaning of each name of the fields [47, p118]:

- *op:* Basic operation of the instruction, traditionally called the *opcode*. This has a unique value for each instruction as per the design of the ISA. Each instruction has a different set value for this field and is usually represented in decimal format.
- *rs:* The first register source specifier, usually represented in decimal format.
- *rt:* The second source register specifier, usually represented in decimal format.
- *offset/address/immv:* This is a 16-bit 2's complement signed immediate value supplied in the instruction itself. Depending on the opcode of the instruction, it can represent either an address, an offset, or simply an immediate value (literal).

Following is a subset of the I-format instructions implemented in this research. The syntax and operation for each instruction is shown (the opcode is underlined for emphasis) along with a brief description and a figure illustrating its encoding layout.

- ***LW** (Load Word)*

  This instruction loads the data from the memory location specified by sum of the *address* value (supplied in the instruction) and the contents of the *base register* $BaseReg (=[$BaseReg]) into the *destination register* $DestReg (=[$DestReg]). Figure 5.9 illustrates the encoding for this instruction.

  Syntax: LW $DestReg , address($BaseReg)

  Operation: [$DestReg] = [Memory [[$BaseReg] + address]]

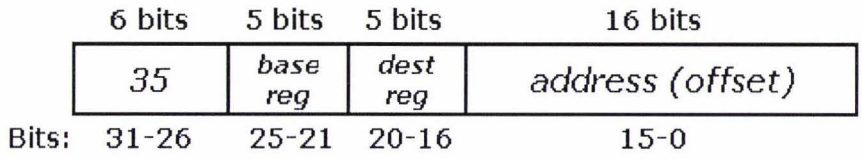

Figure 5.9 LW Instruction format (adapted from [47])

- ***SW** (Store Word)*

This instruction stores the data from *source register* $SrcReg (=[$SrcReg]) into the memory location specified by sum of the *address* value (supplied in the instruction) and the contents of the *base register* $BaseReg (=[$BaseReg]). Figure 5.10 illustrates the encoding for this instruction.

Syntax: SW $SrcReg , address($BaseReg)

Operation: [Memory [[$BaseReg] + address]] = [$SrcReg]

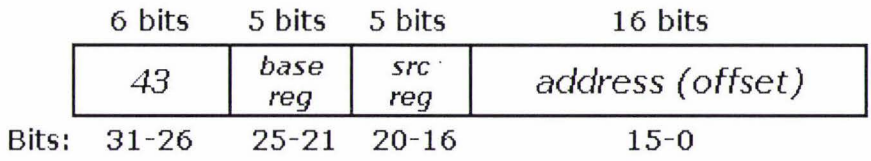

Figure 5.10 SW Instruction format [47]

- ***BEQ** (Branch on Equal)*

This instruction tests the equality between the contents of the *first source register* $Src1Reg (=[$Src1Reg]) and the contents of the *second source register* $Src1Reg (=[$Src1Reg]) and, if these were equal, this causes the instruction execution flow of the program to jump to the memory location calculated by adding the contents of the *program counter register* PC plus 1 (pointing to the next instruction after the current one) plus the *address* value supplied in the instruction itself. This process is referred to as *branch taken*. Otherwise, if the equality condition was not met, then the instruction execution flow of the program resumes as normal by pointing to the next instruction directly following the current one. This process is referred to as *branch not taken*.

An important note is that the value *address* is signed 2's complement with values ranging from -2^{15} to $+2^{15}$, which means that the BEQ instruction allows us to jump 32k locations in the positive or negative direction relative to the program counter register. Figure 5.11 illustrates the encoding for this instruction.

Syntax: BEQ $Src1Reg , $Src2Reg , address

Operation:
if [$Src1Reg] = [$Src2Reg]
then
[PC] = [PC] + 1 + address
else
[PC] = [PC] + 1

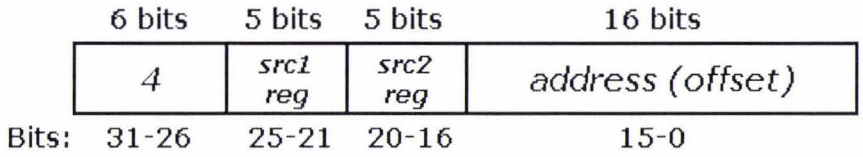

Figure 5.11 BEQ Instruction format [47]

- ***BNE** (Branch on Not Equal)*

This instruction tests the equality between the contents of the *first source register* $Src1Reg (=[$Src1Reg]) and the contents of the *second source register* $Src1Reg (=[$Src1Reg]) and, if these were NOT equal, this causes the instruction execution flow of the program to jump to the memory location calculated by adding the contents of the *program counter register* PC plus 1 (pointing to the next instruction after the current one) plus the *address* value supplied in the instruction itself. This process is referred to as *branch taken*. Otherwise, if the equality condition was met, then the instruction execution flow of the program resumes as normal by pointing to the next instruction directly following the current one. This process is referred to as *branch not taken*.

An important note is that the values *address* is signed 2's complement with values ranging from -2^{15} to $+2^{15}$, which means that the BEQ instruction allows us to jump 32k locations in the positive or negative direction relative to the program counter register. Figure 5.12 illustrates the encoding for this instruction.

Syntax: BNE $Src1Reg , $Src2Reg , address

Operation:
if [$Src1Reg] /= [$Src2Reg]
then
[PC] = [PC] + 1 + address
else
[PC] = [PC] + 1

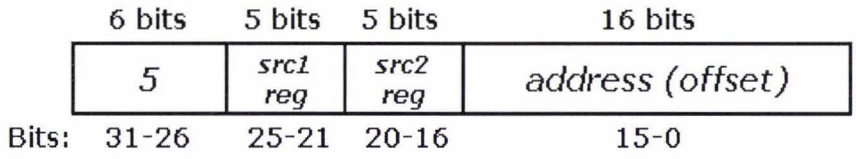

Figure 5.12 BNE Instruction format [47]

- ***ADDI*** *(Add Immediate)*

This instruction adds the operand value found in the *source register* $SrcReg (=[$SrcReg]) to the operand value *immediate* found in the instruction itself and stores the result operand value in the *destination register* $DestReg (=[$DestReg]). This instruction is used for adding constants. Figure 5.13 illustrates the encoding for this instruction.

Syntax: ADDI $DestReg, $SrcReg, immediate

Operation: [$DestReg] = [$SrcReg] + immediate

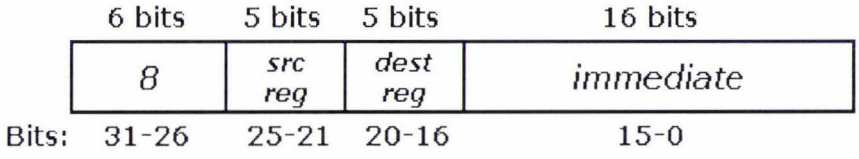

Figure 5.13 ADDI Instruction format [47]

- ***SLTI*** *(Set on Less Than Immediate)*

This instruction sets the value in the *destination register* $DestReg (=[$DestReg]) to 1 only if the operand value found in the *source register* $Src1Reg (=[$Src1Reg]) is less than the operand value *immediate* supplied in the instruction itself. Otherwise, the value in the destination register $DestReg (=[$DestReg]) is reset to 0 (zero). Figure 5.14 illustrates the encoding for this instruction.

Syntax: SLTI $DestReg, $SrcReg, immediate

Operation:
```
if [$SrcReg] < immediate
then
        [$DestReg] = 1
else
        [$DestReg] = 0
```

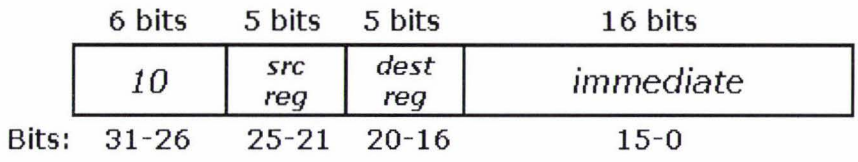

Figure 5.14 SLTI Instruction format [47]

5.4.3 J-format Instructions

The J-format instruction layout is used in control transfer through jumps. Figure 5.15 shows the generic instruction encoding for such format.

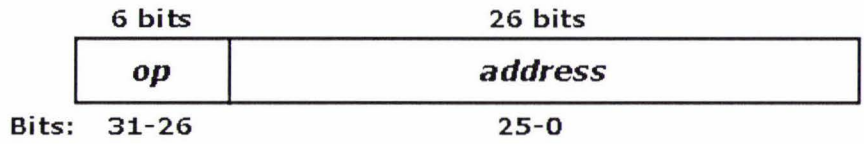

Figure 5.15 J-format Instruction Encoding [47, 4]

Here is the meaning of each name of the fields [47, p131]:

- *op:* Basic operation of the instruction, traditionally called the *opcode*. This has a unique value for each instruction as per the design of the ISA. Each instruction has a different set value for this field and is usually represented in decimal format.
- *address:*
  This is a 26-bit 2's complement signed immediate value supplied in the instruction itself.

Following is a subset of the J-format instructions implemented in this research. The syntax and operation for each instruction is shown (the opcode is underlined for emphasis) along with a brief description and a figure illustrating its encoding layout.

- ***J** (Unconditional Jump)*

  Detailed elaboration on the special functionality of the J instruction is found in [47, 48]. This instruction causes the instruction execution flow to jump unconditionally to a specific target *address*. This *address* value is supplied within the instruction itself and is a 26-bit value (2's complement). Figure 5.16 illustrates the encoding for this instruction.

  Syntax: J address

  Operation: go to target address

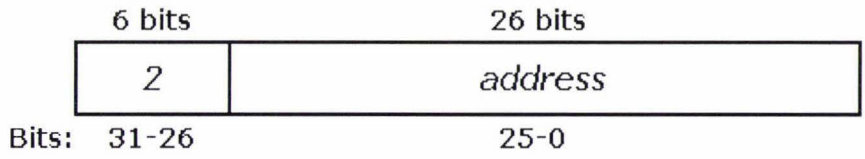

Figure 5.16 J Instruction format [47]

Elaboration [47, p150]:

The 26-bit field in jump instructions is also a word address, which means that it represents a 28-bit byte address. Since the PC (Program Counter) is 32 bits, 4 bits must come from someplace else. The MIPS jump instruction replaces only the lower 28 bits of the PC while leaving the upper 4 bits of the PC unchanged. The loader and linker must be careful to avoid placing a program across an address boundary of 256 MB (=64 million instructions). Otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

However, in the case of the author's implementation for this thesis, a PC of size 8 bits only has been implemented (due to hardware resource restrictions on the FPGA chip). Therefore, the issue of concatenation does not apply to this specific implementation.

5.5 Summary and Conclusions

This chapter presented a review of the MIPS R2000 instruction set architecture. The underlying principles of MIPS hardware design were highlighted, complemented by an outline of the nomenclature implemented in this thesis. The basic and most commonly used MIPS instruction formats were discussed.

This paves the way for the next chapter, which builds upon the material reviewed here, and elaborates on my research results in the hardware implementation of the finalized MIPS R2000 microprocessor.

CHAPTER SIX

VHDL DESCRIPTION AND SYNTHESIS OF MIPS R2000 MICROPROCESSOR

This chapter presents a brief review of the Register Transfer Level (RTL) description of the MIPS R2000 microprocessor followed by my own work on implementing this description in VHDL. This VHDL description (also called RTL Model) of the MIPS R2000 microprocessor includes synthesis onto the target Xilinx Virtex-II FPGA chip followed by simulating a machine language code running on this microprocessor. Again, this appendix is based on and complements the material presented in [47] and [48] and is annotated with my comments and tailored adaptation for the context of this research. The details are covered in Appendices A to C.

6.1 Introduction

This chapter presents the VHDL description, synthesis and simulation of the MIPS R2000 microprocessor hardware implementation of the MIPS instruction subset presented in the previous chapter (Ch.5; section 5.4). This MIPS hardware implementation is built from combining together the basic building blocks and datapath functional components to first build the larger datapath sections then the complete datapath (all of which is described in detail in Appendices A and B). Then, this complete datapath is combined with the control unit (described in detail in Appendix C) to make up the final MIPS R2000 microprocessor hardware implementation, which is covered in this chapter.

The format for presentation of the material in this chapter is the same as that in Appendices A to C, where the author takes each unit and briefly reviews its RTL description as described in [47] and [48], then follows it with his own work implementing this unit in VHDL, along with its synthesis and simulation. This process follows the design cycle (described in chapter 4) and comprised of the following steps: RTL Description, Design Entry and Synthesis, Synthesis Results, FPGA Device Synthesis Summary, and Simulation Results. Also, the logic conventions and clocking methodology followed in this chapter are detailed in Appendix A (section A.2).

It is worth noting here that some of the figures presented in this chapter are at a lower level of clarity, detail, and resolution, due to their complexity. However, at various points in the body of this chapter, reference will be made to higher resolution versions of these figures (allowing zoom in functionality) are found in Appendix D on the *Companion CD* accompanying this dissertation.

This chapter starts with an overview of the MIPS hardware implementation in section 6.2, thereby setting the scene for the material to follow, which is covered in section 6.3. Section 6.3 presents the VHDL description, synthesis and simulation of the MIPS R2000 microprocessor hardware implementation. Section 6.4 concludes the chapter with a summary.

6.2 An Overview of the MIPS Hardware Implementation

For the MIPS instruction subset reviewed in the previous chapter (Ch5; section 5.4) to be implemented in hardware, much of what needs to be done is similar, regardless of the actual instruction class [47].

For all MIPS instructions, the first two steps of execution are identical [47]:

- The program counter (PC) sends the instruction address to the instruction memory that contains the code (instructions) and, as a result, the required instruction is fetched from that memory location specified by the PC [47].

- Decoding the fields of the fetched instruction in order to select which registers (inside the Register File) to read. Then, one or two registers (depending on the class of the instruction) are read [47].

"After these two steps, the actions required to complete the instruction execution depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the instruction opcode" [47, p.339].

There exist some similarities even across the different instruction classes [47]. For example, all instruction classes utilise the Arithmetic Logical Unit (ALU) after reading the registers [47]. After using the ALU, the operations needed to complete executing the different instruction classes vary significantly [47]. More elaboration on this matter is found on pages 339 and 340 of [47]. Figure 6.1 below shows the high-level abstraction view of the MIPS R2000 microprocessor hardware implementation.

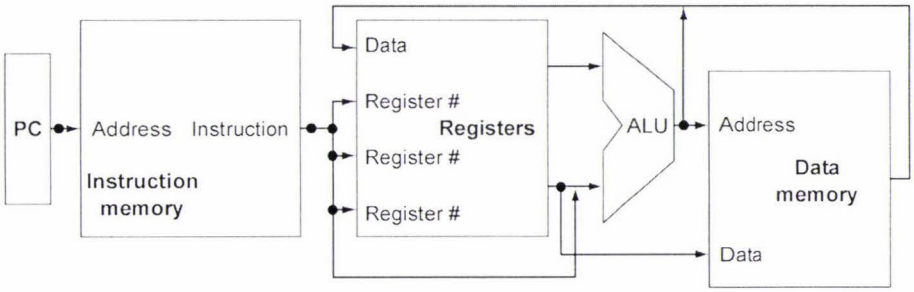

Figure 6.1 Abstract view of the hardware implementation of the MIPS instruction subset showing the major functional units and the major connections between them [47, p.340].

In section 6.3 that follows, this abstract view in figure 6.1 is refined to fill in all the details (to generate the complete datapath) and add the control unit, to form the final MIPS R2000 microprocessor.

It is worth noting that the MIPS hardware implementation in this research is based on the simple implementation detailed in [47] that uses one single clock cycle for the execution of each instruction. This means that each instruction begins execution on one rising clock edge and completes execution before the next rising clock edge [47]. However, in the context of this research, instruction execution is spread over a few clock cycles due to the read/write nature of the memory elements (register file, instruction memory, data memory) implemented on the FPGA chip.

6.3 Putting It All Together: The MIPS R2000 Microprocessor

➢ *RTL Description*

Figure 6.2 shows the RTL diagram for the finalised MIPS R2000 microprocessor hardware implementation of the abstract view shown earlier in Figure 6.1 and based on the MIPS instruction subset reviewed in Chapter 5. The details of this hardware implementation are discussed in [47] and [2] and elaborated in Appendices A to C on the Companion CD accompanying this dissertation.

The following ten MIPS instructions are implemented and tested on this MIPS R2000 microprocessor:

- AND (Logical AND)
- OR (Logical OR)
- ADD (Arithmetic Addition)
- SUB (Arithmetic Subtraction)
- SLT (Set on Less Than)
- LW (Load Word)
- SW (Store Word)
- BEQ (Branch on Equal)
- BNE (Branch on Not Equal)
- J (Unconditional Jump)

In Figure 6.2, the black and grey components and lines are all datapath units and their associated datpath signals, whereas the blue blocks and lines are all control related.

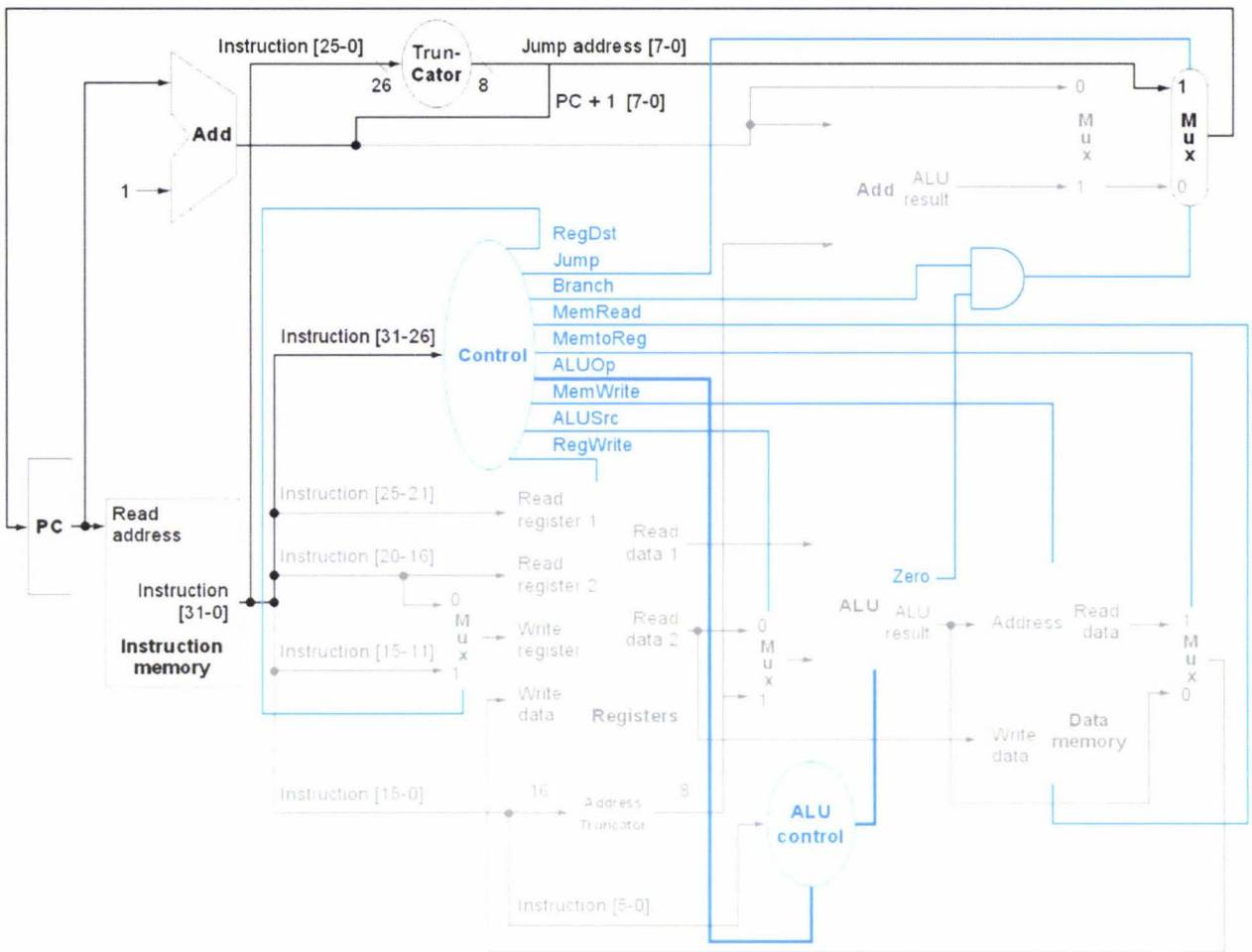

Figure 6.2 RTL Diagram for the finalized MIPS R2000 microprocessor (adapted from [47])

➢ *Design Entry and Synthesis*

The Xilinx Schematic Editor was used to create the design entry for the finalized MIPS R2000 microprocessor shown in figure 6.2. Figure 6.3 shows the final schematic diagram. Two important notes relating to figure 6.3 are worth mentioning here:

- All the datapath components (all the components shown in grey or black in figure 6.2) are combined and synthesized into one entity block called *Complete_Datapath_w_DCM_Div_5* (which is described in detail in Appendix B).

- The Main Control and ALU Control units (both shown in blue in figure 6.2 and are described in detail in Appendix C) and other supporting digital logic and design components make up the rest of figure 6.3.

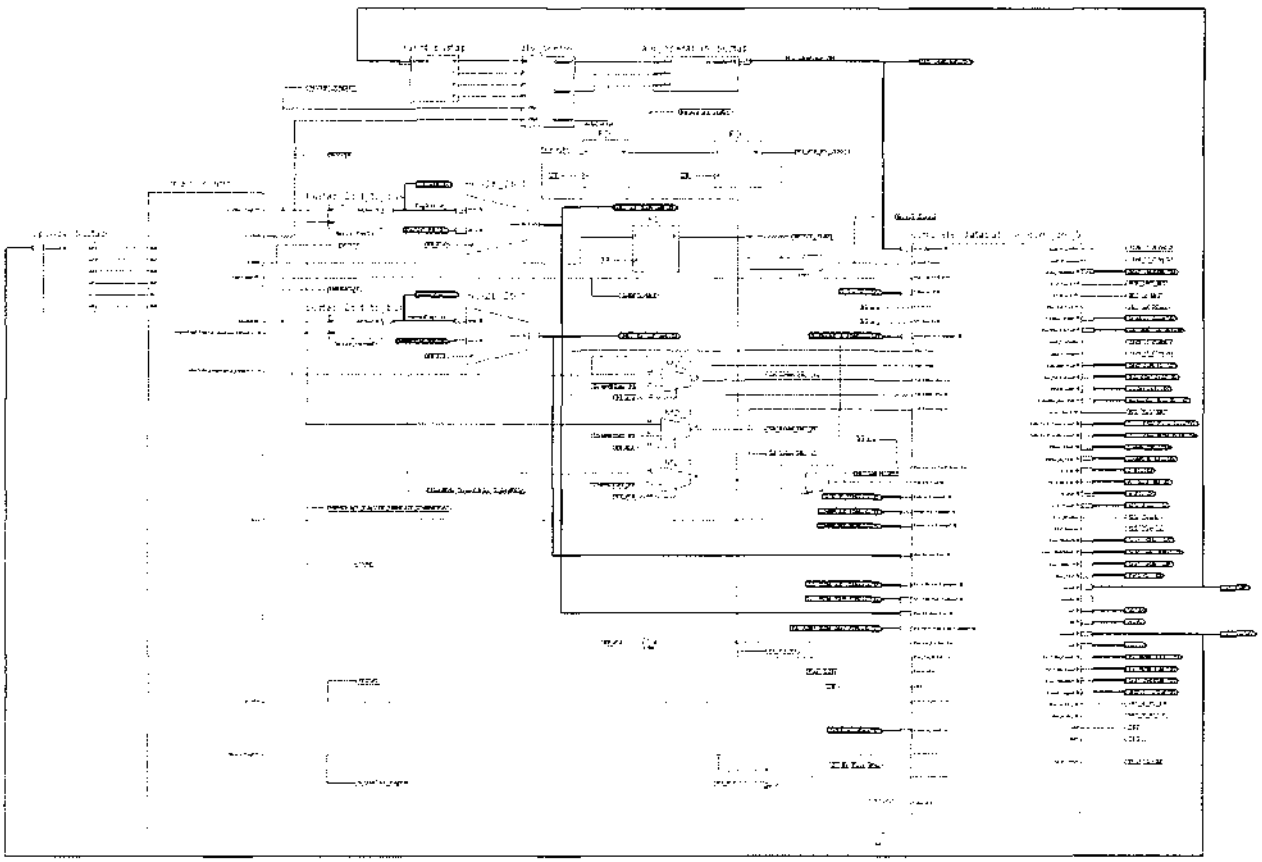

Figure 6.3 Schematic diagram design entry in Schematic Editor for the finalized MIPS R2000 microprocessor (Note: Magnified portions of this figure are shown in figures 6.3A to 6.3D that follow)

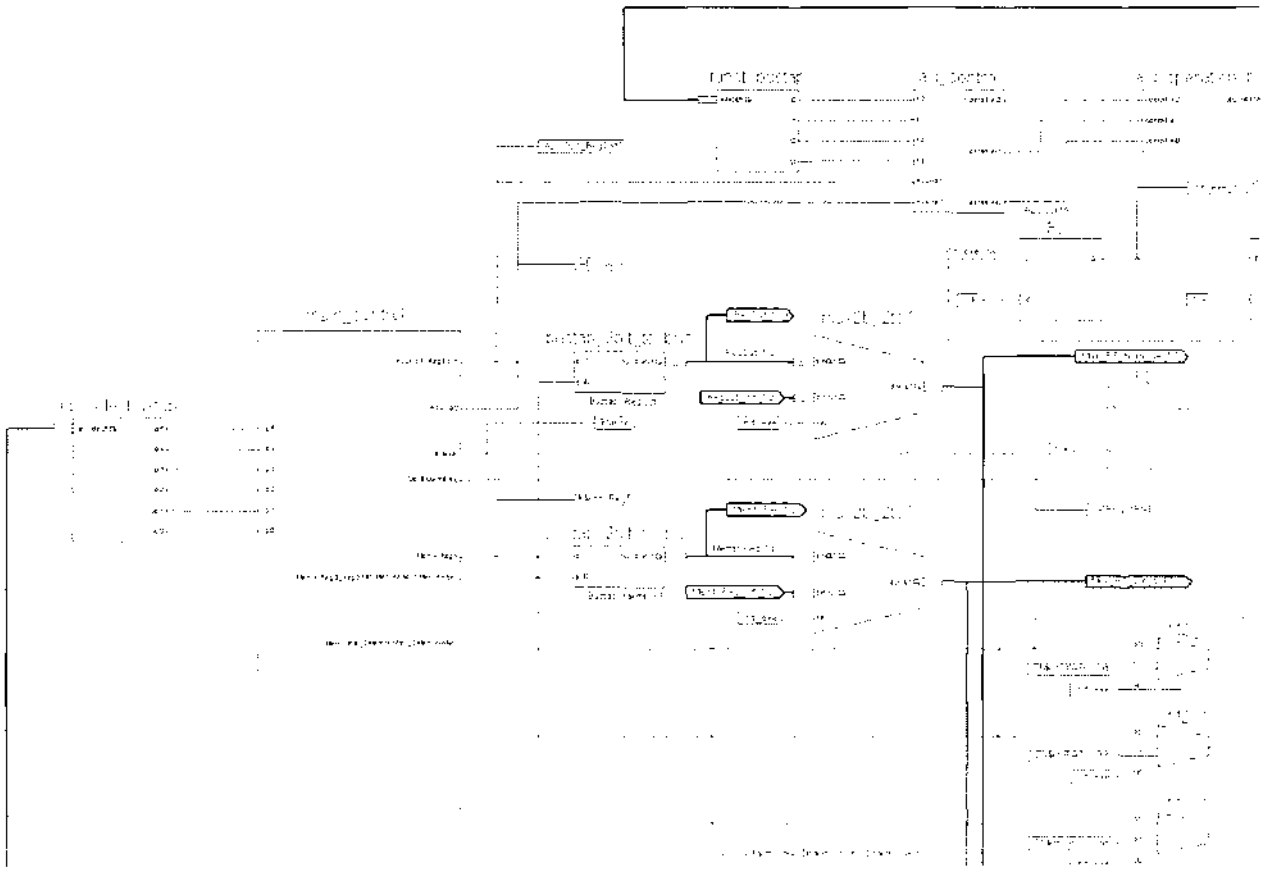

Figure 6.3-A Magnified top-left portion of Figure 6.3

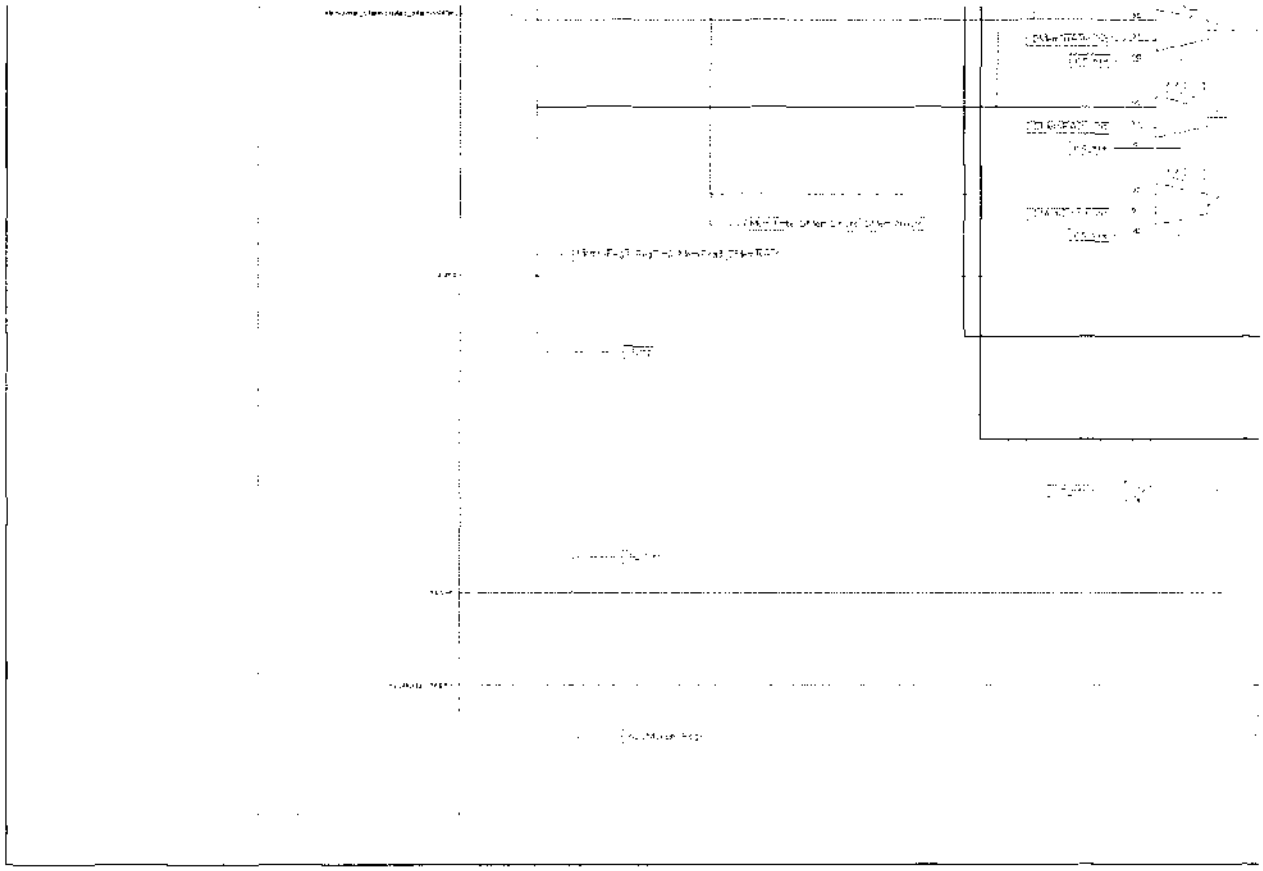

Figure 6.3B Magnified bottom-left portion of Figure 6.3

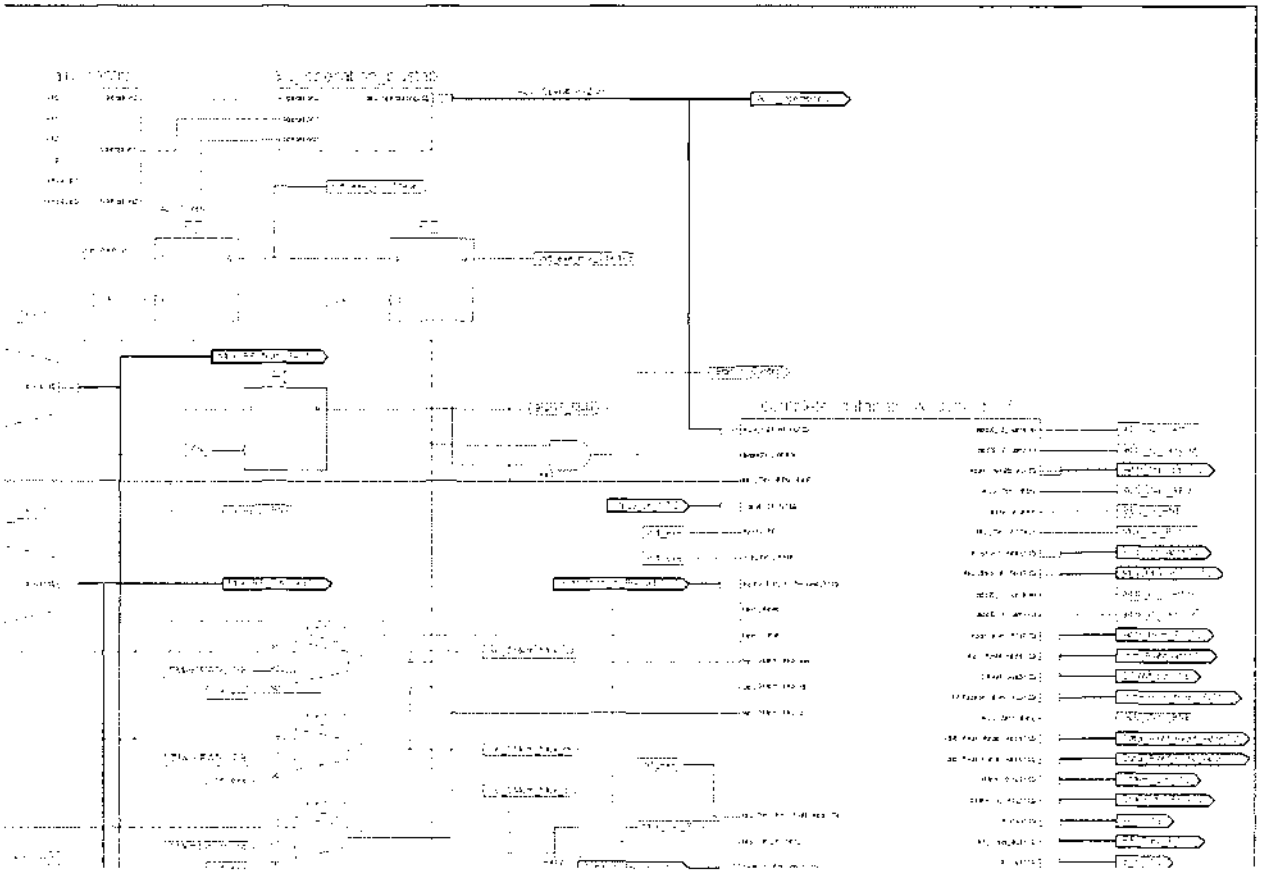

Figure 6.3C Magnified top-right portion of Figure 6.3

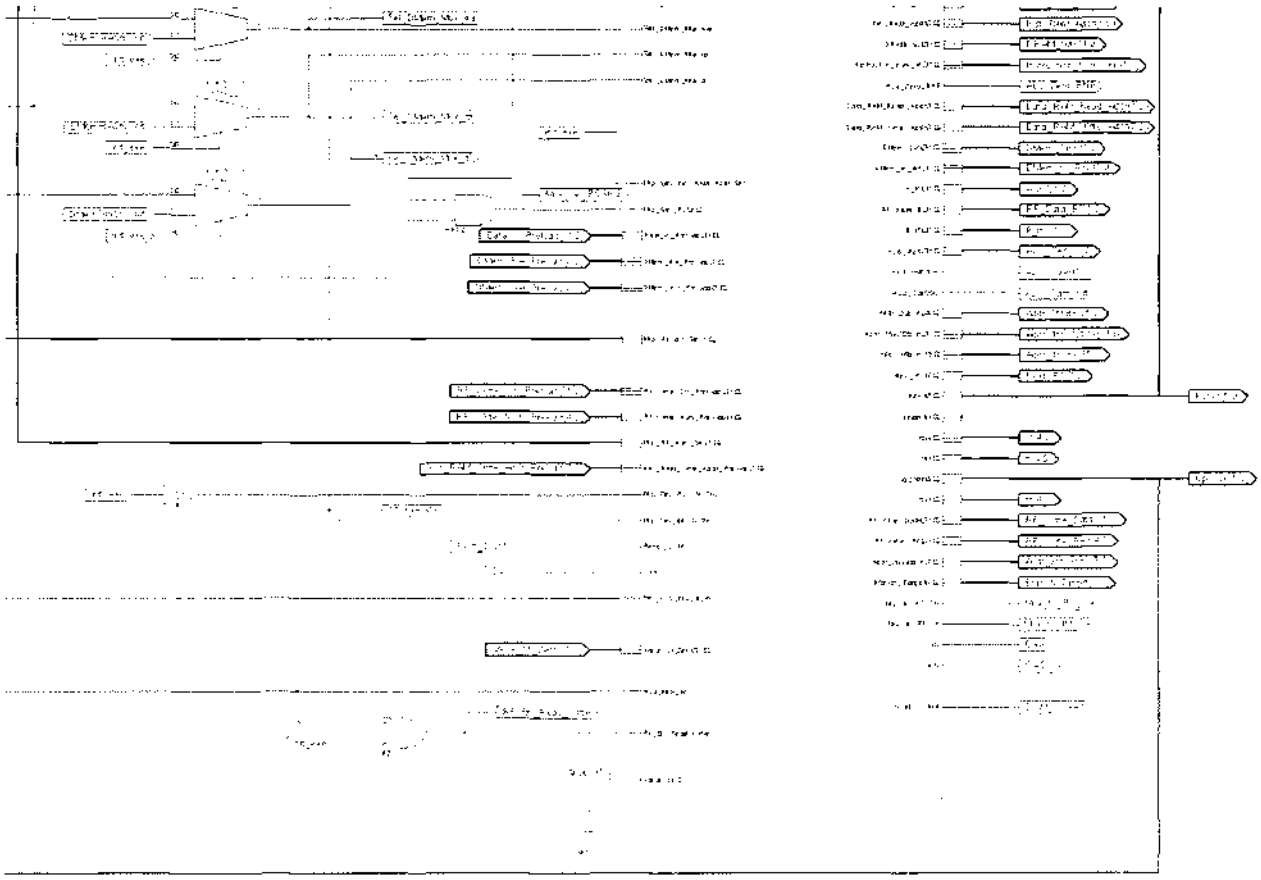

Figure 6.3D Magnified bottom-right portion of Figure 6.3

After synthesis of the schematic diagram in figure 6.3 using XST, the VHDL code was generated. This is found in Appendix D (section D.4).

➢ *Synthesis Results*

Using the Xilinx ISE synthesis tools, the hardware implementation for the finalized MIPS R2000 microprocessor, was generated. Figure 6.4 shows the resulting top level RTL symbol while figure 6.5 shows the resulting top level RTL schematic diagram. However, there is no need for delving into deeper levels of the hierarchy as these are already covered in detail in Appendices A to C.

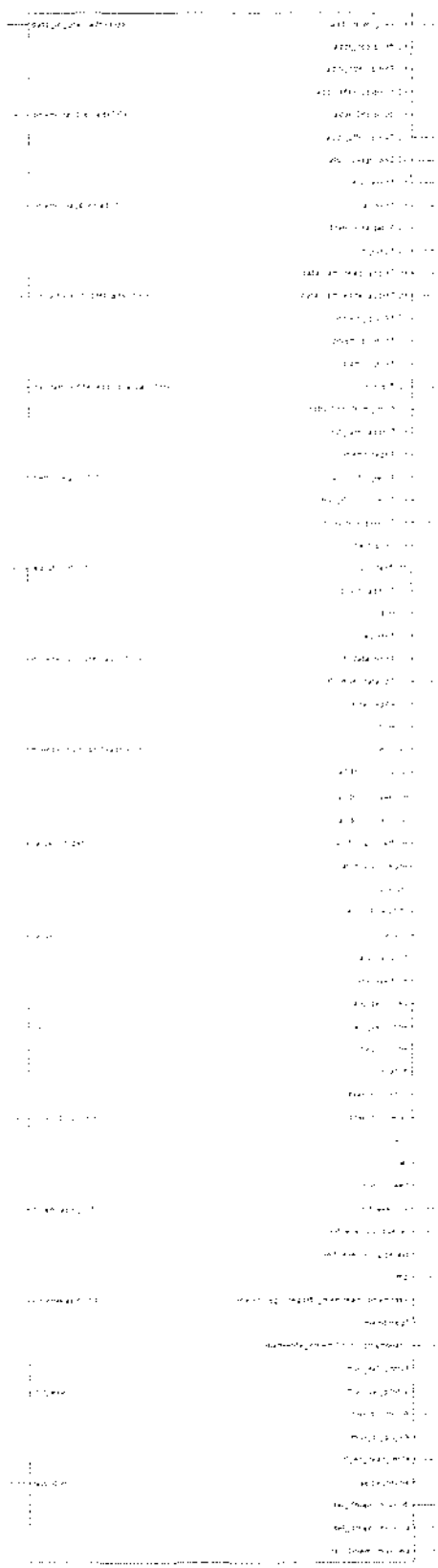

Figure 6.4 Resulting top level RTL symbol for the finalized MIPS R2000 microprocessor (Note: Magnified portions of this figure are shown in figures 6.4A to 6.4B that follow))

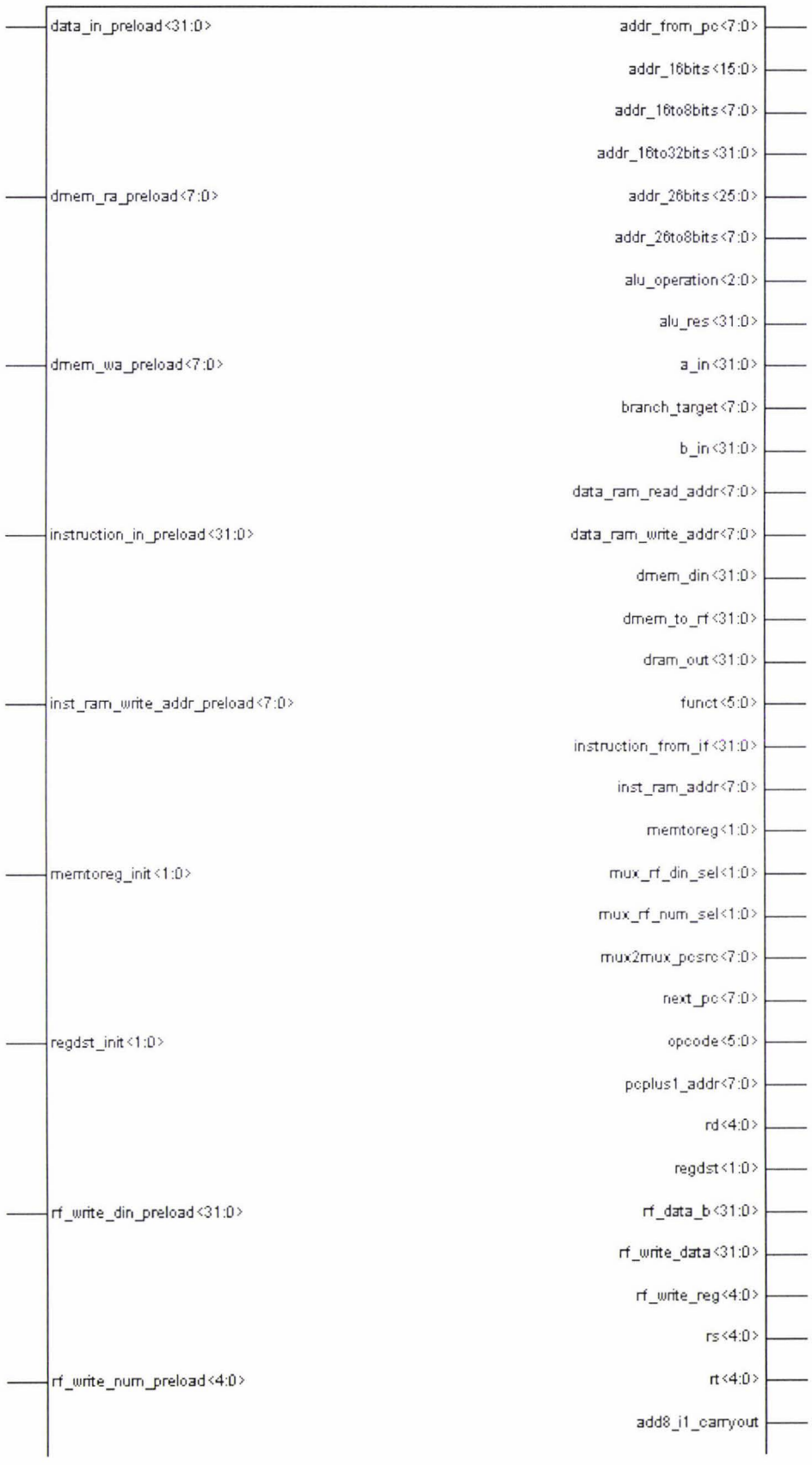

Figure 6.4A Magnified top portion of Figure 6.4

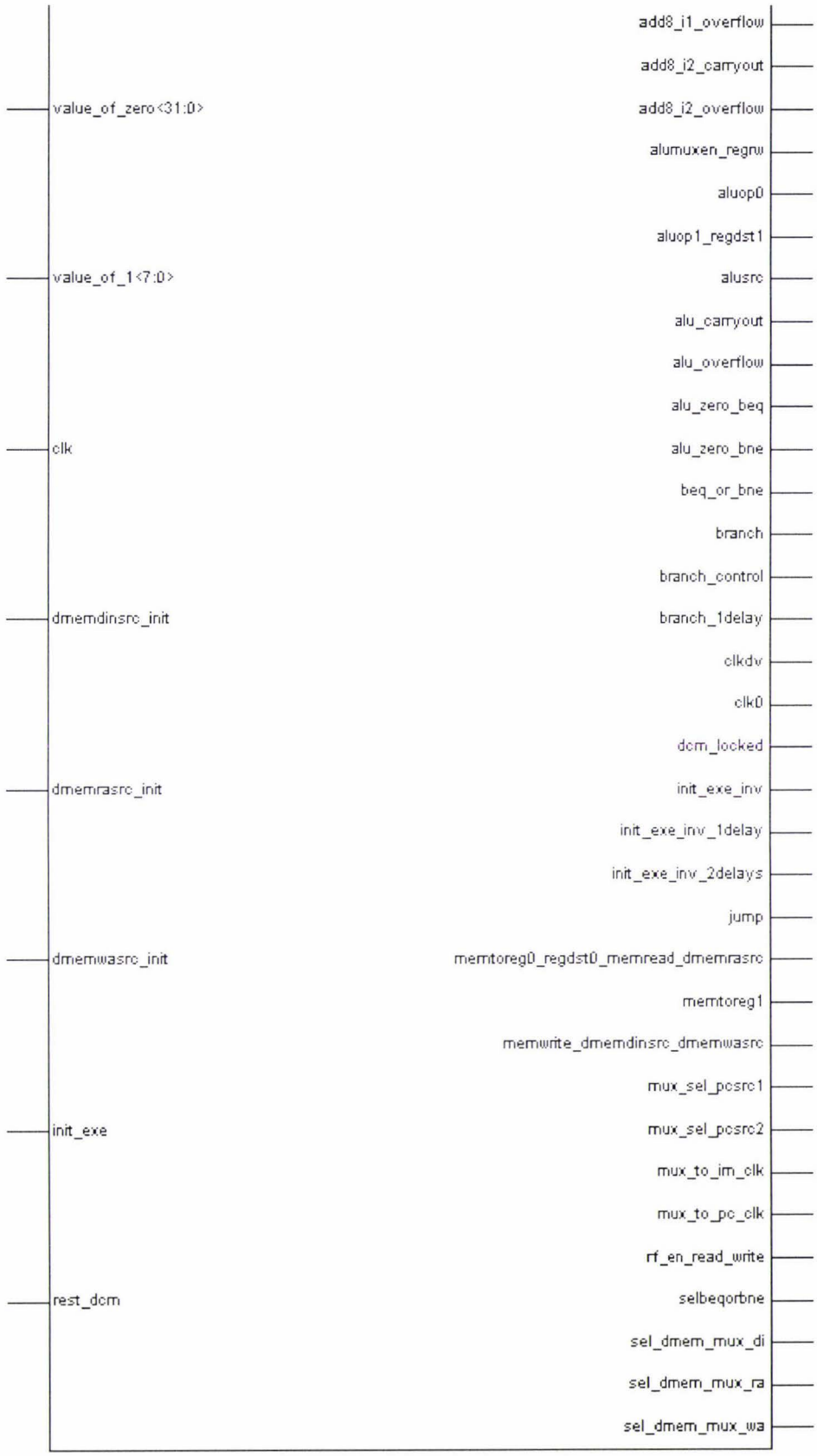

Figure 6.4B Magnified bottom portion of Figure 6.4

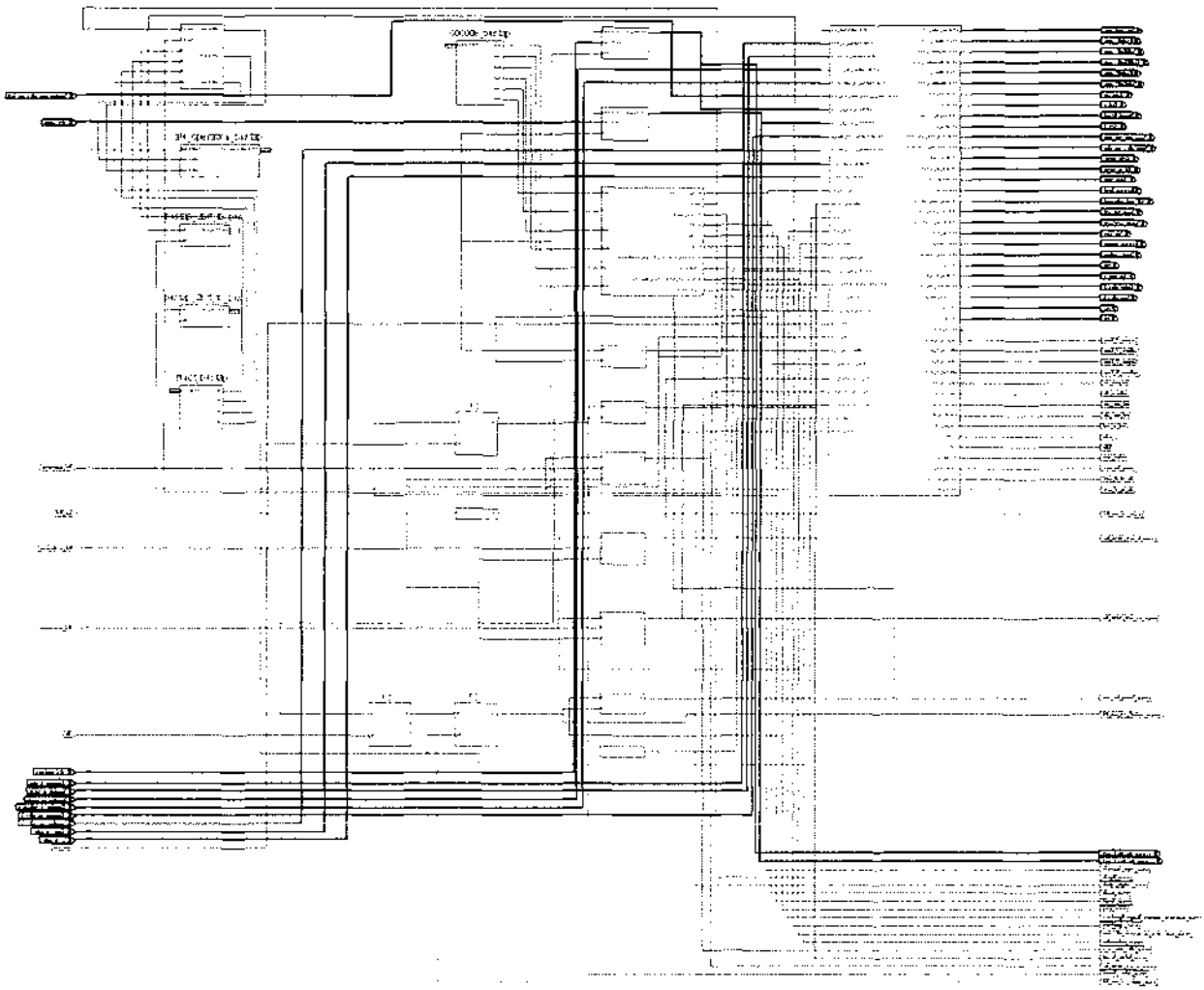

Figure 6.5 Resulting top level RTL schematic for the finalized MIPS R2000 microprocessor ((Note: Magnified portions of this figure are shown in figures 6.5A to 6.5D that follow))

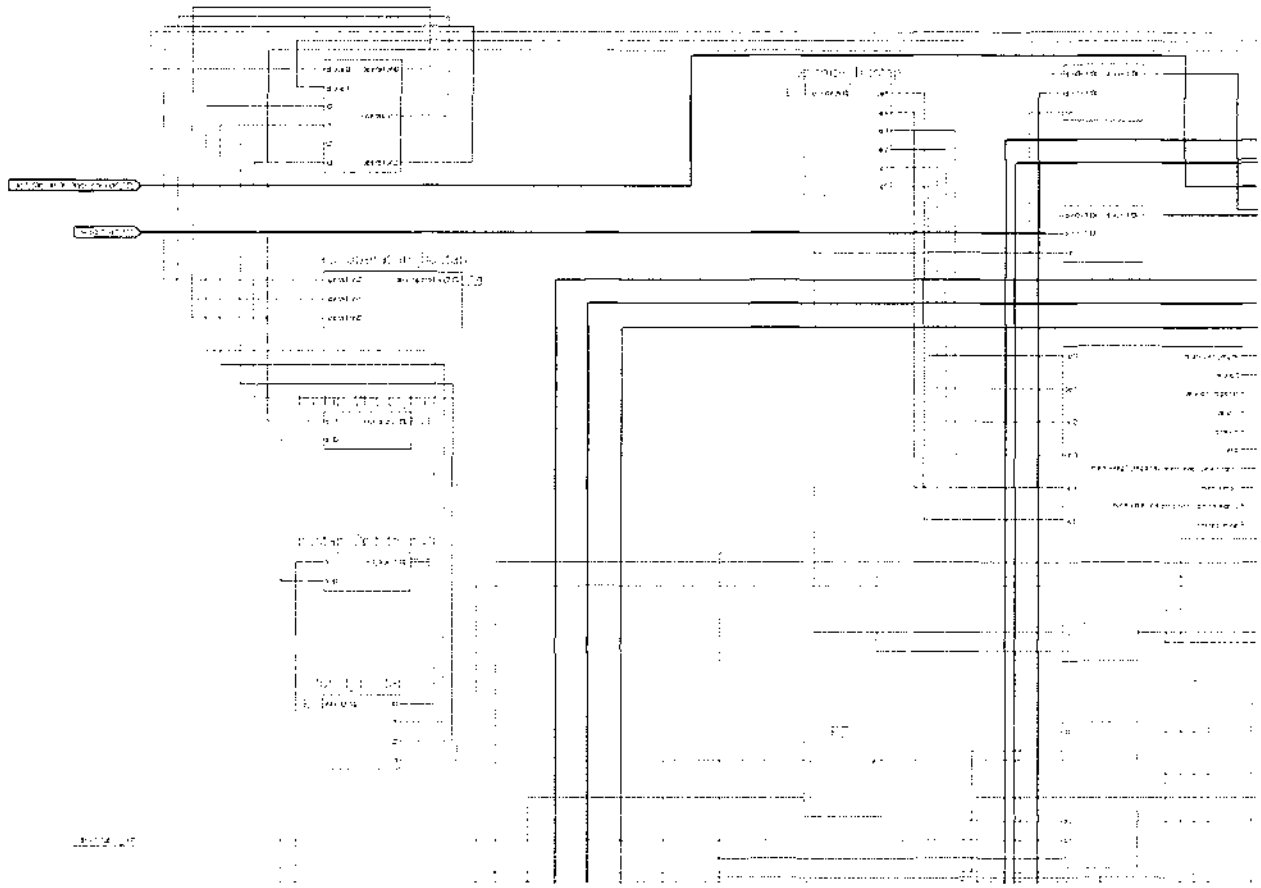

Figure 6.5A Magnified top-left portion of Figure 6.5

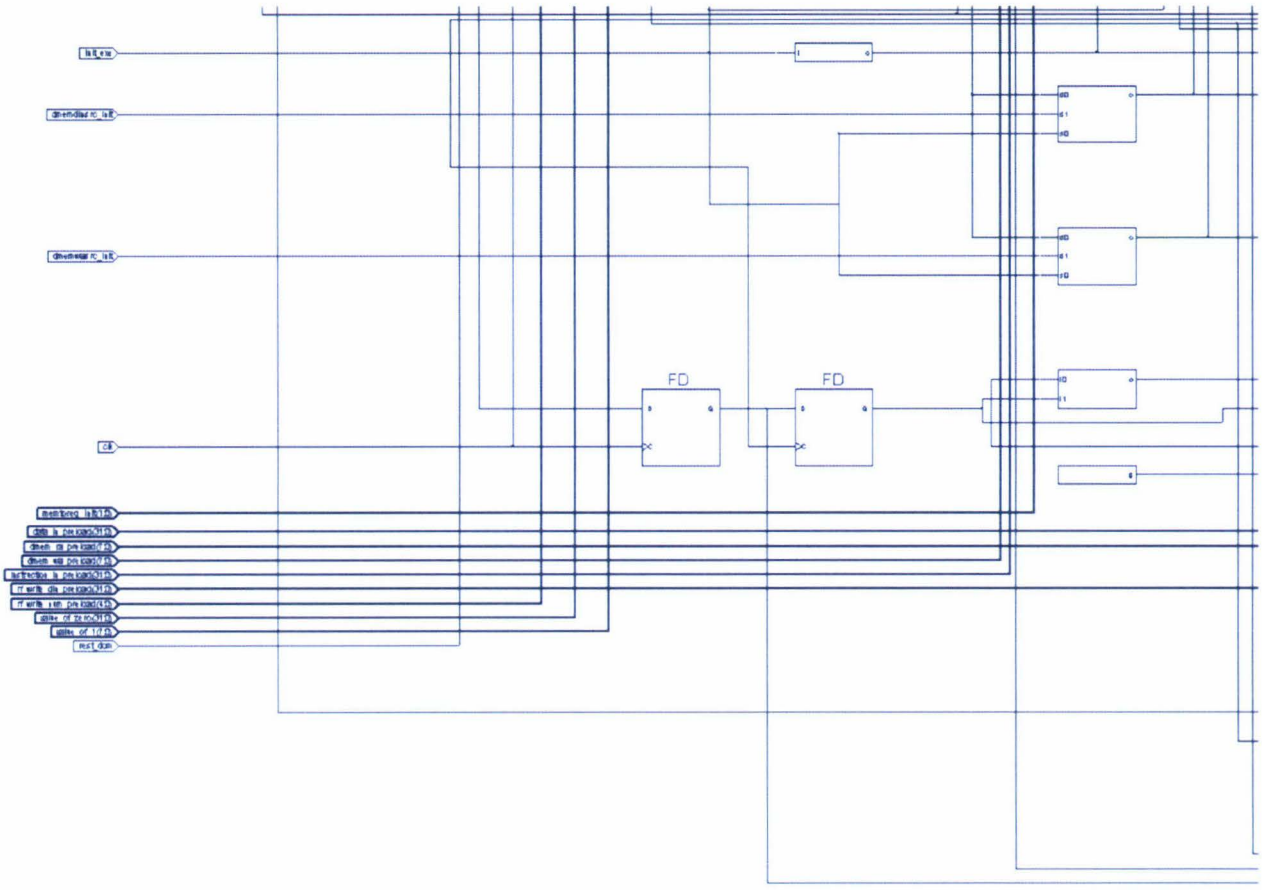

Figure 6.5B Magnified bottom-left portion of Figure 6.5

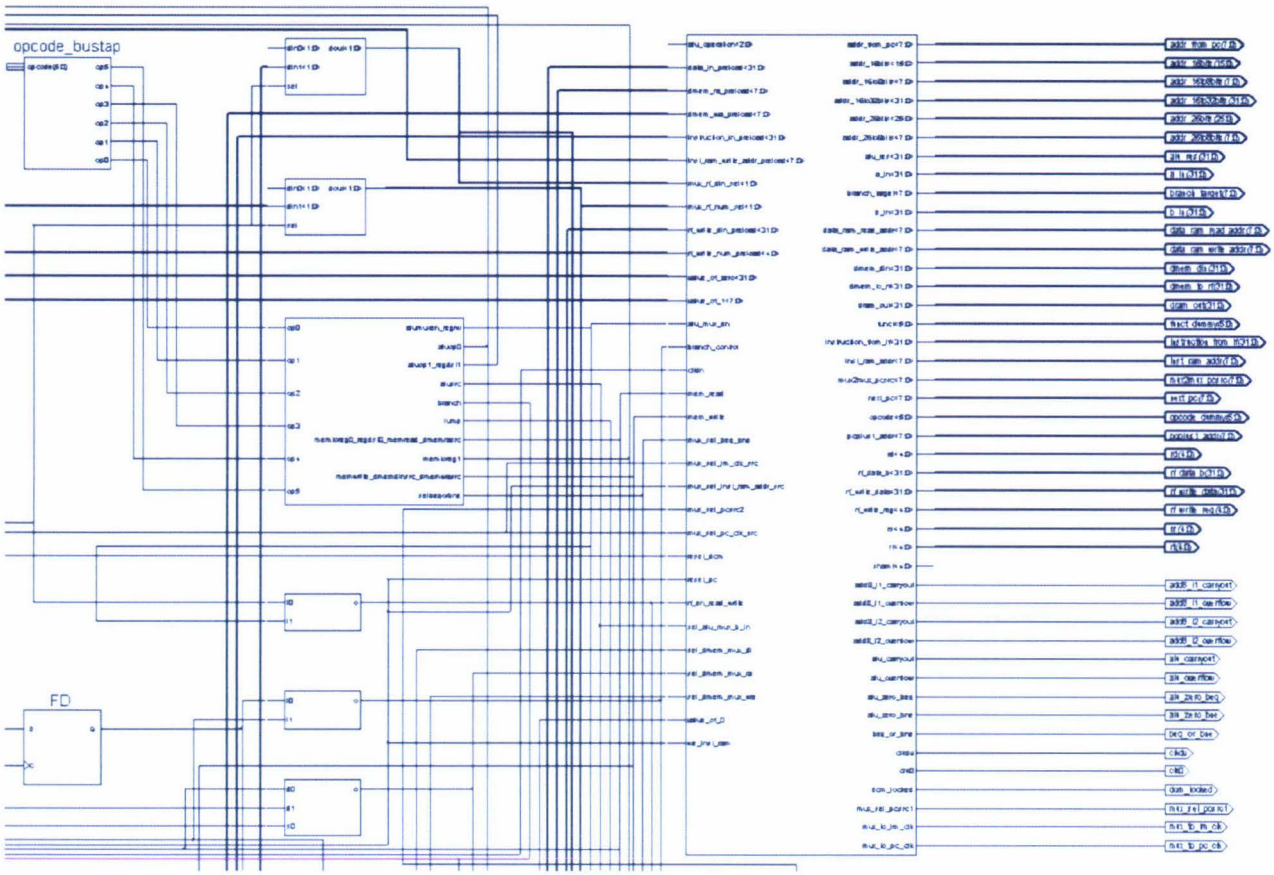

Figure 6.5C Magnified top-right portion of Figure 6.5

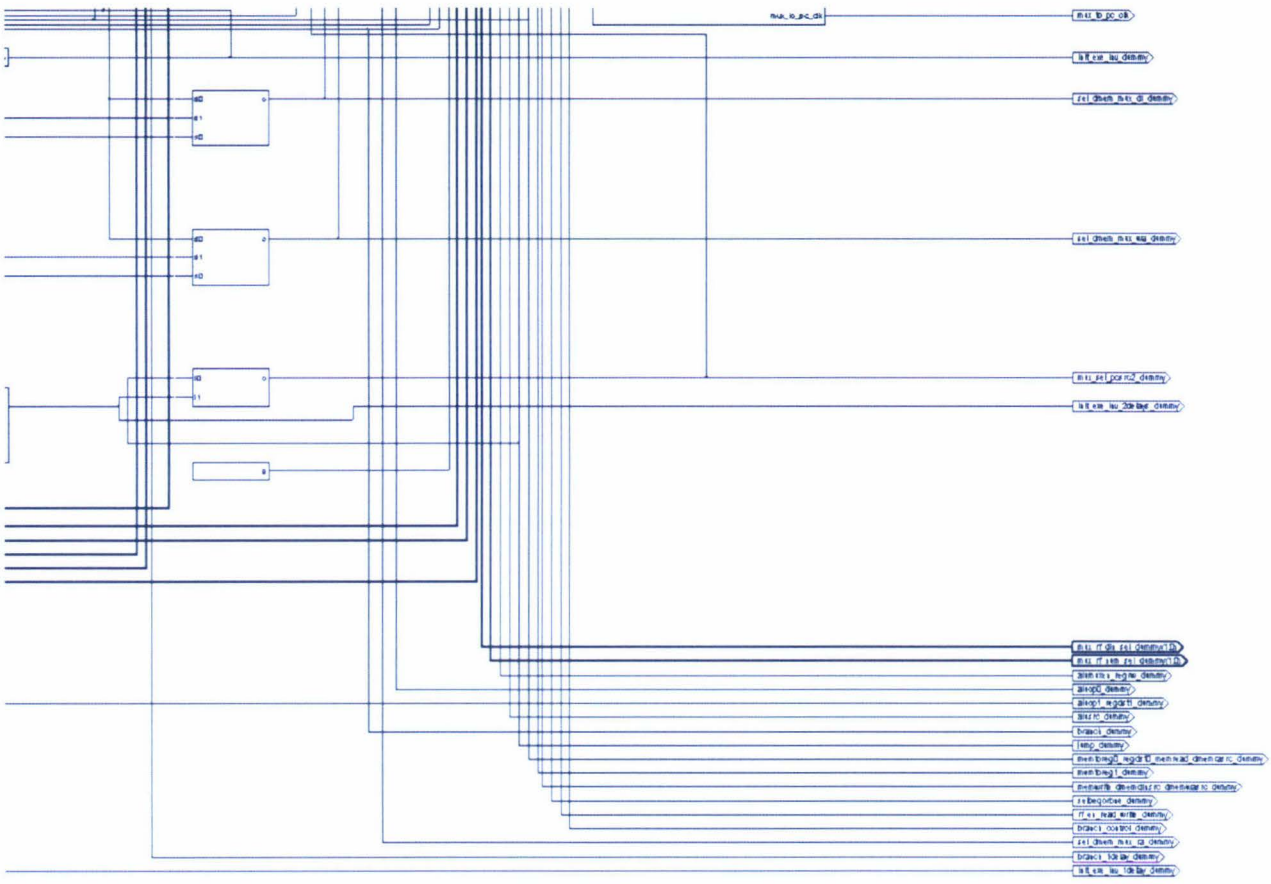

Figure 6.5D Magnified bottom-right portion of Figure 6.5

➢ *FPGA Device Synthesis Summary*

After the hardware implementation for the finalized MIPS R2000 microprocessor, using the Xilinx ISE synthesis tools, the Synthesis Report was generated. The most important FPGA Device Synthesis Statistics from this report, are shown below:

```
Design Statistics:
# IOs                              : 695

Macro Statistics:
# RAM                              : 4
#      256x32-bit dual-port block RAM: 1
#      256x32-bit single-port block RAM: 1
#      32x32-bit dual-port block RAM: 2
# Registers                        : 1
#      8-bit register              : 1
# Tristates                        : 23
#      32-bit tristate buffer      : 9
#      5-bit tristate buffer       : 4
#      8-bit tristate buffer       : 10
```

```
Cell Usage:
# BELS                              : 883
#      and2                         : 197
#      and2b1                       : 38
#      and3                         : 95
#      and3b1                       : 64
#      and4                         : 6
#      and5                         : 16
#      inv                          : 41
#      LUT1                         : 41
#      LUT1_L                       : 10
#      LUT2                         : 6
#      LUT2_L                       : 1
#      LUT3                         : 84
#      LUT3_L                       : 17
#      LUT4                         : 213
#      LUT4_L                       : 17
#      LUT4_D                       : 24
#      muxcy                        : 1
#      muxcy_d                      : 7
#      muxcy_l                      : 17
#      muxf5                        : 70
#      vcc                          : 1
#      xor2                         : 1
#      xorcy                        : 1
#      xorcy_d                      : 1
#      xorcy_l                      : 47
#      xorcy                        : 14
# FlipFlops/Latches                 : 11
#      FD                           : 4
#      FDC                          : 5
# RAMS                              : 4
#      RAM16X1S                     : 1
#      RAM16X1S_1                   : 3
# Tri-States                        : 256
#      BUFT                         : 256
# Clock Buffers                     : 2
#      bufg                         : 2
# IO Buffers                        : 695
#      IBUF                         : 174
#      ibufg                        : 1
#      OBUF                         : 468
#      OBUFT                        : 32
# DCMs                              : 1
#      dcm                          : 1
# Logical                           : 9
#      nor4                         : 8
# Others                            : 36
```

```
#      fmap                        : 36

Device utilization summary:

 Number of Slices:                   120  out of  46592     0%
 Number of Slice Flip Flops:          11  out of  93184     0%
 Number of 4 input LUTs:             207  out of  93184     0%
 Number of bonded IOBs:              695  out of   1108    62%
 Number of TBUFs:                    356  out of  23296     1%
 Number of BRAMs:                      4  out of    168     2%
 Number of GCLKs:                      2  out of     16    12%
 Number of DCMs:                       1  out of     12     8%

Timing Summary:

   Minimum period: 24.522ns (Maximum Frequency: 40.780MHz)
   Minimum input arrival time before clock: 12.513ns
   Maximum output required time after clock: 51.674ns
   Maximum combinational path delay: 18.250ns
```

➢ *Simulation Results*

Figure 6.6 shows the simulation waveforms for the test code simulated to run on this microprocessor. This code is as follows:

Instruction Memory Location No.	Instruction Loaded	Comments
0	**SLT $R7 , $R5 , $R6** ***Assembles to (A6382A)$_{hex}$***	*[$R5] = (15)$_{10}$* *[$R6] = (16)$_{10}$* *[$R5] < [$R6]=> [$R7]= 1*
1	**SW $R7 , 10 ($R5)** ***Assembles to (ACA7000A)$_{hex}$***	*[Memory[25]]= [$R7]= 1*
2	**J 100** ***Assembles to (8000064)$_{hex}$***	
3	**NOP** ***Assembles to (0000000)$_{hex}$***	*Inserting a pipeline bubble*

100	**BNE $R5 , $R6 , 10**	*[$R5] = [$R6]= (15)$_{10}$ => Branch Taken*
	Assembles to (14A6000A)$_{hex}$	*Next PC = Current PC + 1 + 10 = 101 + 1 + 10 = 112*
101	**NOP**	*Inserting a pipeline bubble*
	Assembles to (0000000)$_{hex}$	
112	**LW $R8 , 57 ($R5)**	*[$R8]= [Memory[25]]= 1*
	Assembles to (8CA8000A)$_{hex}$	

This is assembled as follows:

- SLT $R7 , $R5 , $R6

  ------ ------ ------

  rd *rs* *rt*

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00101	00110	00111	00000	101010
---	---	---	---	---	---
op=0	*rs*=$R5	*rt*=$R6	*rd*=$R7	*shamt*	*funct*=42

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000101001100011100000101010)_2 = (A6382A)_{hex}$

- SW $R7 , 10 ($R5)

  ------ ----------

  rt *offset (rs)*

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

101011	00101	00111	0000000000001010
---	---	---	---
op=43	*rs*=$R5	*rt*=$R7	*offset*=10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10101100101001110000000000001010)_2 = (ACA7000A)_{hex}$$

- J 100

  ------

  address

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000010	00000000000000000001100100
---	---
op=2	*address*=100

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(00001000000000000000000001100100)_2 = (8000064)_{hex}$$

- NOP

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000	00000	00000	00000	00000	000000
---	---	---	---	---	---
op=0	*rs*=$R0	*rt*=$R0	*rd*=$R0	*shamt*	*funct*=0

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(00000000000000000000000000000000)_2 = (00000000)_{hex}$$

This NOP (also called a *Pipeline Bubble*) is inserted following every control transfer instruction (in this case J). This extra *ClkDv* clock cycle during which time the NOP is in the pipeline, is needed by the instruction fetch unit to correctly update the program counter with the value of the instruction memory address of the next instruction (after NOP). Without NOP, the pipeline would go into non-deterministic states during this extra *ClkDv* clock cycle causing the instruction fetch unit to update the program counter with an incorrect value leading to the premature termination of code execution.

- BNE $R5 , $R6 , 10

  ------ ------ ------

  rs *rt* *offset*

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000101 00101 00110 0000000000001010

--------- ------- ------- ----------

op=5 *rs*=$R5 *rt*=$R6 *offset*= 10

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00010100101001100000000000001010)_2 = (14A6000A)_{hex}$

- NOP

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

000000 00000 00000 00000 00000 000000

--------- ------- ------- ------- ------- ---------

op=0 *rs*=$R0 *rt*=$R0 *rd*=$R0 *shamt* *funct*=0

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$(00000000000000000000000000000000)_2 = (00000000)_{hex}$

This NOP (also called a *Pipeline Bubble*) is inserted following every control transfer instruction (in this case BNE). This extra *ClkDv* clock cycle during which time the NOP is in the pipeline, is needed by the instruction fetch unit to correctly update the program counter with the value of the instruction memory address of the next instruction (after NOP). Without NOP, the pipeline would go into non-deterministic states during this extra *ClkDv* clock cycle causing the instruction fetch unit to update the program counter with an incorrect value leading to the premature termination of code execution.

- LW $R8 , 57 ($R5)

  ------ ---------------

  rt *offset (rs)*

The corresponding 32-bit assembly language instruction representation (discussed in Ch.5) is:

100011	00101	01000	000000000 111001
---	---	---	---
op=35	*rs*=$R5	*rt*=$R6	*offset*=57

In order to make debugging more manageable, the corresponding hexadecimal representation for this 32-bit instruction is:

$$(10001100101010000000000000111001)_2 = (8CA80039)_{hex}$$

An important note here for this LW instruction as part of this test code, is that it should specify the same offset value of $(10)_{10}$ as the one specified in the SW instruction (2^{nd} line in the test code above) so that when added to $(15)_{10}$ (the contents of $R5 as the base register) would yield the value of $(25)_{10}$ which is the target address in the data memory from which to load the data into the register file. Ideally, the contents of $R5 as the base register are not to be modified at all as part of running this test code. However, this is not the case here, and the offset value of $(57)_{10}$ had to be specified in this LW instruction to offset the incorrect value of $(-32)_{10}$ (the unwanted modified contents of $R5 as the base register). This is caused by the fact that my design of the register file (Appendix A) implements only one combined signal for enabling both read and write to the RF at the same time. This is a design constraint imposed actually by the FPGA chip when the on-chip BlockRAM resources are to be used for synthesizing the register file. It is recognized that in an actual register file implementation, there should be two separate control signals; one for reads, and the other for writes, to prevent the register file from being written with unwanted values during executing code. However, due to time constraints and the need to submit this dissertation by the required deadline, this design issue is recognized and noted, but the remedy of which would have to be part of future research work.

- **Conclusions:**

These resulting waveforms are in line with the expected functionality (described in detail in Appendices A to C) and prove that this finalized MIPS R2000 microprocessor is functioning as expected for this test code.

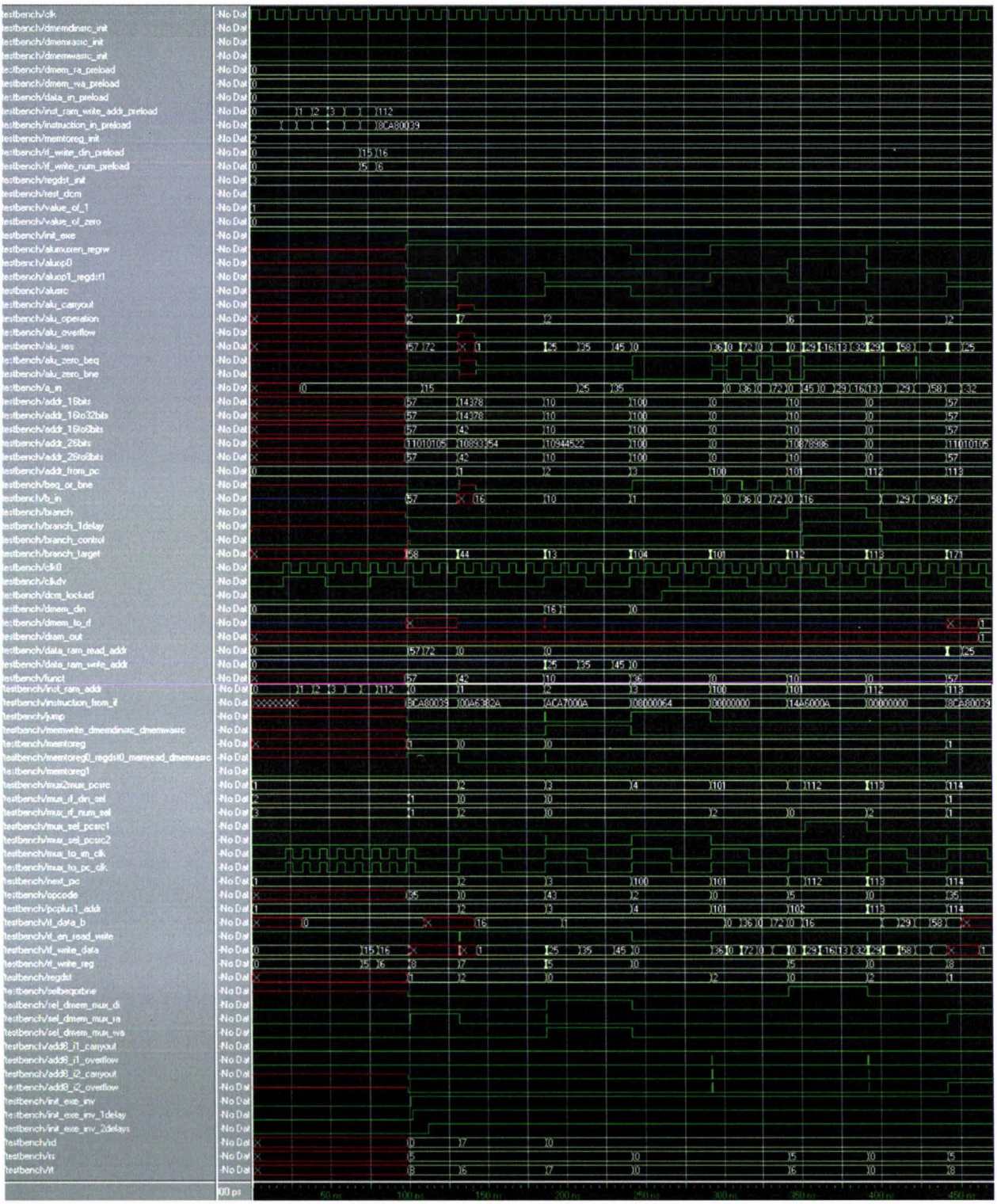

Figure 6.6 Results of simulating the finalized MIPS R2000 microprocessor for the test code (Note: Refer also to Appendix D on the Companion CD for a higher resolution version of this figure)

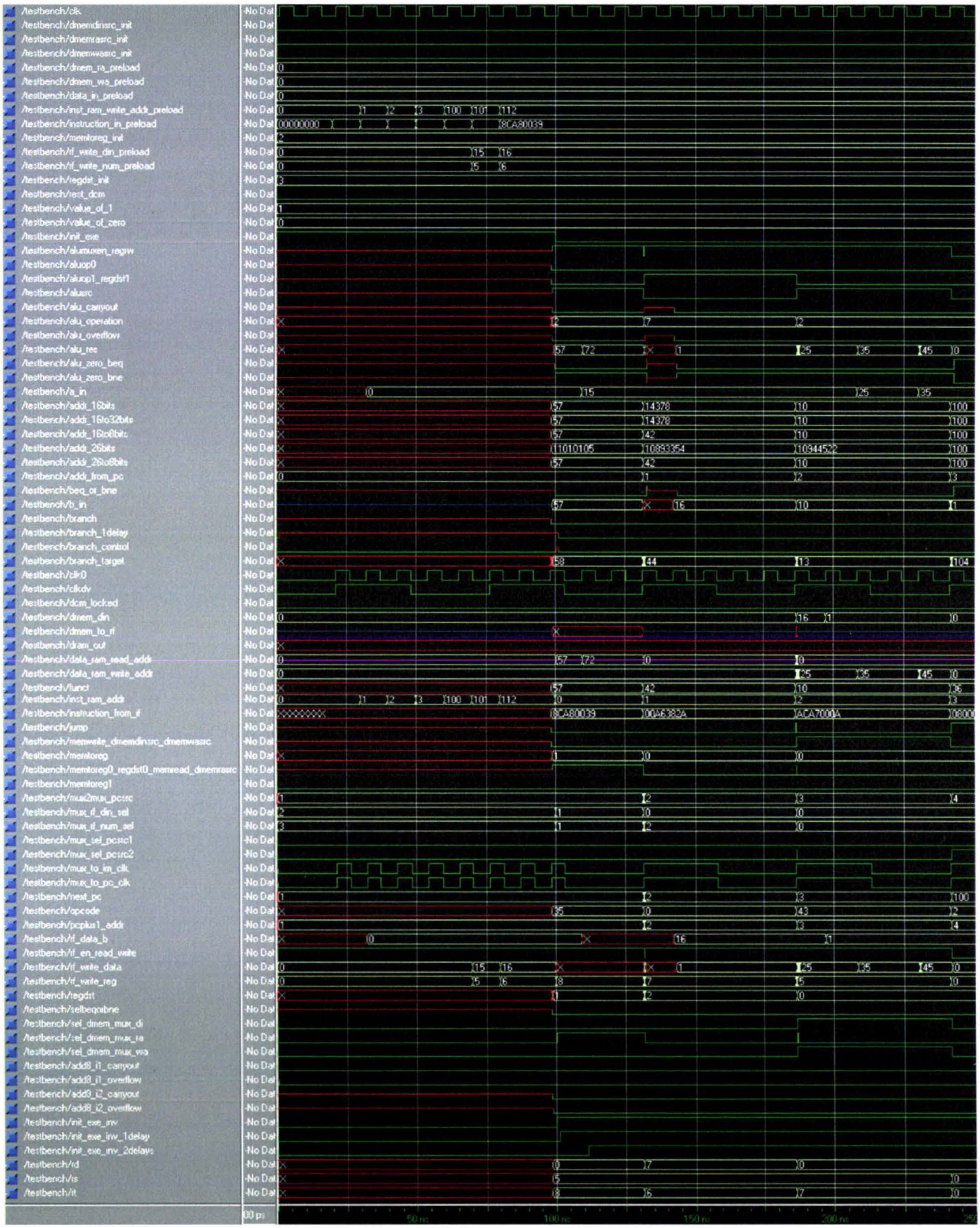

Figure 6.6A Results of simulating the finalized MIPS R2000 microprocessor for the test code (Magnified Version of Figure 6.6 – Part 1 of 2) (Note: Refer also to Appendix D on the Companion CD for a higher resolution version of this figure)

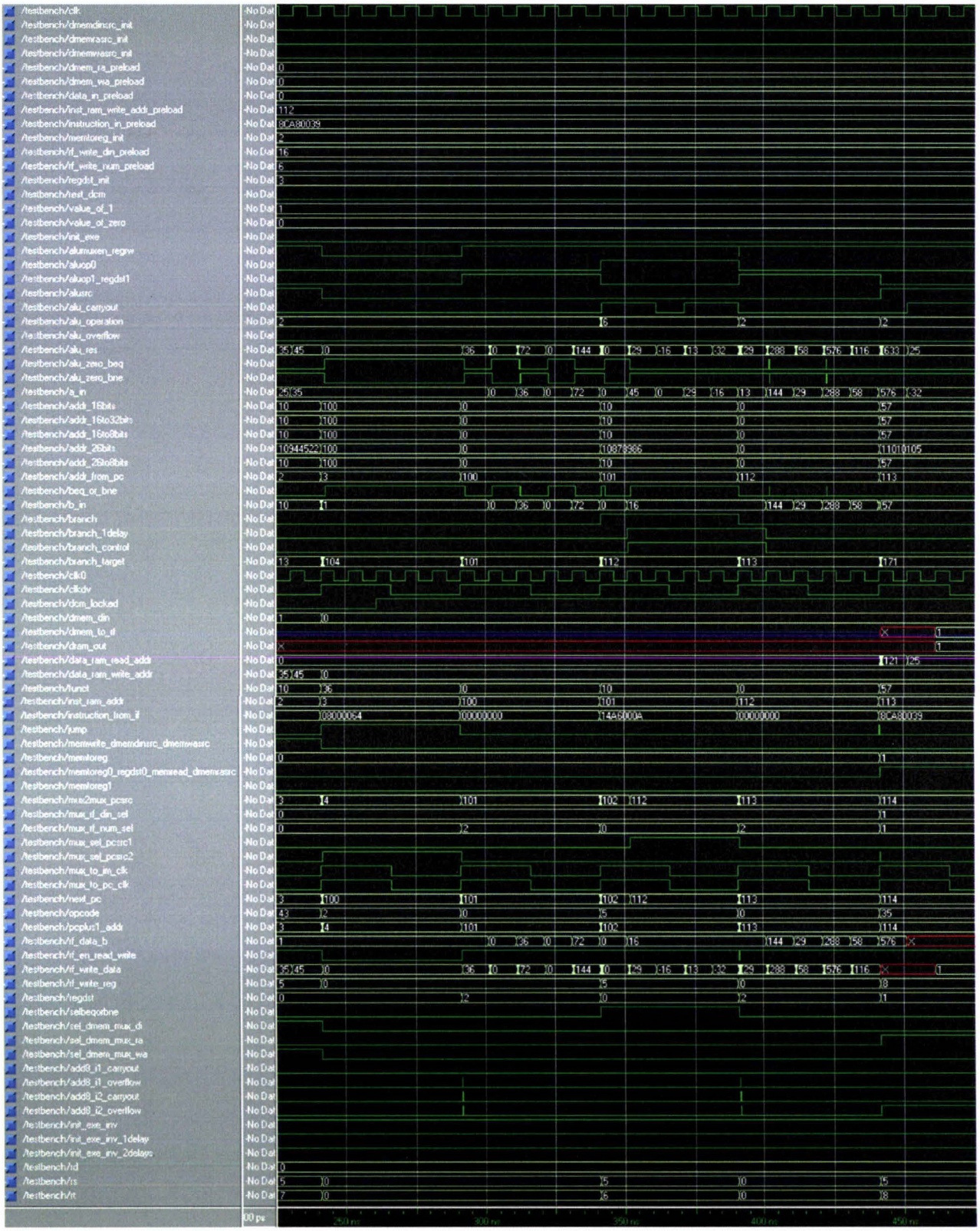

Figure 6.6B Results of simulating the finalized MIPS R2000 microprocessor for the test code (Magnified Version of Figure 6.6 – Part 2 of 2) (Note: Refer also to Appendix D on the Companion CD for a higher resolution version of this figure)

6.4 Summary and Conclusions

This chapter presented the VHDL description, synthesis and simulation of a subset of the MIPS R2000 microprocessor hardware implementation onto the Virtex-II FPGA chip. It is concluded that this design easily fitted on the FPGA chip and did function according to the design specification, with the exception of a few design glitches.

The next chapter presents the assembler/loader developed for this subset of the MIPS R2000 microprocessor synthesized in this chapter.

CHAPTER SEVEN

ASSEMBLER/LOADER FOR THE SYNTHESIZED MIPS R2000 MICROPROCESSOR

This chapter presents an unconventional way of writing an assembler/loader for the MIPS R2000 microprocessor synthesized in chapter six, using the VHDL language. This was simulated in Model Technology Inc. (MTI) ModelSim XE.

7.1 Introduction

This chapter presents the Assembler/Loader for the synthesized MIPS R2000 microprocessor. VHDL was the language of choice for writing this Assembler/Loader.

Although the use of VHDL for writing an Assembler/Loader might seem unconventional, the reason behind this approach is that VHDL is the language of choice for this research due firstly to its suitability for describing large complex digital systems like microprocessors [4] and secondly to the fact that it can also be used, however, for simulating the solution for practically any problem (just as any programming language) [4]. This is why VHDL is the language of choice for developing the assembler/loader too within the context of this research, giving rise to a 100% VHDL fully-integrated hardware/software development environment.

This chapter starts with an overview in section 7.2, thereby setting the scene for the material to follow, which is covered in section 7.3. Section 7.3 outlines the methodology implemented in writing the VHDL Assembler, while section 7.4 presents the results of assembling sample code. Section 7.5 elaborates on the VHDL Loader, and section 7.6 concludes the chapter with a summary.

7.2 Overview

Figure 7.1 shows the relationship between the VHDL assembler/loader entity and the MIPS RTL model and VHDL testbench. This is the environment in which the testing is carried out for the RTL model of the MIPS R2000 microprocessor synthesized in chapter six. This testing environment follows the following workflow (as shown in Figure 7.1):

- The VHDL RTL model for the MIPS R2000 microprocessor is created and synthesized onto the target FPGA chip. A VHDL testbench is then used to stimulate and test this RTL model. This has been achieved in chapter six and appendices A to C.

- The VHDL testbench accepts its input test vectors from the VHDL assembler/loader. This is discussed in this chapter.

An important note here is that all the components in figure 7.1 (assembler, loader, RTL model, and testbench) are implemented in software and simulate the actual finalised MIPS R2000 microprocessor system with its assembler and loader.
Three main challenges (which have been resolved, as part of this research) stemmed from this undertaking:

- Applying the same programming mechanics for standard programming languages to VHDL (which is mainly for describing hardware). An important point is that VHDL is inherently "parallel" and "concurrent" since that is the way that hardware works, and hence VHDL is not well suited to sequential tasks (in particular a stream of instructions).

- The VHDL loader is taking the place of external memory in an actual microprocessor system.

- While in MIPS architecture, register specifiers are preceded by "$" (eg. reg $t3), this "$" is reserved for internal use in VHDL, and cannot be used in the code (eg. reg t3).

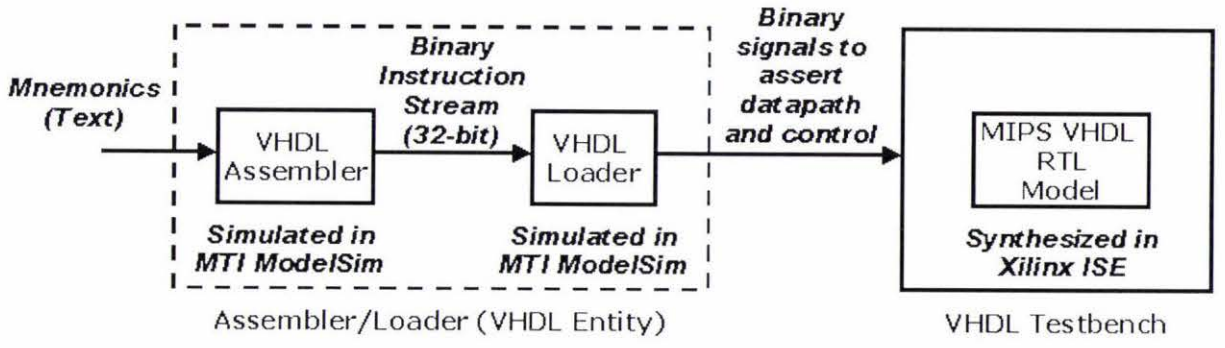

Figure 7.1 VHDL Assembler/Loader/Testbench for the synthesized MIPS R2000 RTL Model [4]

7.3 The VHDL Assembler

The VHDL assembler developed for the context of this research is based on the MIPS ISA (Instruction Set Architecture) reviewed previously in chapter 5. The functionality of this assembler can be demonstrated by way of an example instruction. The example instruction format chosen is R-format and the example instruction chosen is ADD, and are both reviewed in chapter 5. However, for facilitating convenience, the ADD instruction is shown again in Figure 7.2.

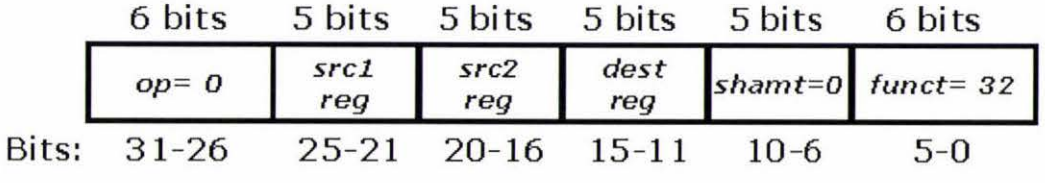

Figure 7.2 ADD instruction encoding [47, 4]

The VHDL representation for an R-format instruction packet in general is shown in figure 7.3. For example, when the VHDL assembler receives the ADD instruction packet in *mnemonics* form (human-readable form), it checks the opcode field for the character string value of "ADD" and generates accordingly a binary value of "000000" in the output bits 31-26 (the opcode for ADD is the value zero). This process is carried out over the rest of the fields in the incoming instruction packet until it is completely converted to output binary format.

```
Type R_packet_bin is record                    Type R_packet_dec is record
        op:     std_logic_vector(5 downto 0);          op:     integer;
        rs:     std_logic_vector(4 downto 0);          rs:     integer;
        rt:     std_logic_vector(4 downto 0);   =>     rt:     integer;
        rd:     std_logic_vector(4 downto 0);          rd:     integer;
        shamt:  std_logic_vector(4 downto 0);          shamt:  integer;
        funct:  std_logic_vector(4 downto 0);          funct:  integer;
```

Figure 7.3 R-format instruction encoding in VHDL [4]

7.4 Assembling Sample Code

Below is a sample loop in C [47] that was used to test the VHDL assembler (Assume $R17=g, $R18=h, $R19=i, $R20=j, $R21=Base[A]):

```
Loop:   g = g + A[i];
        i  = i  + j;
        if (i != h) goto Loop;
```

This loop translates to the following MIPS assembly code using actual physical register specifiers [1,14]:

```
Loop: add $R9,   $R19, $R21      # $R9= address of A[i]
      lw  $R8,   0($R9)          # Temp. reg $R8 = A[i]
      add $R17, $R17, $R8        # g = g + A[i]
      add $R19, $R19, $R20       # i  = i  + j
      bne $R19, $R18, -5         # go to Loop (go back 5
                                 # lines) if i not equal h.
                                 # This points to the first
                                 # "add" instruction line.
```

The VHDL assembler assembles the above 5-line code to the following corresponding machine language code [14]:

```
00000010011101010100100000100000
10001101001010000000000000000000
00000010001010001000100000100000
00000010011101001001100000100000
00010110011100101111111111111011
```

7.5 The VHDL Loader

The VHDL loader receives 1 word (32 bits) from the assembler and, accordingly, performs the following functions (refer back to figure 7.1):

- Accordingly sets the datapath and control signals for the synthesized VHDL RTL model.
- Assigns and asserts the specific memory locations within the synthesized VHDL RTL model (loading into memory).

For the sample code in the last section, the VHDL assembler/loader generates the binary signals (shown in figure 7.4) as input to the VHDL testbench.

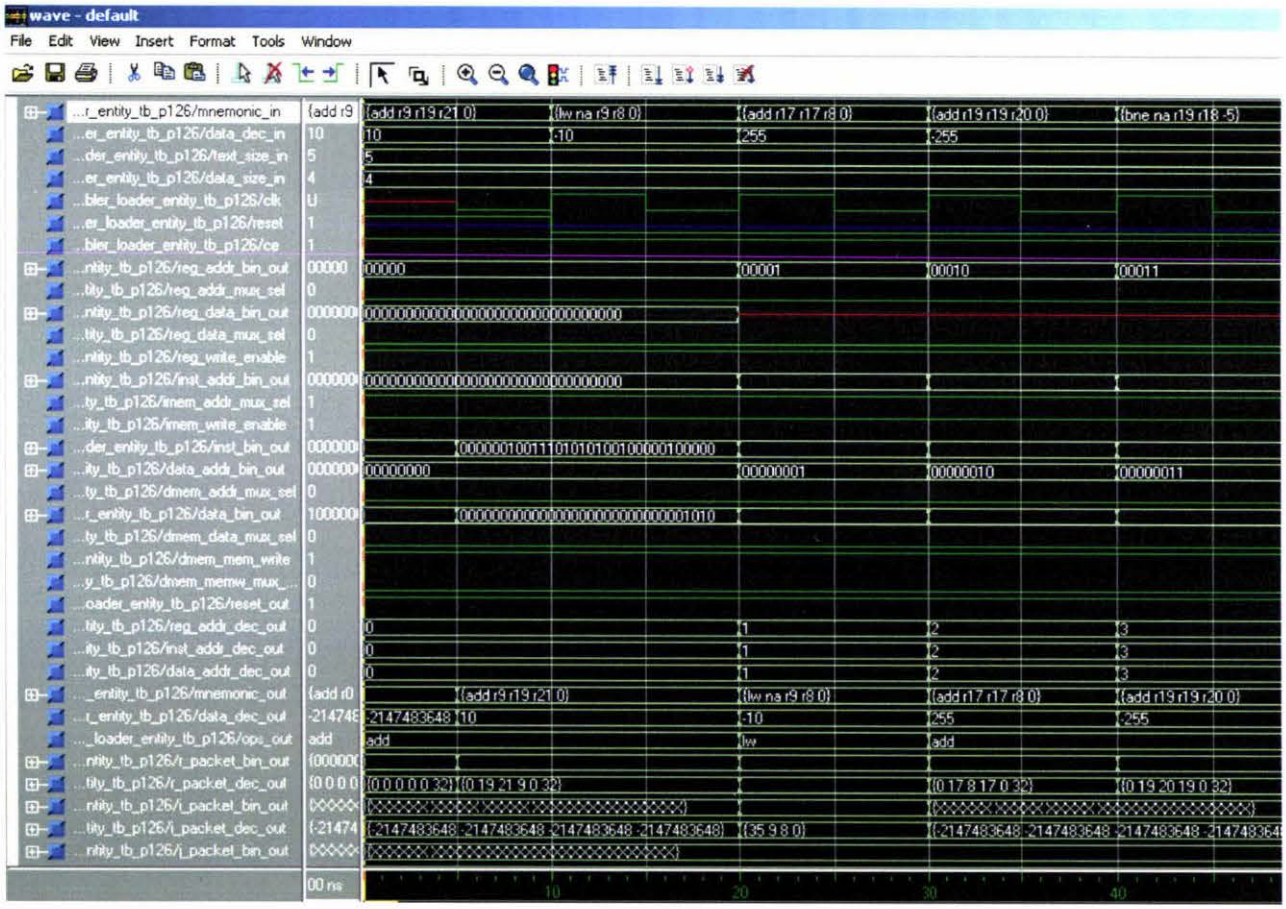

Figure 7.4 Output Signals from the VHDL Assembler/Loader as Input to the VHDL Testbench

7.6 Summary and Conclusions

This chapter presented an unconventional way of writing an assembler/loader for the finalised MIPS R2000 microprocessor, using the VHDL language. The results of assembling a sample loop written in C have been shown along with how that translates to the set of binary signals generated by the loader and to be input into the VHDL testbench.

To summarise, figure 7.1 shows that the VHDL loader provides the VHDL testbench with the binary signals necessary to assert both the datapath and control accordingly. The actual MIPS VHDL RTL model is plugged into this testbench (just like a microchip is plugged into its socket) and the testbench then provides input test vectors (signals) to the RTL model and also captures the output signals and records them into an output file for further analysis and debugging.

The next chapter introduces the VHDL description of the behavioural model for the micro-threading chip multi-CPU.

CHAPTER EIGHT

VHDL Description of the Micro-threading Chip Multi-processor

This chapter briefly describes how the micro-threading architectural add-ons and components are added to the standard MIPS architecture to build the micro-threading microprocessor and also the chip multiprocessor. The micro-threading VHDL description presented in this chapter is at a high level of abstraction as it is a behavioural description augmented with algorithms. As elaborated in chapter four, this is the first step of the hardware design process for the micro-threading microprocessor/multiprocessor and, therefore, paves the way for future research in which these algorithms and high level descriptions are utilized in designing the final micro-threading microprocessor and/or chip multiprocessor.

8.1 Introduction

Micro-threading was introduced in chapter three as a proposed architectural option for microprocessors to extract the maximum amount of instruction-level parallelism from sequential code while tolerating high memory latency and non-determinism. This chapter presents a diagrammatic description of a VHDL-based behavioural model for micro-threading applied to the five-stage MIPS pipeline. This model is the starting point for the hardware synthesis and simulation of the micro-threaded architecture, to be undertaken in future research work.

It is worth mentioning here that the contents of this chapter resulted in a refereed publication in the conference proceedings of an international conference [3].

In this chapter, the VHDL behavioural description for micro-threading is described in section 8.2. Section 8.3 concludes the chapter with summary and conclusions.

8.2 Behavioural Description in VHDL

8.2.1 Micro-threading Chip Multi-processor (CMP)

The VHDL model for the behavioural functionality of the micro-threading architecture is based on the idea of representing building blocks of the multi-processor chip as boxes (components) that communicate with each other via signals. These signals could either be of a standard VHDL type (eg. integer) [75, 11], or enumerated user-defined (eg. composite, record, packets) [75, 11, 49]. This model is based on the Abstract Machine (ABM) pipeline model by Reese [49].

The micro-threading approach can be applied to any standard RISC architecture. In this case, it has been applied to the standard MIPS R2000 architecture described in detail in [47, 48] and synthesized and simulated in Part Two of this thesis. Figure 8.1 shows the top-level abstract view for an example micro-threading multi-processor chip accommodating two Processing Units (PU1 and PU2). Actually, the micro-threading CMP is proposed to be quite scalable [33] and can host a large number of PUs. As shown in Figure 8.2, the PUs themselves are the modified MIPS pipelines. Three new components were added outside the pipelines and are shared across all PUs (as shown in Figure 8.1):

- *Global Continuation Queue (GCQ).* This holds thread descriptors. There is one GCQ per CMP chip, thereby sharing the GCQ among all PUs.

- *Register Allocation Unit (RAU).* This unit performs the dynamic register allocation.

- *Global Register File (GRF).* This contains the registers shared among dependent micro-threads (whether for shared-parameter passing or for synchronisation).

8.2.2 Micro-threading Processing Unit (PU)

Figure 8.2 depicts the internal components of each of the PUs referred to in Figure 8.1, and shows how the micro-threading architecture is still based on the 5-stage generic RISC pipeline implemented in MIPS [47, 48]. The ports interfacing this PU entity to the other components (which were shown in Figure 8.1) are displayed in *italics*. The new component added here is the *Local Continuation Queue (LCQ)*, which is local to every pipeline on the multi-processor chip. The LCQ holds the information and states relating to all the micro-threads scheduled to run (till completion) on that specific PU. The *Local Register File (LRF)* is also held locally within each PU.

8.2.3 Micro-threading Dynamic Register Allocation

For micro-threading to perform dynamic register allocation, four classes of registers are defined in this architecture [32]:

- *Global* to all micro-threads. These are located in the GRF and are denoted by *$G0*, *$G1*, etc. Global registers are allocated statically by the compiler during subroutine invocation.

- *Local* to one instance of a micro-thread. These are located in one PU's LRF and are denoted by *$L0*, *$L1*, etc. Local registers are allocated when an instance of a family of micro-threads is dispatched dynamically to a PU.

- *Shared* between two and only two micro-threads. These are located in the GRF and are denoted by *$S0*, *$S1*, etc. Shared registers are allocated when an instance of a family of micro-threads is dispatched dynamically to a PU.

- *Dependent* on a prior micro-thread. These are located in the GRF and denoted by *$D0*, *$D1*, etc. Dependent registers are allocated to the micro-thread that this current micro-thread is dependent on.

In figures 8.1 and 8.2, source registers *src1* and *src2* and the destination registers *dest* are also composite enumerated VHDL data types. They are of format *record* and consist of a register identifier *reg_id* (L, G, D, S) and a register number *reg_no*.

8.2.4 Micro-threading Algorithm

The algorithm for the execution model has been described in detail in [32, 33]. Suffice it to say, once allocated to a given PU, a micro-thread instance (dispatched from the RAU) comprises a thread pointer *tp* and a base register for the Local, Shared and Dependent registers (*L-base*, *S-base* and *D-base*). The thread runs to completion on the PU that it is allocated to. This does not lead to load imbalance as micro-threads are expected to be very short in length and not maintained as placeholders for repetitive computation. This is possible, as micro-thread creation is so inexpensive [32].

8.2.5 VHDL Description of Micro-threading Components

The VHDL behavioural model described in this chapter specifies the functionality (in terms of behaviour and specifications) for the main blocks in the VHDL behavioural diagrams (figures 8.1 and 8.2) for the micro-threading CMP. The model implements a distributed control concept, where signal packets communicate both data and control between the different components.

- *Global Continuation Queue (GCQ)*

  The GCQ, being global to all PUs, can receive create-thread (thread pointer) signals from multiple PUs in the same clock cycle (figure 8.1).

- *Register Allocation Unit (RAU)*

  The RAU performs the dynamic register allocation by receiving the thread allocation information from the GCQ, checking its internal allocation tables for availability of resources (registers to be allocated) as shown in figure 8.1.

- *Local Continuation Queue (LCQ)*

  The LCQ local to each PU receives the descriptor for the thread scheduled to run on that pipeline (figure 8.2).

- *Instruction Cache (ICache)*

  The state of a micro-thread can be used to determine a pre-fetch and replacement strategy for the ICache [33]. Upon arrival into the LCQ, a micro-thread may be in the waiting state because its code is not in ICache (or because it is waiting on a register for synchronization). A request is then made to ICache to

pre-fetch instructions for that thread (figure 8.2). The request is acknowledged either immediately or when it has been satisfied by a higher level of memory hierarchy. The ICache acknowledge signal changes the thread's state to *ready*.

- *Instruction Fetch (IF)*

At any time, there is only one micro-thread, which is in the running state (initially, this is the main thread) [33]. This running threads's LCQSlot, program counter, and base addresses (*l_base*, *s_base*, *d_base*) are all held in the pipeline's program counter (this is within the IF stage). As shown in figure 8.2, when the running micro-thread encounters an instruction tagged for a vertical transfer (VT) or kill, the IF logic requests a context switch from the LCQ. As a result, one of the ready threads is selected as running and its state is passed to the program counter.

- *Instruction Decode & Register Read (IDRR)*

The IDRR stage receives the modified instruction packet (*tag*, *opcode*, *dest*, *src1*, *src2*, *immv*, *LCQSlot*) from the IF stage (figure 8.2). Then, it sends out a request to the appropriate register file (LRF and/or GRF) to retrieve the values of the register contents, i.e. reads in *src1* register and retrieves and provides its contents; *src1_v* to the next stage (Execute).

- *Global Register File (GRF)*

The GRF fills in the missing register values from the instruction packet by receiving a request from the IDRR stage to provide the values of the register contents (figure 8.1). GRF reads in *src1* register and retrieves and sends its contents (*src1_v*) back to IDRR which in turn includes these retrieved values into the execute packet which is sent to the next stage (EX). GRF also receives write-back values from WB stage and writes these values into the specified registers. Additionally, GRF receives an initialise signal from RAU to allocate registers. The register classes held in the GRF are the Global (*$G*), Shared (*$S*), and Dependent (*$D*) registers.

- *Local Register File (LRF)*

The LRF is similar in concept to the GRF, except for the fact that it does not receive the *pu_id* (identifying the originating pipeline) as part of the data request, as it is local to that PU which initiated the request (figure 8.2). Only local (*$L*) registers are held in the LRF.

➢ *Execute (EX)*

The EX pipeline stage performs the necessary ALU operations on the incoming packet (*tag*, *opcode*, *dest*, *src1*, *src1_v*, *src2*, *src2_v*, *immv*, *LCQSlot*). The slot reference (*LCQSlot*) is available at the EX stage (as well as at all other stages in the pipeline). When a vertically transferred instruction is resolved (i.e. a branch target is calculated), the *LCQSlot* along with that target branch, are both passed back to the LCQ as part of the *wakeup* signal [33].

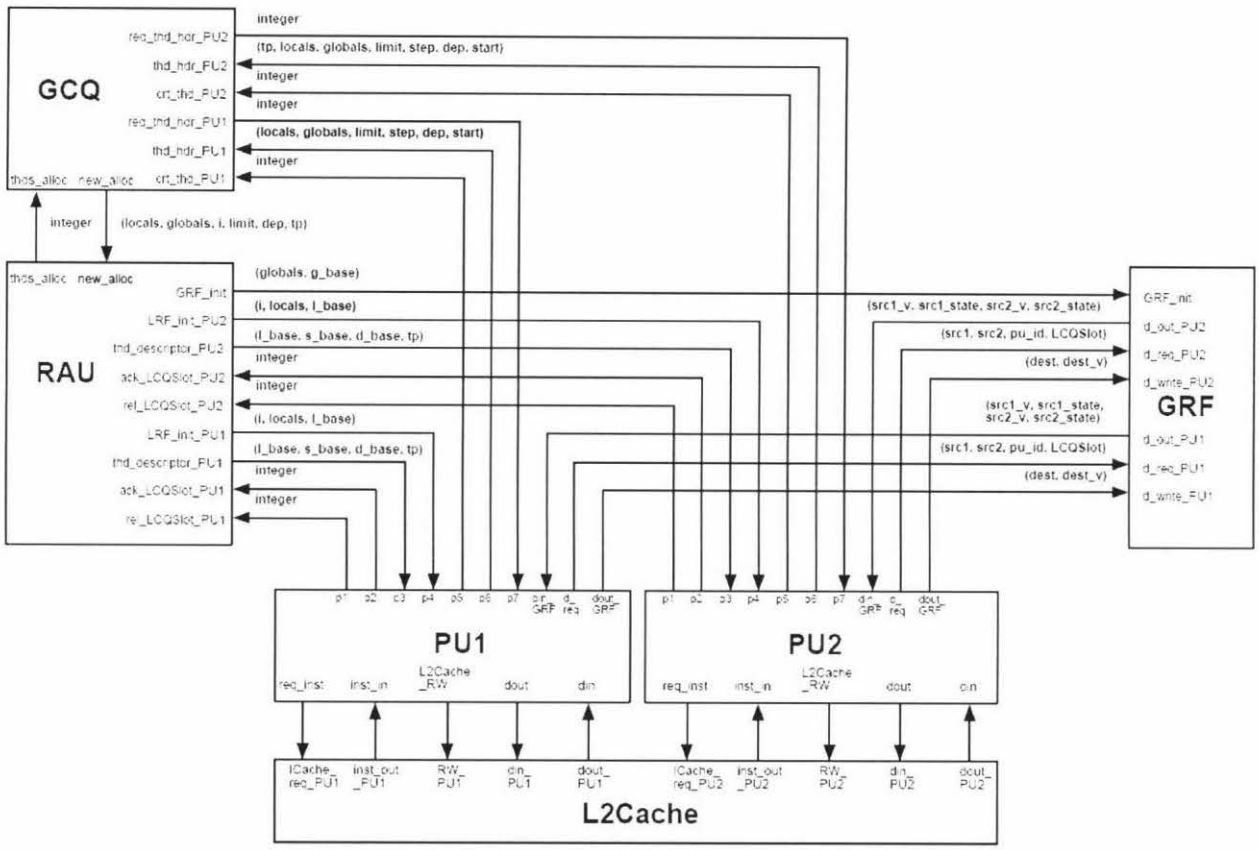

Figure 8.1 Top-level hierarchical VHDL representation of a dual-PU micro-threading CMP.

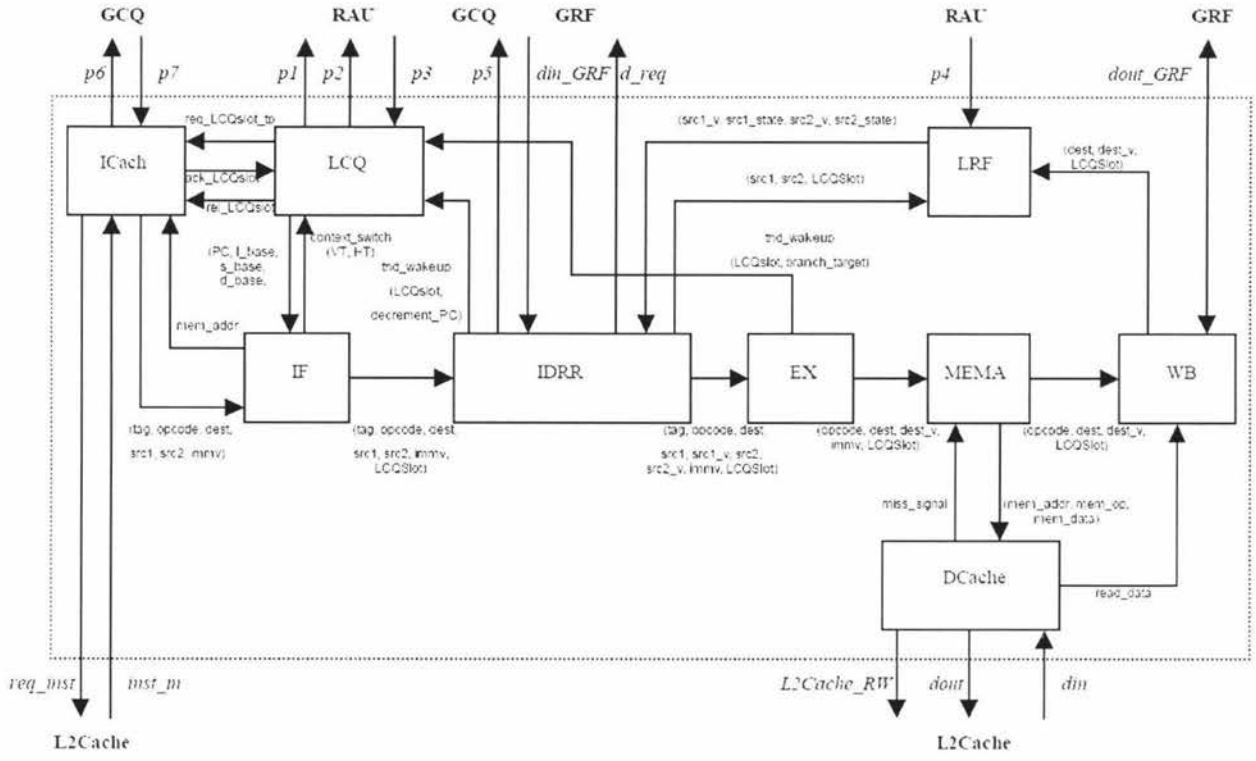

Figure 8.2 Top-level hierarchical VHDL representation of a micro-threading PU.

8.3 Summary and Conclusions

This chapter presented the last piece of research work aimed for this thesis by outlining the high level VHDL behavioural model of the micro-threading architecture.

The next chapter concludes this thesis with the summary, conclusions, and future work.

CHAPTER NINE

Conclusions And Future Work

This chapter concludes this thesis by reviewing the summaries of the key points from the previous chapters along with the important areas of research covered by the thesis. Conclusions are drawn and further areas of enhancement and future research work are listed.

9.1 Thesis Summary

This thesis presented the outcome of research mainly into two areas of the computing technology: microprocessor and multi-processor architectures (specifically from the perspective of how differently they tolerate highly-latent and non-deterministic events), and the hardware design of complex digital systems containing both datapath and control (particularly microprocessors).

As a result, the key achievements of this work are based on three important areas of research investigated and covered in this thesis:

- The problems associated with tolerating highly latent and non-deterministic events in existing microprocessor and multi-processor architectures have been recognized. This was surveyed in chapter 2.

- The high level behavioural VHDL (Very High Speed Integrated Circuit Hardware Description Language) description of the novel vector micro-threading chip multi-processor architecture, which is proposed to efficiently tolerate such high latency and non-determinism. The starting point for the design of this micro-threading architecture is the popular MIPS RISC (Reduced Instruction Set Computing) processor architecture.

- The hardware implementation involving the VHDL description, synthesis and simulation of the MIPS R2000 RISC microprocessor onto an FPGA (Field Programmable Gate Array) chip. The MIPS microprocessor is an existing architecture and is implemented in this research to provide the baseline processor platform for the future micro-threading architectural add-ons and modifications.

A part of the focus of this research is an investigation of the novel vector micro-threading architecture as an alternative approach to the current superscalar-based speculative microprocessor designs. Micro-threading is based on the not-so-novel multithreading technique which avoids speculation altogether and instead, starts running a different thread of instructions while waiting for the non-determinism to be resolved. This utilizes the chip resources more efficiently without waste of any processing power. This was covered in chapters 3 and 8.

As this research progressed, the baseline RISC processor platform, the MIPS R2000, was first reviewed, then synthesized from the RTL (Register Transfer Level) description using VHDL and then simulated and tested. This was covered in chapters 5, 6, and 7. This was conducted in order for future research to build upon and add the micro-threading architectural add-ons and modifications.

One outcome of this research is the publication of a total of five papers (refereed and non-refereed) in five different conference proceedings within New Zealand [2, 14, 36, 55] and abroad [76].

9.2 Thesis Conclusions

The following conclusions are drawn from this research:

- The baseline MIPS R2000 microprocessor has been synthesized onto the Virtex-II FPGA and simulated successfully. This 8-million-gate Virtex-II FPGA chip was much more than sufficient to accommodate this design.

- It is also concluded that the ease of designing with FPGAs compared to ASICs justifies the process of using FPGAs for design prototyping prior to final commercialization onto an ASIC chip.

- It is estimated that this 8-million-gate Virtex-II FPGA chip can accommodate a fully functional micro-threaded microprocessor, at the least. There even exists the possibility that a micro-threaded chip multiprocessor with 2-4 PUs can still be fitted onto this FPGA chip.

9.3 Suggested Future Work

This research can spawn future work in the following suggested areas:

- Re-design the register file to segregate the read signal separately from the write signal in order to prevent the accidental and unwanted overwrite of the operands stored in the registers.

- Implement the rest of the MIPS instructions, especially the multiplication and division.

- Implement the following advanced techniques for the existing baseline MIPS R2000 microprocessor synthesized in this thesis:

    - Pipelining with pipeline registers [1, ch.6].
    - Bypass buses and data forwarding [1, ch.6].
    - Branch prediction, out-of-order execution, and superscalar functionality [1, ch.6].
    - Proper memory hierarchy system including cache memory [1, ch.7].

- Develop and synthesize the micro-threading VHDL model with all the micro-threading add-on components and combine them with the existing baseline MIPS R2000 microprocessor synthesized in this thesis to synthesize the finalised micro-threading microprocessor and chip multi-processor.

REFERENCES

References

1. Akkary H. and Driscoll M., 1998, *A Dynamic Multithreading Processor,* 31st Annual ACM/IEEE International Symposium on Microarchitecture, Nov.30-Dec.2.

2. Al-Ali F.M. and Browne R.F, *An FPGA Implementation of a RISC Microprocessor*, Proceedings of the 11th Electronics New Zealand Conference (ENZCon'04), 15-16 Nov, 2004, p 106-111, ISBN 0-476-01106-X, Massey University, Palmerston North, New Zealand

3. Al-Ali F.M. and Browne R.F, *Behavioural VHDL Model of a Vector Micro-threading Chip-multiprocessor,* Proceedings of the 6th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC Asia 2002), Bangalore, 16-19 December 2002, Vol.2, pp 518-521, Tata McGraw-Hill Publishing Company Ltd., New Delhi, India, ISBN 0-07-049992-6

4. Al-Ali F.M. and Browne R.F, *VHDL Modelling of a RISC Microprocessor: Synthesis, Assembler, Loader, and Testing*, Proceedings of the 12th Electronics New Zealand Conference (ENZCon'05), 14-15 Nov, 2005, pp 63-68, ISBN 0-473-10634-5, Manukau Institute of Technology, Manukau City, Auckland, New Zealand.

5. Al-Ali F.M.and Jesshope C.R, 2000, *Survey of High-latency Tolerance in Contemporary Microprocessor Architectures*, Proc. 7th Annual New Zealand Engineering and Technology Postgraduate Conference, pp339-346, ISBN 0-473-07224-6, Massey University, Palmerston North, New Zealand, 23rd & 24th Nov

6. Al-Ali F.M. and Jesshope C.R, 2001, *Survey of High-latency Tolerance in Future Microprocessor Architectures*, Proc. New Zealand Computer Science research Students' Conference (NZCSRSC 2001), pp86-97, TR-COSC 02/01, University of Canterbury, Christchurch, New Zealand, 19th & 20th April.

7. Alverson G., Kahan S., Korry R., McCann C. and Smith J.B., 1995, *Scheduling on the Tera MTA,* Lecture Notes in Computer Science, 949, Springer-Verlag, Berlin, pp19-44.

8. Alverson R., Callahan D., Cummings D. Koblenz B., Porterfield A. and Smith B, 1990, *The Tera computer System,* Proceedings of the ACM international conference on Supercomputing, pp1-6.

9. Arvind, Gostelow K.P. and Plouffe W., 1978, *An Asynchronous Programming Language and Computing Machine,* Technical Report 114a, Dept. of Information and Computer Science, University of California at Irvine, CA.

10. Arvind and Nikhil R.S., 1987, *Executing a Program on the MIT Tagged-Token Dataflow Architecture,* Lecture Notes in Computer Science 259, Springer-Verlag, Berlin, pp1-29.

11. Ashenden P.J., *The Designer's Guide to VHDL*, 2nd Edition, Morgan Kaufmann Publishers, ISBN 1-55860-674-2, 2002.

12. Bolchevsky A, 1995, *The Fundamental Issues and Construction of a Data-parallel Dataflow Computer,* Technical Report CSRG95-01, Dept. of Engineering, University of Surrey.

13. Bolychevsky A., Jesshope C.R. and Muchnick V.B., *Dynamic Scheduling in RISC Architectures,* IEE Proc. Comput. Digit. Tech., vol. 143 (5), Sept 1996.

14. Burger D , Kaxiras S. and Goodman J.R., 1997, *Datascalar Architectures,* Proc. ISCA-24, Denver, CO, pp338-349.

15. Dennis J.B. and Misunas D.P., 1975, *A Preliminary Architecture for a Basic Dataflow Processor,* Proceedings of the 2nd Annual Symposium on Computer Architecture, Jan., Houston, TX, pp126-132.

16. Eggers S., Emer J., Levy H., Lo J., Stamm R. and Tullsen D, 1997, *Simultaneous Multithreading: A Platform for Next-generation Processors,* IEEE Micro, Sep/Oct, pp 12-18.

17. Flynn M.J., 1995, *Computer Architecture: Pipelined and Parallel Processor Design,* Jones and Bartlett Publishers, Sudbury, MA, ISBN 0-86720-204-1.

18. Franklin M., 1993, *The Multiscalar Architecture,* Computer Science Technical Report No. 1196, University of Wisconsin-Madison, WI.

19. Gaudiot J.L. and Bic L., 1991, *Advanced Topics in Dataflow Computing,* Prentice Hall, Englewood Cliffs, NJ.

20. Geppert L. and Perry T.S., 2000, *Transmeta's Magic Show,* IEEE Spectrum, May.

21. Gostelow K.P. and Arvind, 1982, *The U-interpreter,* Computer 15, Feb., pp42-49.

22. Grafe V.G.and Hoch J.E., 1990, *The Epsilon-2 Multiprocessor System,* Journal of Parallel and Distributed Computing 10, pp309-318

23. Gribbon K., *An FPGA Implementation of a Network Co-Processor*, 4th Year Engineering Project Report, Institute of Information Sciences and Technology, Massey University, Palmerston North, New Zealand, 2002.

24. Gwennap L., 1997, *DanSoft Develops VLIW Design,* Microprocessor Report, 11(2), pp18-22, Feb. 17.

25. Hennessy J.L. and Patterson D.A., 1996, *Computer Architecture: A Quantitative Approach,* 2ed, Morgan Kaufmann Publishers, San Francisco, CA, ISBN 1-55860-372-7.

26. Hockney R.W. and Jesshope C.R., 1988, *Parallel Computers 2: Architecture, Programming and Algorithms,* IOP Publishing Ltd., Bristol, ISBN 0-85274-811-6.

27. Hwang K., 1993, *Advanced Computer Architecture: Parallelism, Scalability, Programmability,* McGraw-Hill Inc., USA, ISBN 0-07031622-8.

28. Iannucci R.A., Gao G.R., Halstead Jr R.H.. and Smith B., 1994, *Multithreaded Computer Architecture: A Summary of the State of the Art,* Kluwer Academic Publishers, USA, ISBN0-7923-9477-1.

29. Intel Corporation, 1999, *IA-64 Application Developer's Architecture Guide,* May, Intel Corp.

30. Intel, *Hyper-Threading Technology (website),* http://developer.intel.com/technology/hyperthread/

31. Intel Itanium Processor Family Home Page, 20th Oct 2000, developer.intel.com/design/ia-64/

32. Jesshope C.R., *Implementing an Efficient Vector Instruction Set in a Chip Multi-processor Using Micro threaded Pipelines,* Proc. Australian Computer Systems Architecture, Gold Coast, Australia, February 2001.

33. Jesshope C.R. and Luo R(B)., *A Microthreaded Chip Multiprocessor with a Vector Instruction Set*, Chris Jesshope's Homepage (downloaded 4th June, 2002): www2.dcs.hull.ac.uk/people/cssrj/index.html

34. Jesshope C.R. and Luo R(B)., *Micro-threading: A New Approach to Future RISC,* Proc. 2000 ACAC, Canberra, IEEE Computer Society Press, ISBN 0-7695-0512-0, pp34-41, Jan 2000.

35. Lipasti M.H.and Shen J.P., 1997, *Superspeculative Microarchitecture for Beyond AD 2000,* Computer, 30, pp59-66, September.

36. Lipasti M.H. and Shen J.P., 1997, *The Performance Potential of Value and Dependence Prediction,* Lecture Notes in Computer Science, 1300, pp1043-1052, Springer-Verlag, Berlin.

37. Lipasti M.H., Wilkerson C.B. and Shen J.P., 1996, *Value Locality and Load ValuePrediction,* Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, pp134-147, October

38. Lo J.L., 1998, *Exploiting thread-level parallelism on simultaneous multithreaded processors,* Ph.D. Thesis, Dept. of Computer Science and Engineering, University of Washington.

39. Luo R(B). and Jesshope C.R., *Performance Evaluation on a Microthreading Pipeline,* Proc. 2002 ACAC, Canberra, Jan., IEEE Computer Society Press, 2002.

40. Mano M. and Kime G., *Logic and Computer Design Fundamentals*, 2nd Edition, Prentice Hall, 2001.

41. Nikhil R.S., Papadopoulos G.M. and Arvind, 1992, **T: A Multithreaded Massively Parallel Architecture,* Proceedings of the 19th Annual Symposium on Computer Architecture, May, Gold Coast, pp156-167.

42. Patt Y.N., Patel S.J., Evers M., Friendly D.H. and Stark J., 1997, *One Billion Transistors, One Uniprocessor, One Chip,* Computer, 30, pp51-57, September.

43. Papadopoulos G.M., 1988, *Implementation of a General-Purpose Dataflow Multiprocessor,* Technical Report TR-432, Laboratory For Computer Science, MIT, Cambridge, MA.

44. Papadopoulos G.M., 1991, *Implementation of a General-Purpose Dataflow Multiprocessor,* Research Monographs in Parallel and Distributed Computing, The MIT Press, Cambridge, Massachusetts, Pitman Publishers, London, ISBN 0-273-08835-1.

45. Papadopoulos G.M. and Culler D.E., 1990, *Monsoon: An Explicit Token-Store Architecture,* Proceedings of the 17th Annual Symposium on Computer Architecture, May, Seattle, WA, pp82-91.

46. Papadopoulos G.M. and Traub K.R., 1991, *Multithreading: A Revisionist View of Dataflow Architectures,* Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Ontario, pp342-351, May.

47. Patterson D.A. and Hennessey J.L., *Computer Organisation and Design: The Hardware/Software Interface,* 2nd Edition, Morgan Kaufmann Publishers, San Francisco, 1998.

48. Patterson D.A. and Hennessey J.L., *Computer Organisation and Design: The Hardware/Software Interface,* 3rd Edition, Morgan Kaufmann Publishers (An Imprint of Elsevier), San Francisco, 2005

49. Reese R., *ABM Pipeline VHDL Simulator,* Robert (Bob) Reese's Homepage (downloaded October 2000): www.erc.msstate.edu/~reese/

50. Rotenberg E., Jacobson Q., Sazeides Y. and Smith J.E., 1997, *Trace Processors,* Proc. of the MICRO-30, Research Triangle Park, NC, pp138-148.

51. Schlanker M.S. and Rau B.R., 2000, *EPIC: Explicitly Parallel Instruction Computing,* IEEE Computer, Feb., pp37-45.

52. Silc J., Robic B. and Ungerer T., 1998, *Asynchrony in Parallel Computing: From Dataflow to Multithreading,* Parallel and Distributed Computing Practices 1, pp57-83.

53. Silc J., Robic B. and Ungerer T., *Processor Architecture: From Dataflow to Superscalar and Beyond.* Berlin Heidelberg, Germany: Springer-Verlag, 1999.

54. Silc J., Ungerer T. and Robic B, 2000, *A Survey of New Research Directions in Microprocessors,* Microprocessors and Microsystems, 24, pp175-190.

55. Smith B.J., 1981, *Architecture and applications of the HEP multiprocessor computer system,* Proc. SPIE, 298, pp241-248

56. Smith J.E. and Vajapeyam S., 1997, *Trace Processors: Moving to Fourth-Generation Microarchitectures,* Computer, 30, pp 68-74.

57. Sohi G.S., 1997, *Multiscalar: Another Fourth-Generation Processor,* Computer, 30, pp72

58. Sohi G.S., Breach S.E. and Vijaykumar T.N., 1995, *Multiscalar Processors,* Proc. ISCA-22, santa Margherita Ligure, Italy, pp414-425.

59. Srini V.P., 1986, *An Architectural Comparison of Dataflow Systems,* Computer 19, Mar., pp68-88.

60. Sun Microsystems, *MAJC Architecture Tutorial (website),* http://www.sun.com/microelectronics/majc/documentation/docs/majctutorial.pdf

61. Texas Instruments, 1998, *TMS320C6201 Digital Signal Processor Advance Information,* March, Texas Instruments, Houston, TX.

62. Tomasulo R.M., 1967, *An Efficient Algorithm for Exploiting Multiple Arithmetic Units,* IBM Journal of Research and Development, (January), Vol. 11 (1), pp25-33.

63. Transmeta Home Page, 29th Oct. 2000, www.transmeta.com/crusoe

64. Treleaven P.C., Brownbridge D.R. and Hopkins R.P., 1982, *Data-Driven and Demand-Driven Computer Architectures,* ACM Computing Surveys 14, pp93-143.

65. Tremblay M., 1999, *MAJC-5200: A VLIW Convergent MPSOC,* Microprocessor Forum '99.

66. Tsai J.Y. and Yew P.C., 1996, *The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation*, Proc. 1996 Conference on Parallel Architectures and Compilation Techniques (PACT'96), pp35-46, Boston, MA, October

67. Unger A., Ungerer T. and Zehendner E., Website for *Compiler Techniques for Multithreaded Processors,* http://goethe.ira.uka.de/people/ungerer/mf-compiler/sss.html

68. Vajapeyam S. and Mitra T., 1997, *Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences,* Proc. of the ISCA 24, Denver, CO, pp1-12.

69. Xilinx Inc. (website), *Virtex-II Platform FPGAs: Introduction and Overview*, www.xilinx.com, Downloaded Oct 2003.

70. Xilinx Inc. (website), *Virtex-II Platform FPGA User Guide*, www.xilinx.com, Downloaded Oct 2003

71. Xilinx Inc. (website), *Xilinx/Google Search Engine*, xgoogle.xilinx.com, Accessed June/July 2006.

72. Xilinx Inc. (website), *Xilinx ISE 5 Software Manuals*, www.xilinx.com, Downloaded Oct 2003.

73. Xilinx Inc. (website), *Xilinx PLD Quick Start Handbook*, 2nd Edition, www.xilinx.com, Jan 2002, Downloaded 2003.

74. Xilinx Inc. (website), *Xilinx Support Website*, support.xilinx.com, Accessed June/July 2006.

75. Yalamanchili S., *Introductory VHDL: From Simulation to Synthesis*, Prentice Hall, 2001.

76. Zehendner E. and Ungerer T., 1987, *The ASTOR Architecture,* Proceedings of the 7th International Conference on Distributed Computing Systems, Sept., Berlin, pp424-430

GLOSSARY

A

ABM	ABstract Machine
ADDI	ADD Immediate
ALU	Arithmetic and Logical Unit
ASIC	Application Specific Integrated Circuit

B

BEL	Basic Element
BEQ	Branch on Equal
BNE	Branch on Not Equal
BRAM	Block Random Access Memory
BUFG	Global BUFfer
BUFT	Tri-state BUFfer

C

CAD	Computer Aided Design
CE	Clock Enable
CISC	Complex Instruction Set Computing
CLA	Carry Look Ahead
CLB	Configurable Logic Block
CMP	Chip Multi-Processor
CPI	Cycles Per Instruction
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CQ	Continuation Queue

D

DCM	Digital Clock Manager
DF	Data Fetch
DFG	Data Flow Graph
DMT	Dynamic Multi-Threading

E

EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read Only Memory

EPIC	Explicitly Parallel Instruction Computing
ETS	Explicit Token Store
EX	EXecute

F

FDC	Data Flip flop with Clear
FMAP	Function MAPping
FP	Floating Point
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit

G

GCLK	Global CLocK
GCQ	Global Continuation Queue
GRF	Global Register File

H

HEP	Heterogeneous Element Processor
HPL-PD	Hewlett Packard Laboratories – Play Doh
HT	Horizontal Transfer
HT	Hyper-Threading

I

IA	Intel Architecture
IBUF	Input BUFfer
ID	Instruction Decode
IDRR	Instruction Decode and Register Read
IEEE	Institute of Electrical and Electronic Engineers
IF	Instruction Fetch
ILP	Instruction-Level Parallelism
IOB	Input/Output Block
IOC	Input/Output Cache unit
IOP	Input/Output Processor
IPC	Instructions Per Cycle

ISA	Instruction Set Architecture
ISE	Integrated Synthesis Environment

J

J	Jump
JR	Jump Register

L

LCQ	Local Continuation Queue
LRF	Local Register File
L/S	Load/Store
LUT	Look Up Table
LW	Load Word

M

MAJC	Microprocessor Architecture for Java Computing
MEMA	MEMory Access
MIMD	Multiple Instruction stream Multiple Data stream
MTA	Multi-Threaded Architecture
MTI	Model Technology Inc.
MU	Memory Unit
MXE	Modelsim - Xilinx Edition

N

NOP	No OPeration
NRE	Non-Recurring Engineering cost

O

OBUF	Output BUFfer

P

PC	Program Counter
PE	Processing Element
PLD	Programmable Logic Device
POE	Plan Of Execution

PSW	Process Status Word
PU	Processing Unit

R

RAU	Register Allocation Unit
RF	Register File
RISC	Reduced Instruction Set Computing
RR	Register Read
RTL	Register Transfer Level

S

S3	Simultaneous Speculation Scheduling
SAL	Single Assignment Language
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SMT	Simultaneous Multi-Threading
SRAM	Static Random Access Memory
STC	Space Time Computing
SW	Store Word

T

T2	Terminator 2
TBUF	Tri-state BUFfer
TP	Thread Pointer
TS	Target Store
TSAG	Target Store Address Generation

U

u-T	Micro-Threading

V

VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VMT	Vertical Multi-Threading
VT	Vertical Transfer

W

WB	Write Back

X

XST	Xilinx Synthesis Tools