THE UNIVERSITY OF HULL

CHIP MULTI-PROCESSORS USING A MICRO-THREADED MODEL

Being a Thesis submitted for the Degree of PhD
in the University of Hull

By

Eng. Nabil M. Hasasneh, BSc. (Hons), MSc. (Hons)

October 2006

Abstract

Most microprocessor chips today use an out-of-order (OOO) instruction execution mechanism. This mechanism allows superscalar processors to extract reasonably high levels of instruction level parallelism *(ILP)*. The most significant problem with this approach is a large instruction window and the logic to support instruction issue from it. This includes generating wake-up signals to waiting instructions and a selection mechanism for issuing them. Wide-issue width also requires a large multi-ported register file, so that each instruction can read and write its operands simultaneously. Neither structure scales well with issue width leading to poor performance relative to the gates used. Furthermore, to obtain this ILP, the execution of instructions must proceed speculatively.

An alternative, which avoids this complexity in instruction issue and eliminates speculative execution, is the *microthreaded model.* This model fragments sequential code at compile time and executes the fragments OOO while maintaining in-order execution within the fragments. The fragments of code are called microthreads and they capture ILP and loop concurrency. Fragments can be interleaved on a single processor to give tolerance to latency in operands or distributed to many processors to achieve speedup. The major advantage of this model is that it provides sufficient information to implement a penalty free distributed register file organisation.

However, the scalability of the microthreaded register file in terms of the number of required logical read and write ports is not clear yet. In this thesis, we looked at the distribution and frequency of access to the asynchronous (non-pipeline) ports in the synchronising memory and provide a detail analysis and evaluation of this issue.

It concluded, using an analysis of a range of different code kernel, that a distributed shared synchronising memory could be implemented with 5-ports per processor, where three ports provided single instruction issue per cycle and the other two asynchronous ports were able to manage all other demands on the local register file.

Also, in the microthreaded CMP a broadcast bus is used for thread creation and to replicate the compiler-defined global state to each processor's local register file. This is done instead of accessing a centralised register file for global variables. The key problem is that, accessing this bus by multiple processors simultaneously caused contention and unfair communication between processors. Therefore, to avoid processor contention and to take the advantages of asynchronous communication, this thesis presents a scalable and partitionable asynchronous bus arbiter for use with chip multiprocessors *(CMP)* and its corresponding pre-layout simulation results using *VHDL*. It is shown in this thesis that this arbiter can be extended easily to support large numbers of processors and can be used for chip multiprocessor arbitration purposes. Furthermore, the microthreaded model requires dynamic register allocation and a hardware scheduler, which can support hundreds of microthreads per processor and their associated microcontexts. The scheduler must support thread creation, context switching and thread rescheduling on every machine cycle to fully support this model, which is a significant challenge. In this thesis, scalable implementations and evaluation of these support structures are presented and the feasibility of large-scale CMPs is investigated by giving detailed area estimate of these structures using 0.07-micron technology.

Acknowledgements

First of all, I would like to thank my supervisors, Professor Chris Jesshope and Mr. Ian Bell, for their many suggestions and constant support during this research. This dissertation would not exist without their guidance, support and continual encouragement. Many of the results in this thesis have come out of long conversations and meetings with Prof. Chris Jesshope and Mr. Ian Bell. I thank you both very much. I would also like to thank Dr. Anthony Wilkinson for the comments and feedback he has given me throughout my PhD.

I am very grateful to the examiners Dr. Colin Egan and Eur Ing Brian Tompsett for their patience in reading my thesis and for their comments that made this thesis more accurate, and more complete.

Support must also come from outside work, and in this respect, I would like to thank my parents, my brothers (especially Alaa) and my sisters for their concern and support throughout the years of studies as well as for instilling the value of education in me. Also, I want to thank my wife and two lovely sons, Qossai and Seleen, for their patience, understanding during three years of my study.

I would also like to thank both the Hebron University and Ministry of Education and Higher Education for granting me a scholarship for my PhD study at the University of Hull. Also, I would like to thank Dr. Nabil Al-Jabari, Chairman of the Board of Trustees of Hebron University for his continual encouragement. Many thanks to the Engineering Department and the Department of Computer Science for their help and support.

Last, but not least, special thanks to God, who has inspired me to accomplish this research successfully.

NABIL M. HASASNEH

THE UNIVERSITY OF HULL
October, 2006

To my Parents, my Brothers, my Sisters, my Wife and my lovely Sons.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There is an ever increasing demand for high performance microprocessors in a variety of application areas including PCs, servers and mobile devices. Most current processor architectures are reaching fundamental limits in terms of scalability, speed, die area and power consumption. It is generally agreed that the way forward for high performance processors is to increases the number of cores, and a variety of multiprocessor architectures have been proposed. This thesis concerns chip multiprocessor *(CMP)* based on the microthreaded model. We investigate the feasibility of implementing these processors, with particular emphasis on scalability, by developing and analysing high level architecture models.

Microthreading is a hardware threading technique, where each thread is a very small fragment of code. Using a considerable number of parallel threads allows the multiprocessor system to exploit more parallelism, which in turn improves the overall system performance. Before introducing the microthreaded model and the CMP in detail, it is useful to explain how microthreads differ from operating system threads.

In operating system such as UNIX and Windows thread are viewed as simultaneously running multiple tasks and are popular method to improve application software

through parallelism. The exact implementation of threads differ from one operating system to another, but in general multiple programs can be run at once, such as a word processor alongside an audio playback program and each program can be split into multiple threads. Each thread has an execution state (running, ready,....etc.) and has an execution stack and some per-thread static storage for local variables.

The notion of a thread in the context of multithreaded processors differs from the notion of software threads in multithreaded operating systems [1]. In the case of multithreaded processors a thread is always seen as a hardware-supported thread. User-defined or compiler generated threads in microthreading do not require interaction with the operating system and each thread is represented by a program counter, registers and a small control block. Hardware multithreading is a general technique for hide long memory latencies by automatically switching to a new thread when one thread blocked. Thread switching is performed by the processor using a hardware-based thread-switching policy. More details about the multithreading processors are available in [1]. In this work, unless otherwise stated, the term thread is used to describe very small code fragments with minimal context.

As previously indicated many researchers are interested in the idea of achieving major increases in the computational power of computers by the use of CMP. Examples of CMP are the Compaq Piranha [2], Stanford Hydra [3] and Hammond et. al. [4]. Several architectures have been proposed and some manufacturers have produced commercial designs, such as the IBM Power PC [5] Sun Niagara [6] and Intel's Montecito [7].

Ideally, the performance of such systems should be directly proportional to the number of processors used, i.e. they should be scalable. CMPs scale well, with the

limit to scalability defined by Moore's law. We calculate that current technology could support hundreds of in-order processors and achieve significant speedup over current architectures that use implicit concurrency and achieve minimal speedup through concurrent instruction issue [8]. One of the major barriers to the use of CMPs is the problem of programming them without using explicit concurrency in the user code. Ideally they should be programmable using legacy sequential code.

In theory, there is no limit to the number of processors that can be used in a CMP provided that the concurrency derived from the sequential code scales with the problem size. The problem is how to split the code into a number of independent threads, schedule these on many processors and to do this with a low and scalable overhead in terms of the control logic and processor efficiency. In fact on general-purpose code it will be impossible to eliminate all dependencies between threads and hence synchronisation is also required. The goal of this work therefore is to define a feasible architecture for a scalable CMP that is easy to program, that maximises throughput for a given technology and that minimises the communication and synchronisation overheads between different threads.

Today Intel's Itanium-2 *(Madison)* microprocessor features over 410 million transistor in a 0.13μm semiconductor process technology operating at a speed of 1.5GHz. This is a dual-processor version of the previous Itanium processor *(Mckinley)*, which has an issue width of six. Moore's law would indicate that the billion-transistor chip will become feasible in 65nm technology within the next three or four years [9]. Intel's Montecito is the first Itanium processor to feature duplicate, dual-thread cores on a single chip with 1.72 Billion transistors. The questions we must ask are where do we go from here and what is the best way to utilise this wealth of transistors, while

maximising performance and minimising power dissipation?

Another problem area in future technology is the scaling of wire delays compared with gate delays. As transistor dimensions scale down, the number of gates which are reachable within the scaled clock is at best constant, which means that distributed rather than monolithic architectures need to be exploited [10]. Superscalar processors today issue up to eight instructions per clock cycle but instruction issue is not scalable [11] and a linear increase in parallelism requires at least a quadratic increase in area [12]. The logic required occupies about 30% of the total chip area in a 6-way superscalar processor [13].

In addition, more and more area is being used for on-chip memory. Typically the second level on-chip cache occupies 25-30% of the die area on a modern microprocessors, and between 50-75% on the recently announced Itanium-2. Moreover, a significant delay and power consumption are seen in high-issue-width processors due to tag matching, wake-up signals to waiting instructions, and selection mechanisms for issuing instructions. These delays increase quadratically for most building blocks with the instruction window size [14]. Finally, even with the implementation of a large instruction window, it is difficult for processors to find sufficient fine-grain parallelism, which has made most chip manufacturers like Compaq, Intel and Sun look at simultaneous multithreading *(SMT)* [15] to expose more ILP through a combination of coarse- and fine-grain parallelism.

Multithreading can expose higher levels of concurrency and can also hide latency by switching to a new thread when one thread stalls. SMT appears to be the most popular form of multithreading. The main draw back to SMT is that it complicates the instruction issue stage, which is central for the multiple threads [1]. Scalability

in instruction issue is no easier to achieve because of this and the other scalability problems remain unchanged. Thus SMT suffers from the same implementation problems [16] as superscalar processors.

An alternative approach to multithreading that eliminates speculation and does provide scalable instruction issue is the *microthreaded model*. The threads in this model are small code fragments with an associated program counter. Little other state information is required to manage them. The model is able to expose and support much higher levels of concurrency using explicit but dynamic controls. In pipelines that execute this model, instructions are issued in-order from any of the threads allocated to it but the schedule of instructions executed is non-deterministic, being determined by data availability. Threads can be deterministically distributed to multiple pipelines based on a simple scheduling algorithm. The allocation of these threads is dynamic, being determined by resource availability, as the concurrency exposed is parametric and not limited by the hardware resources. The instruction issue schedule is also dynamic and requires linear hardware complexity to support it. Instructions can be issued from any microthread already allocated and active. If such an approach could also give linear performance increase with number of pipelines used, then it can provide a solution to both CMP and ILP scalability [17]. However, in this model and its CMP architecture, there are still problems which need to be resolved. In this thesis we highlight these problems and provide an implementation solution with required analysis and evaluation.

Finally, it is necessary to define the terms used in this thesis heading:

Microthread : (not hyphenated to distinguish it from other uses of the same combination of terms), refers specifically to code fragments managed by the model described

in this thesis and the previous, related papers.

Microcontext : refers to the private state associated with a microthread. This includes a microthread's program counter and an offset into the register file, which locates its private register variables.

Microgrid : refers to a CMP where all processors have a microthreaded scheduler and a synchronising, distributed shared register file.

1.1 Motivation

Chip multi-processors *(CMPs)* are a very promising solution for future high-performance computing and we anticipate that many new microprocessor designs will be based on such an approach. As described previously, several projects have already investigated CMPs, and manufacturers are beginning to produce commercial designs.

The appeal of CMP architectures comes from factors that limit the scalability of multiple instruction issue in conventional processors [18], such as the superscalar paradigm, which continue to use more silicon and power for very little improvement in ILP. Evidence of this is provided by Intel's cancellation of its 4GHz Pentium4 [19], which has effectively reached a limit in both performance and scalability. Scaling up concurrency in these processors gives very large circuit structures and this is exacerbated by lengthy global communication arising from the increasing problems of wire delay in technology scaling. These require excessive chip area and increased power consumption respectively. For example, the logic required for OOO issue does not scale with issue width [20] and will eventually dominate the chip area and power consumption.

The Semiconductor Industry Association (SIA) roadmap indicates that by 2018

the number of transistors on a single chip will reach 4 billion to 25 billion depending on the circuit type [21]. How to gain performance from this level of integration within acceptable power budgets is a major problem. Performance can not be achieved by simply increasing the speed of conventional processors or by squandering a large number of transistors on unscalable support structures, as used in OOO issue. Instead, we have to embrace explicit parallelism, but systematic solutions to parallel programming and parallel architectures have yet to emerge, even with small-scale concurrency. In the near future, we will be able to integrate thousands of arithmetic units on a single chip [22]. Note that a 32-bit integer ALU occupies less than $0.05mm^2$ in an $180nm$ CMOS process technology and typical chip sizes are between 100 to $400mm^2$. However, before such chips can be utilised, we need programming paradigms for generating this level of concurrency and support structures for scheduling and managing this concurrency, which are fully scalable.

Today's small-scale CMPs are based on the same complex processor designs that preceded them and use high-level or software-based concurrency (e.g. threads). These threads may be scheduled in software or hardware and even used to extend the pool of instructions to support OOO issue. The latter, in particular, suffers from major problems, which limits performance and prevents overall-system scalability. These problems are summarised in [23] and systems based on this approach scale badly and are unable to exploit Moore's law to the full.

In general, there are only a few requirements for the design of efficient and powerful general-purpose CMPs, these are: scalability of performance, area and power with issue width, and programmability from legacy sequential code. Issue width is defined here as the number of instructions issued on chip simultaneously, whether in

a single processor or in multiple processors and no distinction is made here. To meet these requirements a number of problems must be solved, including the extraction of ILP from legacy code, managing locality, minimising global communication, latency tolerance, power-efficient instruction execution strategies (i.e. avoiding speculation), effective power management, workload balancing, and finally, the decoupling of remote and local activity to allow for an asynchronous composition of synchronous processors. Most CMPs address only some of these issues as they attempt to reuse elements of existing processor designs, ignoring the fact that these are suitable only for chips with relatively few cores.

In this thesis a CMP is evaluated, that is based on microthreading, which addresses either directly or indirectly, all of the above issues and, theoretically, provides the ability to scale systems to very large number of processors [24]. It will be shown in this thesis that such CMPs use hardware scheduling and synchronisation and have structures to support this that are distributed, fully scalable and have locality in communication wherever possible. This is achieved with distributed schedulers that jointly manage large parametric families of threads and a distributed register file that provides synchronisation and sharing of data between them. These structures provide support for a shared-register, instruction-level model of concurrency in which synchronisation occurs between instructions and in the registers. The model requires instructions in the ISA to specify and manage this concurrency, but this is achieved by adding just a few additional instructions to a conventional ISA. The result, is that concurrency can be captured in an abstract and parameterised way in the binary code, rather than by calls to an operating system. This is a large advantage in exploiting efficient execution of concurrency in CMPs. This concurrency provides both speedup

and latency tolerance in a single processor.

1.2 Research Objectives

Microthreading is a model of concurrency limited to a single context, which shares the registers allocated to that context. The major advantage of this model is that it provides sufficient information to implement a penalty free distributed register file organisation. Such a proposal is given in [8] where each processor in a CMP has its own register file. However, it is not clear what are the requirements on the register file in term of the number of required read/write ports. This thesis therefore derived and analysed the detailed requirements of the microthreaded distributed register-file ports, in terms of the frequency of accesses to each logical port.

Another advantage of the microthreaded model is allocating and de-allocating registers dynamically [25] prior to thread scheduling to supports concurrent threads when executing code for multiple iterations of a loop concurrently. However, allocating registers dynamically requires an efficient hardware scheme to model and allocate register usage and this scheme is examined thoroughly in this thesis.

Another problem in the microthreaded CMP is that the distributed implementation of a microthreaded CMP includes two forms of asynchronous communication. The first is the broadcast bus, used for creating threads and distributing invariants. The second is the shared-register ring network which is used to perform communication between the register files in the producer and consumer threads. Therefore, to avoid contention during bus access, and to provide fairness in communication between processors, we need some form of arbiter. Also, it is not clear how the bus interface between processors can be implemented. In this thesis we discuss these issues and we

introduce a novel asynchronous arbiter optimised for this application.

The major advantage of the microgrid CMP is its scalability in terms of performance, power and area with instruction-issue width. The first two issues are demonstrated in [24, 26]. The third issue is demonstrated in this thesis, which shows scalable implementations in instruction-issue width of the chip support structures and the feasibility of large-scale CMPs.

1.3 Thesis Contributions

The high-level contribution of this thesis is to investigate the architectural implementation of the microgrid CMPs based on a distributed register file organisation and to contribute the scalable implementation of the chip support structures. In particular, the contributions of this thesis are detailed in the following points:

- We provide a comprehensive summary and survey of current and alternate microarchitecture approaches and their challenges.
- A detailed evaluation and analysis of the requirements of the microthreaded distributed register file. The results shows that the register file can be distributed between the processors and that each register file requires only 5 fixed ports, making it compact and scalable. This work has been published in the British Computer Journal [26].
- A detailed design and implementation of a scalable and partitionable asynchronous arbiter together with required bus interface for the microgrid chip multiprocessor. The arbiter was designed to the gate level, and pre-layout simulation results using VHDL are presented in this thesis. This contribution also

has been published in [27, 28].

- A detailed design and implementation of a hardware scheme for dynamically allocating and de-allocating registers for the microthreaded chip multiprocessor. Detailed evaluation and simulation results of this scheme are presented in this thesis. Also, to demonstrate the feasibility and scalability of the multiprocessor in term of silicon implementation, we perform a detailed area estimate of a microgrid core and its support structures using 0.07 micron technology. Scalable implementations of such support structures are given in this thesis and the feasibility of large-scale CMPs is investigated. We show also that the support structures are of a manageable size and moreover are scalable in issue width. This work also has been published in the Parallel Programming Journal [29].

- A detailed design and implementation of a microthreaded scheduler and the first two stages of the microthreaded pipeline. This contribution also has been submitted to [30].

1.4 Thesis Organisation

The remaining chapters in this thesis are organised as follows.

In **Chapter 2**, we present background information about the existing micro-architecture approaches, alternative approaches, and recent chip multiprocessor architectures. Then we outline some challenges facing these approaches. The chapter also presents an overview of distributed memory multiprocessor architectures and their design parameters. Finally, techniques for system evaluation that are employed in this thesis are presented.

In **Chapter 3**, we consider a microthreaded microprocessor concurrency model highlighting features that support the implementation of a scalable and powerful CMP and show the problems that need to be resolved. In particular, we first present a microthreaded ISA, and a microthreaded in-order execution pipeline. The chapter then presents the concurrency controls used in this model in full detail. Scalability of the instruction issue and thread state in microthreaded model are also discused in this chapter. The chapter also shows how the microthreaded model provides register file partitioning and support a mechanism for dynamically allocating registers. Finally, the chapter explains how the model support prefetching mechanism that avoids any instruction cache misses.

In **Chapter 4**, we first examine modern register files. The chapter then, describes the method of sharing registers in microthreaded model. An analysis and evaluation of the requirements of the microthreaded register file ports is also given. A comparison between centralised and distributed allocation organisation is also presented. In addition, we show an alternative allocation scheme that we have already discussed during our research. Finally, we present a hardware scheme for dynamically allocating and de-allocating registers to families of microthreads with its implementation and simulation results using VHDL.

In **Chapter 5**, we describes the microgrid chip multiprocessor architecture model, its features and also highlight problems that need to be resolve. In particular, we first introduce the top-level architecture for the microgrid CMP, and then describe the microgrid CMP communication buses. The chapter also discusses the Globally-Asynchronous Locally-Synchronous *(GALS)* design approach and its features. A method of thread distribution and scheduling in microgrid CMP is also presented.

Finally, we discuss microgrid scalability.

In **Chapter 6**, we present a scalable and partitionable asynchronous arbiter for microgrid chip multiprocessors. The chapter first introduces an asynchronous design methodology and its communication protocols. Techniques for deriving throughput, latency and wavelength for the ring self-time scheme are described. The chapter also provides a full detail design, analysis and implementation for the arbiter including a pre-layout simulation results using *VHDL.*

In **Chapter 7**, we provide an overview of the chip architecture, and gives an area estimate for the microthreaded support structures. We also, provide an estimated area for the microthreaded core and show the feasibility of large CMP using an emerging technology. Finally, the chapter presents full simulation results for the top-level model of the continuation queue and the scheduling system in VHDL in order to verify their correct behaviour.

In **Chapter 8**, we present our conclusions and suggestions for future research. A number of directions for future development of microgrid chip multiprocessors are given and areas of research are then outlined. Specification details are presented in the appendices.

1.5 Publications

The following papers based on the work presented in this thesis, have been published:

1.5.1 Journal Papers

- Bell, I.M., Hasasneh, N.M. and Jesshope C.R. (2006) Microgrids and Microcontexts: Support Structures for Microthread Scheduling and Synchronisation, International Journal of Parallel Programming, Volume. 34, No. 4, August 2006 ,pp. 1-39.

- Hasasneh N.M., Bell I.M., and Jesshope C.R. (2006) Asynchronous arbiter for microthreaded Chip multiprocessors, to be published, Journal of Systems Architecture (JSA).

- Bousias, K., Hasasneh N.M. and Jesshope C.R. (2005) Instruction-level parallelism through Microthreading - a scalable Approach to chip multiprocessors, BCS, Comput. J. Vol. 49(2), (2006), pp. 211-233.

1.5.2 Conference and Workshop Papers

- Hasasneh, N.M. Bell, I.M.,and Jesshope C.R. (2005) High Level Modelling and Design For a Microthreaded Scheduler to Support Microgrids, Submitted to 2007 ACS/IEEE International Conference on Computer Systems and Applications, AICCSA 2007 May 13-16, 2007, Amman, Jordan, 2007.

- Hasasneh N.M., Bell, I.M. and Jesshope C.R. (2006) Scalable and Partitionable Asynchronous Arbiter for Microthreaded Chip Multiprocessors, Proc. Architecture of Computing Systems - ARCS 2006, Vol. 3894, ISBN: 3-540-32765-7, Germany, March 13-16, Lecture Notes in Computer Science (LNCS), Volume 3894, ARCS 2006 (Frankfurt/Main, Germany), pp. 252-267.

- Hasasneh N.M., Bell I.M., and Jesshope C.R. (2006) Modular Asynchronous Arbiter for Microthreaded Chip Multiprocessors, The Institution of Engineering and Technology Postgraduate Workshop on Embedded Systems 11 October 2006, NEC, Birmingham, UK Embedded Systems at ESS 2006.

- Hasasneh, N.M., Bell, I.M. and Jesshope, C.R. (2005), Asynchronous Arbiter for Microthreaded Chip Multiprocessors, UK Design Forum (UKDF), Manchester University, April 13-15, 2005.

- Jesshope, C.R., Bell, I.M., Hasasneh, N.M. (2004), Chip Multiprocessors Using a Microthreaded Model, Proceeding of the 1st Computer Science Graduate Research Conference, The University of Hull, July 2004.

Chapter 2

Background and Related Work

2.1 Current Approaches

It was shown in the previous chapter that approaches such as OOO execution, VLIW, and multithreading suffer from hardware and software problems. In this section, we explain these approaches and their challenges in more detail.

2.1.1 Out-of-Order (OOO) Execution

To achieve a higher performance, modern microprocessors use an OOO execution mechanism to keep multiple execution units as busy as possible. This is achieved by allowing instructions to be issued and completed out of the original program sequence as a means of exposing concurrency in a sequential instruction stream. More than one instruction can be issued in each cycle, but only independent instructions can be executed in parallel, other instructions must be kept waiting or, under some circumstances, can proceed speculatively.

Speculation refers to executing an instruction before it is known whether the results of the instruction will be used or not, this means that a guess is made as to

the outcome of a control or data hazard as a means to continue executing instructions, rather than stalling the pipeline. Register renaming is also used to eliminate the artificial data-dependencies introduced by issuing instructions OOO. This also enables the extension of the architectural register set of the original ISA, which is necessary to support concurrency in instruction execution. Any concurrent execution of a sequential program will require some similar mechanism to extend the synchronisation memory available to instructions. Speculative execution and OOO issue are used in superscalar processors to expose concurrency from sequential binary code. A reorder buffer or future file of check points and repairs is also required to re-order the completion of instructions before committing their results to the registers specified in the ISA in order to achieve a sequentially consistent state on exceptions.

Control speculation predicts branch targets based on the prior history for the same instruction. Execution continues along this path as if the prediction was correct, so that when the actual target is resolved, a comparison with the predicted target will either match giving a correctly predicted branch, or not, in which case there was a misspredict ion. A misspredict ion can require many pipeline cycles to clean up and, in a wide-issue pipeline, this can lead to hundreds of instruction slots being unused, or to be more precise, if we focus on power, to be used unnecessarily. It can therefore be described as a wasteful of chip resources and moreover has unpredictable performance characteristics [25]. We will show that it is possible to obtain high performance without speculation and, moreover, to save power in doing so.

As already noted in the previous chapter, as the issue width increases in an OOO, superscalar architecture, the size of the instruction window and associated logic increase quadratically, which results in a large percentage of the chip being devoted

to instruction issue. The OOO execution mechanism therefore prevents concurrency from scaling with technology and will ultimately restrict the performance over time. The only reason for using this approach is that it provides an implicit mechanism to achieve concurrent execution from sequential binary code.

2.1.2 Very-long Instruction Word (VLIW)

An alternative and explicit approach to concurrency in instruction issue is VLIW, where multiple functional units are used concurrently as specified by a single instruction word. This usually contains a fixed number of operations that are fetched, decoded, issued and executed concurrently. To avoid control or data hazards, VLIW compilers must hoist later independent instructions into the VLIW or if this is not possible, must explicitly add *no-op* instructions instead of relying on hardware to stall the instruction issue until the operands are ready. This can cause two problems, firstly, a stall in one instruction will stall the entire width of the instruction, secondly, adding no-op instructions, increases the program size. In terms of performance, if the program size is large compared to the I-cache or Translation Lookaside Buffer (TLB) size, it may result in higher miss rates, which in turn degrades the performance of the processor [31].

It is not possible to identify all possible sources of pipeline stalls and their duration at compile time. For example, suppose a memory access causes a cache miss, this leads to a longer than expected stall. Therefore, memory reference instructions (loads and stores) have a non-deterministic delay within the memory subsystem. The number of no-op instructions required is not known and most VLIW compilers will schedule load instructions using the cache-hit latency rather than the maximum latency. This

means that the processor will stall on every cache miss. The alternative of scheduling all loads with the cache miss latency is not feasible for most programs because the maximum latency may not be known due to bus contention, or memory port delays, and it also requires considerable ILP. This problem with non-determinism in cache access limits VLIW architectures to become cacheless unless speculative solutions are embraced. This is a significant problem with modern technology, where processor speeds are significantly higher than memory speeds [17]. Also pure VLIW architectures are not good for general purpose applications, due to their lack of compatibility in binary code [32]. The most significant use of VLIW architectures, therefore is in embedded systems, where these constraints are both solved (i.e. single applications using small fast memories). A number of projects described below have attempted to apply speculation to VLIW in order to solve the scheduling problems and one, the Transmeta Crusoe [33], has applied dynamic binary code translation to solve the backward compatibility problem.

The Sun MAJC 5200 [34] is a chip multiprocessor based on four-way issue, VLIW pipelines. This architecture provides a set of predicated instructions to support control speculation. The MAJC architecture attempts to use speculative, thread-level parallelism to support the multiple processors. This aggressively executes code in parallel that can not be fully parallelised by the compiler [35, 36]. It requires new hardware mechanisms to eliminate most squashes (threads are speculatively executed in parallel and if a cross-thread dependence is violated at run time, a corrective action is triggered to repair the state) due to data dependencies [35]. This method of execution is again speculative and can degrade the processor's performance when the

speculation fails. MAJC replicates its shared registers in all pipelines to avoid sharing resources. From the implementation point of view, replicating the registers costs significant power and area [37] and also restricts the scalability. Furthermore, the MAJC compiler must know the instruction's latencies before it can create a schedule. As described previously, it is not simple to detect all instructions' latencies due to the variety of the hardware communication overheads.

2.1.3 Explicitly Parallel Instruction Computing (EPIC)

Intel's explicitly parallel instruction computing (*EPIC*) architecture is another speculative evolution of VLIW, which also solves the forward (although not backward) code compatibility problem. It does this through the run-time binding of instruction words to execution units. The IA-64 [38] architecture supports binary code compatibility across a range of processor widths by utilising instructions packets that are not determined by issue width. This means a scheduler is required to select instructions for execution on the available hardware from the current instruction packet. This gives more flexibility as well as supporting binary code compatibility across future generations of implementation. The IA-64 also provides architectural support for control and data speculation through predicated instruction execution and binding pre-fetches of data into cache. In this architecture each operation is guarded by one of the predicate registers, each of which stores one bit that determines whether the results of the instruction are required or not. Predication is a form of delayed branch control and this bit is set based on a comparison operation. In effect, instructions are executed speculatively but state update is determined by the predicate bit so an operation is completed only if the value of its guard bit is true, otherwise the processor

invalidates the operation. This is a form of speculation that executes both arms of some branches concurrently but this action restricts the effective ILP, depending on the density and nesting of branches.

Prefetching is achieved in a number of ways. For example, by an instruction identical to a load word instruction that does not perform a load but touches the cache and continues, setting in motion any required transactions and cache misses up the hierarchy. These instructions are hoisted by the compiler up the instructions stream, not just within the same basic block. They can therefore tolerate high latencies in memory, providing the correct loads can be predicted. There are many more explicit controls on caching in the instruction set to attempt to manage the non-deterministic nature of large cache hierarchies. Problems again arise from the speculative nature of the solution. If for some reason the prefetch fails, either because of a conflict or insufficient delay between the prefetch and genuine load word instruction, then a software interrupt is triggered incurring a large delay and overhead.

EPIC compilers face a major problem in constructing a plan of execution, they can not predict all conditional branches and know which execution path is taken [39]. To some extent this uncertainty is mitigated by predicated execution but as already indicated, this is wasteful of resources and power and like all speculative approaches can cause unpredictability in performance. Although object code compatibility has been solved to some extent, the forward compatibility is only as good as the compiler's ability to generate good schedules in the absence of dynamic information. Also the code size problem is still a challenge facing the EPIC architecture [39].

2.1.4 Multithreading

In order to improve processor performance, modern microprocessors try to exploit thread-level parallelism *(TLP)* through a multithreading approach even at the same time as they exploit ILP. Multi-threading is a technique that tolerates delays associated with synchronising, including synchronising with remote memory accesses, by switching to a new thread, when one thread stalls. Many forms of explicit multithreading techniques have been described, such as interleaved multithreading *(IMT)*, blocked multithreading *(BMT)* and simultaneous multithreading *(SMT)*. A good survey of multithreading is given in [1].

A number of supercomputers designed by Burton Smith have successfully exploited IMT, these include the Delencor HEP, the Horizon and culminated in the Tera architecture [40]. This approach is perhaps the closest to that of microthreading described in this thesis, although the processor was designed as a component of a large multi-computer and not as general purpose chip. The interleaved approach requires a large concurrency in order to maintain efficient pipeline utilisation, as it must be filled with instructions from independent threads. Unlike the earlier approaches, Tera avoids this requirement using something called *explicit-dependence lookahead*, which uses an instruction tag of 3 bits that specifies how many instructions can be issued from the same stream before encountering a dependency on it. This minimises the number of threads required to keep the pipeline running efficiently, which is about 70 in the case of memory accesses. It will be seen that microthreading uses a different approach that maintains full backward compatibility in the ISA, as well as in the pipeline structure.

Unlike IMT, which usually draws concurrency from ILP and loops, BMT usually

exploits regular software threads. There have been many BMT proposals, see [1] and even some commercial designs such as the Sun's Niagra processor [6]. However the concurrency exposed in BMT architectures is limited, as resources, such as register files must be duplicated to avoid excessive context switching times. This limits the applicability of BMT to certain classes of applications, such as servers.

SMT, is probably the most popular and commercial form of multithreading in use today. In this approach, multiple instructions from multiple threads provide ILP for multiple execution units in an OOO pipeline. Several recent architectures have either used or proposed SMT, such as the Hyper-Thread Technology in the Intel Xeon processor [41] and the Alpha 21464 [42]. As already described, the main problem with an SMT processor is that it suffers from the same scalability issues as a superscalar processor, i.e. layout blocks and circuit delays grow faster than linearly with issue width. In addition to this, multiple threads share the same level-1 I-cache, which can cause high cache miss rates, all of which provides limits to its ultimate performance [16].

2.2 Alternate Approaches

The complexity and the effectiveness of the instruction issue, long wire delay, and the centralised components of those are difficult to scale in existing approaches, requires researchers either to extend the concepts of a superscalar processors or to build new architectures as an alternative to the superscalar processor. The Multiscalar [43] architecture from the university of Wisconsin is an example for extending the concept of superscalar processors. Also, a number of recent projects have attempted to build new architectures as an alternative to a superscalar processor; including the

Reconfigurable Architecture Workstation (RAW) at MIT [44, 45], the Tera-op, Reliable, Intelligently adaptive Processing System (TRIPS) [46, 47] at UT-Austin, and Wavescalar [48] at Washington. These projects attempt to minimise communication costs and try to exploit locality and improve system scalability.

However, these architectures change the baseline processor design drastically, and use a non-conventional architecture design. In contrast, the microthreaded microprocessor model is a general purpose architecture and can be applied to any RISC or VLIW instruction set. This allows backward compatibility of binary-code with no speed-up, and full speed-up from recompiled code that uses additions to base Instruction Set Architecture (ISA) to support the explicit concurrency controls [49]. The following sections explain in more detail some existing and alternative approaches for processor architecture.

2.2.1 Microthreading

The microthreaded model was first described in [50], and was then extended in [8, 17, 49, 51] to support systems with multiple processors on-chip. Like the Tera, this model combines the advantages of BMT and IMT but does so by explicitly interleaving microthreads on a cycle-by-cycle basis in a conventional pipeline. This is achieved using an explicit, context-switch instruction, which is acted upon in the first stage of the pipeline. Context switching is performed when the compiler can not guarantee that data will be available to the current instruction and is used in conjunction with a synchronisation mechanism on the register file that suspends the thread until the data becomes available. The context switch control is not strictly necessary, as this can be signaled from the synchronisation failure on the register read. However, it

significantly increases the efficiency of the pipeline, especially when a large number of thread suspensions occur together, when the model resembles that of an IMT architecture. Only when the compiler can define a static schedule are instructions from the same thread scheduled in BMT mode. Exceptions to this are cache misses, iterative operations and inter-thread communications. There is one other situation where the compiler will flag a context switch and that is following any branch instruction. This allows execution to proceed non-speculatively, eliminates the branch prediction and cleanup logic and fills any control hazard bubbles with instructions from other threads, if any are active.

The model is defined incrementally and can be applied to any RISC or VLIW instruction set. The incremental nature of the model allows a minimal backward compatibility, where existing binary code can execute unchanged on the conventional pipeline, although without any of the benefits of the model being realised.

Microthreading defines ILP in two ways. *Sets* of threads can be specified where those threads generate MIMD concurrency within a basic block. Each thread is defined by a pointer to its first instruction and is terminated by one or more Kill instructions depending on whether it branches or not. Sets of threads provide concurrency on one pipeline and share registers. They provide latency tolerance through explicit context switching for data and control hazards. *Iterators*, on the other hand, define SPMD concurrency by exploiting a variety of loop structures, including for and while loops. Iterators give parametric concurrency by executing iterations in parallel subject to dataflow constraints. Independent loops have no loop-carried dependencies and can execute with minimal overhead on multiple processors. Dependent loops can also execute on multiple processors, exploiting instruction level concurrency but

during the execution of dependency chains, activity will move from one processor to another and speedup will not be linear. Ideally dependency chains should execute with minimal latency and parameters for the concurrency instruction provided by the model, which allow dependencies to be bypassed on interactions executed on a single processor giving the minimal latency possible, i.e. 1 pipeline cycle per link in the chain.

Iterators share code between iterations and use a set of threads to define the loop body. This means that some form of context must be provided to differentiate multiple iterations executing concurrently. This is achieved by allocating registers to iterations dynamically. A *family* of threads then, is defined by an iterator comprising a *start*, and *limit* of the loop over a set of threads. Information is also required that defines the microcontext associated with an iteration and, as each iteration is created, registers for its microcontext are allocated dynamically. To create a family of threads a single instruction is executed on one processor, which points to a thread control block *(TCB)* containing the above parameters. Iterations can then be scheduled on one or more processors as required to achieve the desired performance.

Virtual concurrency on a single pipeline defines the latency that can be tolerated and is limited by the size of the local register file or continuation queue (CQ) in the scheduler. The latter holds the minimal state associated with each thread. Both are related by the two characteristics of the code; the number of registers per microcontext and the cardinality of the set of threads defining the loop body. In this model, all threads are drawn from the same context and the only state manipulated in the architecture is a thread's execution state, its PC and some information about the location of its microcontext in the local register file. This mechanism removes any

need to swap register values on a context switch.

Theoretically, physical concurrency is limited only by the silicon available to implement a CMP, as all structures supporting this model are scalable and are related to the amount of the virtual concurrency required for latency tolerance, i.e. register file, CQ and register allocation logic. Practically, physical concurrency will be limited by the extent of the loops that the compiler can generate, whether they are independent or contain loop-carried dependencies and ultimately, the overheads in distribution and synchronisation that frame the SPMD execution. Note that thread creation proceeds in a two stages. A conceptual schedule is determined algorithmically on each processor following the creation of a family of microthreads but the actual thread creation, i.e. the creation of entries in the CQ, occurs over a period of time at the rate of one thread per cycle, keeping up with the maximum context-switch rate. This continues while resources are available.

The next chapter provides more details about the microthreaded microprocessor model and its concurrency controls. The model has multiple features which make it a good candidate to future scalable and powerful CMPs.

2.2.2 Multiscalar

Another paradigm to extract even more ILP from sequential code is the *multiscalar* architecture. This architecture extends the concepts of superscalar processors by splitting one wide processor into multiple superscalar processors. In a superscaler architecture, the program code has no explicit information regarding ILP; only the hardware can be employed to discover the ILP from the program. In multiscalar, the program code is divided into a set of tasks or code fragments, which can be identified

statically by a combination of the hardware and software. These tasks are blocks in the control flow graph of the program and are identified by the compiler. The purpose of this approach is to expose a greater concurrency explicitly by the compiler.

The global control unit used in this architecture distributes the tasks among multiple parallel execution units. Each execution unit can fetch and execute only the instructions belonging to its assigned task. So, when a task misspredict is detected, all execution units between the incorrect speculation point and the later task are squashed [52]. Like superscalar, this can result in many wasted cycles, however as the depth of speculation is much greater, the unpredictability in performance is correspondingly wider.

The benefit of this architecture over a superscaler architecture is that it provides more scalability. The large instruction window is divided into smaller instruction windows, one per processing unit, and each processing unit searches a smaller instruction window for independent instructions. This mitigates the problems of scaling instruction issue with issue width. The multiple tasks are derived from loops and function calls, allowing the effective size of the instruction window to be extremely large. Note that not all instructions within this wide range are simultaneously being considered for execution [43]. This optimisation of the instruction window is offset by a potentially large amount of communication, which may effect the overall system performance.

Communication arises because of dependencies between tasks; examples are loop-carried dependencies and function arguments and results. Results stored to register, which are required by another task, are routed from one processor to another at run time via a unidirectional ring network. Recovery from misspeculation is achieved by

additional hardware that maintains two copies of the registers along with a set of register masks, in each processing unit [53]. In summary then, although the multiscalar approach mitigates against instruction window scaling allowing wider issue width, in practice it requires many of the same complex mechanisms as superscalar and being speculative is unlikely to be able to perform as consistently as a scalable CMP.

2.2.3 Intrathreads

The intrathreads [54] or inthreads represent a context of computation (independent threads of control) executing simultaneously on the processor. Thus, the processor holds a context and this context contains information necessary for its execution. A set of condition registers are used for synchronisation between intrathreads and to suspend the inthread's context until a specific condition has been resolved. The architecture defines a set of instructions for creation, synchronisation, and termination. Generally the intrathread architecture is similar to SMT, where it tries to operate on a low level of ILP by using a shared registers for data communication rather than shared memory as in SMT.

Intrathreads adopt the same principle as microthreads but with a different approach to implementation. The architecture supports a fixed number of intrathreads. In fact, the intrathread architecture has many limitations and ends up requiring many of the same complex mechanisms as SMT such as: complex issue window, register renaming, speculative execution, and recovery mechanisms to handle misspeculation of branches which affect instructions in several threads.

As described in [24], there are many differences between intreathreds and microthreads. First, intrathreads use bounded concurrency and statically-partitioned

resources, while microthreads describe parametric concurrency and the resources are managed dynamically through the concept of microcontexts. Secondly, intrathreads separate synchronisation and data storage, while a microthreaded processor implements registers as *i-structures* to synchronise between code fragments. In addition, Inthreads have a limited number of threads, and the implementation targets a wide issue pipeline rather than a chip multiprocessor.

The microthread model requires dynamic register allocation and a hardware scheduler, which can support hundreds of microthreads per processor and their associated microcontexts. The allocation of these microthreads is dynamic; being determined by resource availability, as the concurrency exposed is parametric and not limited by the hardware resources. The instruction issue schedule in the microthreaded model is also dynamic and requires linear hardware complexity to support it.

2.2.4 Raw Machine (RAW)

The RAW processor is a single chip consisting of 16 identical single issue processor tiles connected by a mesh interconnection network. Each tile in the mesh contains a data and instruction memory, register file and an 8-stage in-order pipeline.

The RAW architecture and its compiler tries to exploit ILP within basic blocks of code. The architecture supports data and ILP by explicitly distributing computation to different tiles and running them as different threads (independent instruction streams). In order to route a value between two tiles a static router is used to set up an appropriate path between the source tile and the destination tile on the static network. In fact, this architecture exploits TLP rather than ILP by using independent instruction streams. Also, the RAW compiler puts all statically predictable

communications on the static on-chip network and the ordering never changes [56], the problem is how to adapt this method for dynamic code to evaluate ILP with considerable data dependencies.

The RAW architecture not only statically orders communication between the 16 tiles, but also statically partitions code onto the tiles. One draw back of this architecture is the inter-node communication latency, which is extremely sensitive and high. The RAW architecture suffers a three cycle penalty in the case of misspreddiction or inter-tile ALU-to-ALU operand delivery and up to 54 cycles for L1 cache miss latency [56]. In contrast, microthreading provides a mechanism to avoid any delays in instruction cache misses and is also fully decoupled from any remote accesses, including memory access.

2.2.5 Explicit Data Graph Execution (EDGE) and TRIPS

The Explicit Dataflow Graph Execution *(EDGE)* instruction set architecture is another approach targeting a scalable issue width, which tries to turn thread and data level parallelism to ILP, and attempts to minimise global communication delays. The TRIPS architecture is an evolution of the EDGE ISA, which uses a dataflow order execution and its architecture contains two OOO, 16-wide issue processor cores.

The program graph in TRIPS EDGE architecture is broken explicitly by the compiler into a sequence of blocks called hyperblocks [57]. Each block is fetched from the memory at run time and is scheduled independently. The compiler is responsible for statically placing this block of instructions into the issue window and mapping each block into the array of execution nodes. The renaming logic at the register file bank is used to forward register values that one block produces directly to be

consumed in another block.

Within a block, the TRIPS ISA supports large graphs of computation mapped to hardware components, with instructions in each graph communicating directly with other instructions, rather than going through a shared name space. The hyperblocks are scheduled sequentially with conventional control-flow semantics, then allocated to processors in a cluster statically. Because of this partioning between data flow and sequential semantics, the approach does not scale seamlessly [55]. Subsequent hyperblocks are selected speculatively, and executed concurrently. Scaling the hardware will require scaling the hyperblocks that provides the data flow concurrency, which is a compile-time optimisation and would require frequent recompilation [55]. The architecture is similar to wavescalar in that both use of direct communication between instructions of the same hyperblock.

A TRIPS compiler unrolls loops statically to extract higher levels of concurrency up to its execution width. It is also focused on statically mapped parallelism, which is automatically extracted by the compiler. Conversely, in the microthreaded model, parametric concurrency based on loops can be expressed through the ISA, using a control block associated with the *Cre* instruction. Therefore, the concurrency is not limited by hardware constraints.

2.2.6 Wavescalar

Wavescalar [48] is a tagged-token dataflow architecture. Instructions execute in sequence and according to the dataflow firing rule. Wavescalar instructions execute in-place in the memory system and explicitly send their results to their dependents.

Thus, wavescalar instructions are cached in the processing elements. The main motivation of wavescalar architecture was to build a distributed superscalar processor core in order to provide a scalable issue window and to avoid a long wire delays problem. It also attempts to solve the problems of source language and concurrency expansion. It does this by introducing a wave number across multiple instances of a given context such as a loop or function call. This sequentialises execution and provides a mechanism for resolving multiple writes to the same variable, something not allowed in single assignment languages [55].

This architecture relies on the compiler to minimise the communication delays by minimising the physical distance between the dependent operands and hence minimising the execution time. Execution of instructions occurs in a desired order within each wave. Wave number tags are used in identifying each individual instance of data used when executing the program. Thus, the Wavescalar architecture uses a wave-ordering memory mechanism to order memory operations by statically assigned unique sequence numbers for the predecessor and successor operations. Wavescalar dynamically groups multiple instructions as a block and assigns this block to a fixed number of processing cache elements. The Wavescalar approach however, still suffers from inefficiencies in managing control flow and will typically execute more instructions for a given computation than are executed in a RISC processor [55]. Also, there is no flexibility in the execution and this follows from the adaptive ordering which reduces the parallelism.

2.3 Recent CMPs

From the above discussion we see that most current techniques for exploiting concurrency suffer from software and/or hardware difficulties, and the focus of research and development activity now seems to be on chip multiprocessors *(CMP)*. These designs give a more flexible and scalable approach to instruction issue, freeing them to exploit Moore's law though system level concurrency. Some applications can exploit such concurrency through the use of multithreaded applications. Web and other servers are good examples; however, the big problem is how to program CMPs for general purpose computation and whether performance can ever be achieved from legacy sequential code, either in binary or even source form.

Several recent projects have investigated CMP designs [2, 3, 4, 58]. Typically, the efficiency of a CMP depends on the degree and characteristics of the parallelism. Executing multiple processes or threads in parallel is the most common way to extract a high level of parallelism, but this requires concurrency in the source code of an application. Previous research has demonstrated that a CMP with four 2-issue processors will reach a higher utilisation than an 8-issue superscalar processor [1]. Also, work described in [4] shows that a CMP with an eight 2-issue superscalar processor would occupy the same area as a conventional 12-issue superscalar. The use CMPs is a very powerful technique to obtain more performance in a power efficient manner [59]. However, using superscalar processors as a basis for CMPs; with their complex issue window, large on chip memory, large multi-ported register file and speculative execution is not such a good strategy because of the scaling problems already outlined. It would be more efficient to use simpler in-order processors and exploit more concurrency at the CMP level, provided that this can be utilised by a sufficiently wide range

of applications. This is an active area of research in the compiler community and until this problem is solved, CMPs based on user-level threads will only be used in applications which match this requirement, such as large server applications, where multiple service requests are managed by threads.

2.4 Microarchitecture and Architecture Challenges

As described above, CMPs architectures must overcome multiple challenges if they are to deliver their full potential. In this section, we provide more detail on these challenges, and outline some of the existing approaches to solving them.

2.4.1 Scalability and Performance Improvement

To keep multiple execution units as busy as possible in the presence of significant latency in obtaining operands, modern processors use an aggressive OOO instruction-execution. This allows instructions to be issued and completed out of the original program sequence, thereby exposing concurrency in the legacy, sequential instruction stream. OOO execution increases the performance of a superscalar processor by reducing the number of stall cycles in the pipeline. Synchronisation is managed by the instruction-issue logic, which keeps track of resources required by an instruction and any dependencies on the results of other instructions, which may not yet have been scheduled or completed. The instruction window maintains the set of decoded instructions on the currently predicted execution path that have not yet been issued. Its logic triggers those instructions for execution but requires an area that is quadratic in issue width, i.e. the number of instructions that can be issued simultansously [12].

Other support for renaming registers and retiring instructions adds to this cost. The key problem is that the mechanism for synchronisation is centralised.

Monolithic processors (i.e. wide-issue from a single instruction stream) have other structures that do not scale, these are the register file [60] and bypass logic [14], which are also centralised. Finally, the concurrency exposed in OOO instruction execution is limited due to the inefficient use of the instruction window. In practice its size is limited by scalability constraints but its use is required for all instructions, independent of whether those instructions can be statically scheduled or not.

SMT is an attempt to make more efficient use of OOO scheduling by fetching instructions into the instruction window from a number of independent threads, thus guaranteeing fewer dependencies between the instructions found there and hence allowing more efficient use of the wide instruction issue. However, it does not address any of the issues outlined above and suffers from the same scalability problems as conventional OOO processor, i.e. layout blocks and circuit delays grow faster than linearly with issue width, and synchronising memory is used inefficiently. Indeed it introduces other problems, such as multiple threads that share the same level-1 I-cache, which can cause high cache miss rates, all of which limit the ultimate performance [16].

The latency across a memory hierarchy may require hundred of cycles, which can significantly impact performance. The only way to avoid an impact on a processor's performance is to provide instruction-level concurrency, in addition to wide instruction issue, to provide tolerance to this latency. That, by definition, requires hundreds of independent instructions per processor. With CMPs comprising thousands of processors, this means providing synchronising memories capable of supporting hundreds of thousands of synchronisation will be required in future CMPs and they must be

designed with this in mind.

An alternative approach to on-chip concurrency is to exploit user-level threads rather than dynamically extracting concurrency from legacy binary code. Sun has proposed a commercial, 32-way threaded version of the SPARC architecture in its Niagara device. The chip has eight cores, each able to handle the contexts of four threads. Each core has it own L1 cache and all cores share a 3MB L2 cache. Key problems with the Niagara approach are the significant resource consumption for the aggressive speculative techniques used, and the significant time wasted waiting for off-chip misses to complete, see [61]. Also, the basic implementation of this SPARC chip is a superscalar processor and, as already described, the superscalar approach provides diminishing returns in performance for increasing issue width. The performance of a 6-issue OOO processor will achieve only 1.2 to 2.3 IPC, compared with 0.6 to 1.4 IPC in a 2-issue processor [13].

It should be noted that Niagara is better suited to server rather than general-purpose workloads, as a profusion of high-level threads are available in server applications, e.g. where a server's users are each managed by a concurrent thread. However, for general purpose workloads, typical programs are not so heavily threaded and unless an automatic means of generating them can be found, this will severely limit the software-thread approach.

2.4.2 Concurrency and Programmability

Exposure and management of concurrency are the key issues in supporting CMP design and implementation. This is the case for distributed systems as well chip-level

systems, but in the latter situation, the constraints and opportunities dictate a different approach that is able to minimise the overheads of managing that concurrency. Concurrency has the ability to increase overall system performance as well as provide power savings in obtaining a given performance, by scaling frequency and voltage.

The use of OOO instruction execution to expose and manage concurrency is ideal in one respect and one respect only. That is the ability to obtain concurrency from legacy code, without the programmer having to be aware of it. This has great commercial appeal. However, the model has no tacit knowledge of concurrency and synchronisation and this must be extracted dynamically in hardware, using complex support structures, not all of which scale with issue width. This is wasteful of chip resources, does not have predictable performance and is not able to conserve power in the execution of instructions. If concurrency were explicitly described in the instruction stream, some of these unscalable structures could be avoided.

User-level concurrency based on threaded applications is one alternative solution exemplified by the Niagara described above, but not all codes contain thread-level concurrency and therefore tools are required to extract threads from sequential programs. One example of such tools is the use of speculative, pre-execution threads to provide latency tolerance in memory access. This can be performed statically by a compiler, dynamically in the hardware, or indeed by a hybrid of the two [62]. However, as its name suggests, the model is speculative, which can again result in unpredictable performance and, like all speculative methods, is not conservative in its use of energy, e.g. when the speculation fails.

An alternative approach to extracting threads from user-level sequential code is described in [63], which compiles legacy applications for a multithreaded architecture.

The most important goal of this work is to create a sufficiently large number of threads so that there is sufficient parallelism to hide communication latency. A second goal is to create threads of a sufficient granularity so that the context switching cost is relatively small compared with the cost of the actual computation. These goals are contradictory but can be achieved by distributing remote data dependencies between different threads and using these dependencies to schedule the thread when data dependencies are resolved, i.e. by using non-blocking threads. The approach described here, microthreading, has extremely efficient context switching and consequently does not require threads to be non-blocking.

Most approaches to extracting concurrency use the well-known fact that most computation is performed in loops and that loop iterations can often be performed concurrently, LLP. Compilation can extract software concurrency, as [63], or provide instruction-level concurrency as in the case of microthreading, which has an ISA extension for the compiler to target; this instruction dynamically creates a whole family of threads. Alternatively, in conjunction with control speculation, loops facilitate the concurrency exposed in OOO instruction execution by using branch prediction.

However, not all loops are independent and concurrency is often limited by data dependencies, which may arise between different loop iterations when executed concurrently. The vector computers of the 1970s and 1980s were unable to deal with LLP that involved dependencies. OOO instruction execution, on the other hand, manages these dependencies, which are often regular, just like any other irregular dependency. It has no contextual data to optimise and structure such management. There are other explicit approaches to manage loops containing data dependencies [3, 15] but in these, loop-carried dependencies are expressed as concurrently executing threads

that share memory. This is bad as it induces high latencies in the dependency chains. In contrast, microthreads, synchronise in registers rather than in memory but this requires large register files as well as large support structures. This can only be achieved using distributed structures and in microthreading, unlike monolithic wide-issue approaches, synchronisation and scheduling are managed by distributed register files and schedulers even though the concurrency is specified and managed at the instruction level.

The requirements on these support structures are severe; they must support a context switch on every cycle, as the compiler identifies context switch points in the code and can flag any instruction to context switch. They must also support thread creation on every cycle, as thread creation occurs concurrently with instruction execution and must keep pace with the rate at which context switches can occur. Finally, they must support thread rescheduling at one thread per cycle, as when all threads are created, rescheduling must also keep up with context switch rates.

2.4.3 Scaling Processor Support Structures

In superscalar processors, the logic necessary to handle OOO instruction issue typically occupies 20-30% of the chip area [13] and the issue logic in processors that support speculation can be responsible for 46% of the total power [64]. On-chip caches are another critical challenge in modern processors, occupying large die areas, consuming significant power, and in some cases restricting system performance and scalability. Large cache bandwidth requirements and slow global wires will sharply diminish the effective performance of processors sharing a large monolithic cache as advances in fabrication processes effectively decrease global propagations times.

The alternative is to build thousands of processing elements on a die and surround each with a small amount of fast storage. Compare this to Intel's Montecito processor where cache memory occupies some 70% of the total die area or the equivalent to 32,000 32-bit integer ALUs (0.18μm technology). Huh et. al. [65] address this issue by comparing the architectural trade-offs between in-order and OOO issue processors for serial applications. Their study demonstrated that if no L-2 cache area were required, then it is possible to integrate 556 2-way in-order processors on a single chip, or 201 4-way OOO processors with a maximum area of $400mm^2$ in 35nm technology.

Clearly, the use of ever larger hierarchical memory systems does not serve scalability and does not guarantee better performance. Instead, as argued above, there is a requirement for large, fast and distributed synchronisation memory to support very wide instruction issue as well as latency tolerance. Ideally a deterministic distribution of instruction execution and data mapping is required in order to explicitly manage locality and to eliminate, as far as is possible, slow and power-hungry global communication. This goal is not served by using a large and monolithic processors connected to a large and monolithic on-chip memory. In short, some form of distribution becomes essential and without a deterministic distribution of data and computation on chip, very wide-issue CMPs are just not feasible.

Rixner et. al. [22] analysed register file area, delay and power dissipation for streaming applications. The analysis showed that for a central register file, area and power dissipation grows as N^3 and delay grows as $N^{3/2}$. Examples of the effects of this scaling can be found in the proposed Alpha 8-way issue 21464, which used a 512 location register file requiring 24 ports to serve the wide-issue processor. Even with a clustering, the 4Kbytes of register file occupied an area some 5 times larger

than that used by the L1 D-cache [66] (64KB plus tags). That is a per-bit, density ratio of 100:1 and graphically illustrates Rixner's results. Power is also an issue in such large structures and in Motorola's M.CORE architecture, the register file energy consumption is 16% of the total processor power and 42% of the data path power [67]. These examples, which support only modestly-wide instruction issue, confirm that multi-ported register files in modern microprocessors are not the way to proceed in future CMPs.

ILP processors communicate and synchronise using a namespace interpreted at the instruction level, i.e. the register specifiers. This is typically limited to 5 or 6 bits and the question that must be asked is how can a large and distributed synchronisation memory be addressed with such small addresses? OOO processors use register renaming for subsequent uses of the same register specifier and thus expand the namespace dynamically. (This also removes the artificial dependencies introduced by executing instructions out of program sequence). Of course, additional hardware is now required to perform this mapping and to re-establish the mapping back to the original binary code to give the illusion of sequentially executed instructions.

Microthreading on the other hand executes loops as concurrent code fragments and in order to share code for a loop body, each iteration must have its own registers, which are unique. In contrast to renaming, this is achieved by addressing a register file relative to some unique offset, so that the same instruction will access a different location in the register file for different iterations. Those offsets are a part of the microthread's state. This mechanism extends the ISA's namespace so that it is limited only by the parametric concurrency expressed in the creation of the microthreads that execute the loop.

2.4.4 Power Dissipation

Two challenges in modern processors is power consumption and heat dissipation, which are already a serious problems and will only become worse in future [68]. For example, Intel's Madison consumes up to 130W, the alpha 21364 EV-7 consumes 155W and the International Technology Roadmap for Semiconductors expects that power consumption in processors will reach close to 300W by 2015 [21]. This 300W does not follow the past exponential growth in power dissipated and recognises this as a major constraint on processor design. This problem is exacerbated as in future process technologies, the leakage power will also become a significant percentage of the overall power dissipated [69].

Several researchers have considered power reduction in CMPs [68, 69] but these techniques can not hope to find significant principle solutions as branch prediction and OOO issue do not provide a significant performance improvement relative to the area and power consumed and do not execute instructions conservatively with respect to power dissipation. Indeed, the only solution is to remove these features to save power [70]. Another current trend that highlights this problem is the current practice of increasing the number of pipeline stages in order to reduce the clock period and hence increase performance. This also can not continue, as it is simply not feasible to continue to extract exponentially growing amounts energy from a chip as heat, as the result of power dissipated. Indeed, there is a case for the trend to higher and higher clock frequencies to be stopped or even reversed and instead to use concurrency as a means of providing performance improvements without excessive power consumption.

Concurrency can also provide power reduction for a given performance. With a scalable processor, two processors acting concurrently should give the same overall

performance as one at double the speed. The scalability required is performance with issue width, logic or area with issue width and power dissipated with instructions issued per cycle (IPC). Although the above comparison breaks even in power dissipated, power can be saved by scaling supply voltage with frequency. As power dissipated is proportional to V_{DD}, this gives a quadratic reduction in power for a given performance, over the linear portion of frequency-voltage scaling.

The use of IPC rather than issue width as a base for power scaling assumes that when a processor is inactive it can be powered down. As a result, speculation or eager execution must be avoided, as by definition an eager processor can never determine when there is nothing to do!

Microthreading uses simple in-order instruction issue without branch prediction and has explicit control of instruction scheduling, it can therefore provide all the hooks required to support conservative instruction issue and hence take advantage of this power scaling [24]. Processors with no active threads are aware that instructions can not be scheduled and can therefore go into standby mode dissipating minimal power. This power usage can be scaled with IPC rather than issue width.

This conservative scheduling also provides an insight into asynchronous partitioning of a CMP. By definition, if a processor has all of its threads inactive, then any event triggering further computation must either be external to the processor (asynchronous) or the processor must be deadlocked. A microthreaded CMP can therefore use a local clocking with asynchronous communication between processors, further reducing power requirements. This fact, together with the processor's inherent latency tolerance provides all the hooks required to implement a globally-asynchronous, locally-synchronous (GALS) implementation. Additional power savings come from

not requiring such powerful drivers in distributing the clocks to the entire chip.

2.5 Distributed Memory Multiprocessor Architecture

It is very well known that most current multiprocessor systems organise their processors and memory using one of the two architecture methods [71], Uniform-Memory Access (UMA) and Non-Uniform Memory Access (NUMA). In an UMA architecture, as shown in figure 2.1, multiple processors links up to a global memory storage through a common system bus. The access times to this memory from each processor are the same, hence the name UMA. This memory architecture has the advantage of being easy to program as there is no explicit communication between processors and all communications are handled through a global memory system. However, this architecture does not scale well and has a communication bottleneck when multiple processors attempt to access the centralised resource (system bus or global memory) at the same time.

The second alternative memory organisation is NUMA, and it also known as a Distributed Shared Memory *(DSM)* architecture. The general structure of NUMA architectures is shown in figure 2.2, which avoids the drawbacks of the UMA architecture and allows the construction of large scalable machines [72, 73]. This architecture can be constructed as a clustered or shared local memories as shown in figures 2.2a and 2.2b respectively. In a clustered configuration, each cluster is itself an UMA or a NUMA multiprocessor, where all processors belong to the same cluster and have a uniform access to the memory attached to it within the cluster. The interconnection

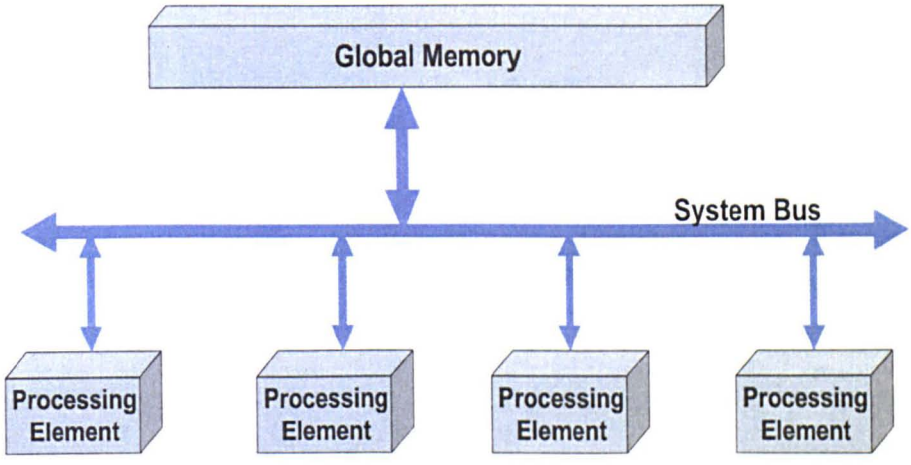


Figure 2.1: UMA architecture model.

network is used by each cluster to connect other remote clusters. In shared local memories, each processor accesses its local memory which is attached directly to it and accesses the high performance interconnection network for the remote data. The access time in this memory architecture varies, hence the name Non-Uniform Memory Access *(NUMA)*. Access to the local memory can occur much faster than the remote memory (due to the different physical distances), and the latter is effected by the way the processors are connected.

As described in [74], the distributed-memory multiprocessor architecture is essential in developing massively parallel machine, however one of the most important design issues in such a distributed memory multiprocessor architecture is a latency problem, which is caused by remote memory access. This problem forces the processes to suspended their execution until the response to remote memory is received. Such a design strategy places a greater challenge on the memory system, where on average memory operations account for about third of all instructions [75]. The long latency across the memory hierarchy in modern processors requires hundred of cycles

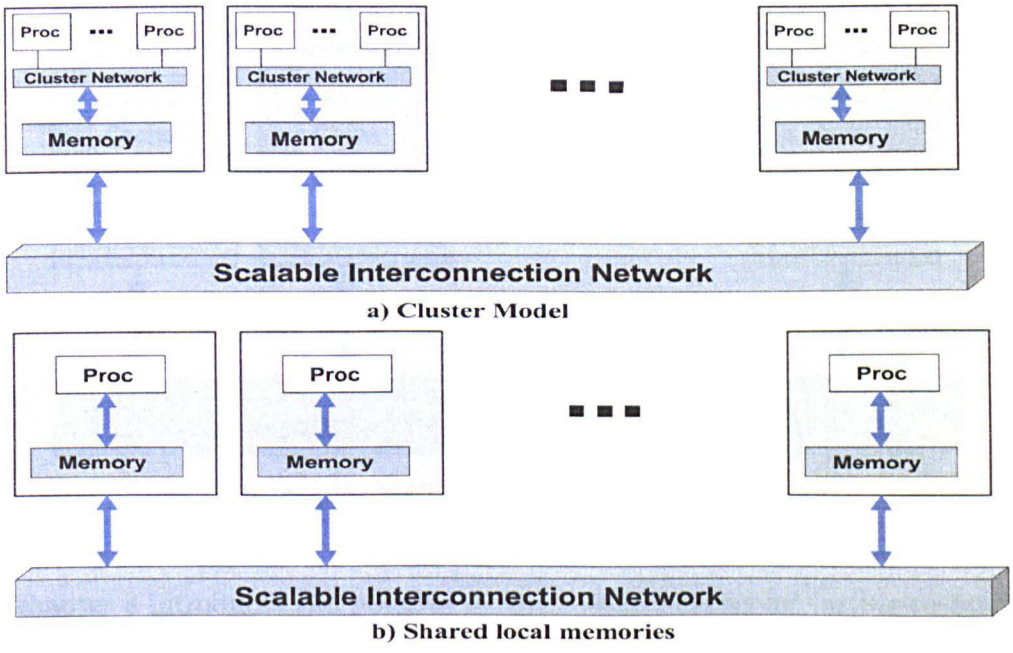


Figure 2.2: NUMA models system architectures.

to traversed data, which significantly impacts performance. For example, in 1-GHz microprocessors accessing main memory can take about 100 cycles and such access may stall a pipelined processor for many cycles [23]. Therefore, the memory system is an important design issue, which must be considered carefully in designing any scalable system on-chip.

The latency tolerance provided by microthreaded microprocessor model makes the design of the memory system somewhat flexible. For example, a large, banked, multi-ported memory would give a solution that would provide all the buffering required for the large number of concurrent requests generated by this model. It is important to note that using in-order processors and a block-based memory consistency model, memory ordering does not pose the same problem as it does in an OOO processor. The following subsections cover some design choices for distributed memory architectures

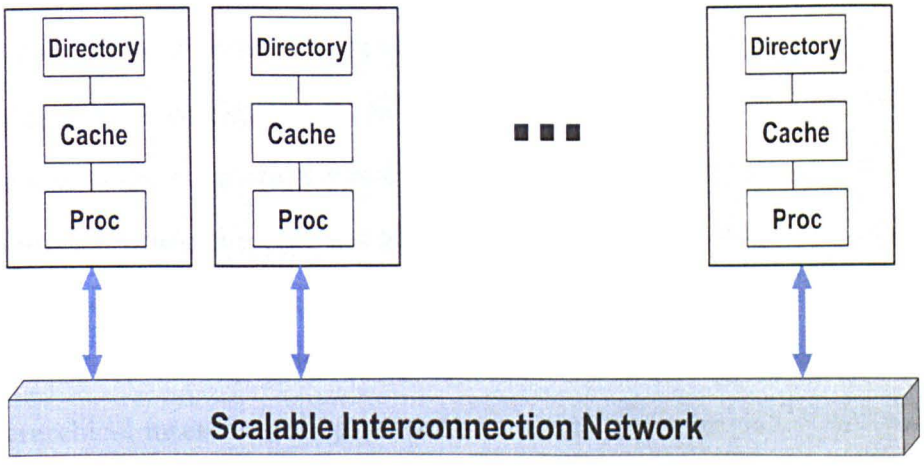


Figure 2.3: The COMA architecture model.

and chapter 8 introduces two possible memory organisations for further research for distributed memory architectures for microthreaded CMP.

2.5.1 Cache Only Memory Architecture (COMA)

One possible organisation of memory modules is to use a cache only memory *(COMA)* architecture [76, 77, 78, 79, 80, 81]. The COMA model is a distributed shared memory and it is a special case of NUMA architecture. COMA attempts to improve memory bandwidth by organising the local memory as a large cache, called an attraction memory *(AM)*, without traditional main memory. COMA provides the ability to automatically copy or migrate data and replicate it to where it is being used by the processors [79, 80]. In fact, COMA has multiple advantages [82] for scalable distributed shared-memory and the first commercial COMA architecture was the KSR-1 [83, 84], which used a hierarchical ring interconnection network.

Figure 2.3 shows the COMA architecture model, where each processor has a local cache and a virtual part of the shared memory and all the caches form a global address

space. Distributed cache directories are needed to store the coherence information and there are two types of directory organisation: *hierarchical* and *flat* [85, 86]. In the Hierarchical-COMA configuration, each directory must keep coherence information about the rest of the subhierarchy nodes. Thus, a long latency occurs when a request traverses up and down through the hierarchy to search for a desired memory block, or when replacements are required.

The Flat-COMA configuration is an alternative directory organisation, which uses a non-hierarchical interconnection network to search for a desired memory block. In this configuration, the memory block can migrate or replicate to any memory node, but the directory entries must remain fixed in their home nodes. Thus, instead of traversing a hierarchical interconnection network to find the memory block at a miss condition in an AM, an enquiry request goes to the home directory. If the home directory does not have a copy from the required memory block, the request is forwarded to the next directory home.

El Naga et. al. [87] proposed the multithreaded COMA (MCOMA) architecture with Flat-COMA, which uses a scalable non-hierarchical interconnection network to connect all processing nodes. In this architecture, all the group directories are connected through a dedicated search interconnection network for fast data search, while the processors and memory modules communicate through a separate interconnection network. In this architecture, each group of processors are connected to its local directory node, where the directory contains information about the data item physically allocated in the node of its cluster group of processors.

2.5.2 Multibanking

Attempts have been made to improve memory performance by using different cache implementation techniques [75, 88, 89, 90]. For example cache replication, in which multiple copies will allow multiple load instruction to be satisfied simultaneously as in Alpha 21164 Microprocessor [91]. However, this technique only improves the load instruction, while the store instruction is still a bottleneck. Also, from the hardware point-view, replicating multiple copies means more die area and more power consumption. Therefore, this technique is costly, and does not scale well.

Another technique is multibanking, where a cache is divided into multiple cache banks, so that multiple banks can be accessed simultaneously. The MIPS 10000 [92] is an example of a 2-bank (interleaved) data cache, where a pair of memory references can address different banks at the same time. However, one well known problem with multibanking is bank conflicts when multiple memory requests goes to the same memory bank. An alternative solution to bank conflicts is to use bank prediction techniques. Several researchers have proposed bank predictors based on branch prediction such as [93, 94]. However, these techniques still suffer from the problem of erroneous bank prediction, where a misspredict requires a recovery mechanism to steer the misspredict load or store to the proper queue [94, 95]. Other proposals [89, 96] described a combined mechanism to avoid bank conflicts, where extra hardware is required to detect and combined multiple memory accesses to same cache bank. Typically, multibanked memory uses a crossbar switch interconnection network to distribute memory references across cache banks. The crossbar switch is cost-effective because its die area increases super-linearly with increasing number of cache banks [75].

Table 2.1: Current and upcoming Microprocessors.

Processor	Processor Speed GHz	L1-Cache Size	L2-Cache Size	L3-Cache Size	Process Technology nm	Transistor Count Million	Area Size mm2
AMD Athlon64 FX	2.2	64KB/ 64KB	1MB	None	130	105.9	193
Intel McKinley	1.0	32KB/32KB	256KB	3MB	180	221	421
Intel Madison	1.5	16KB/16KB	512KB	9MB	130	500	374
Intel Montecito	1.87	64KB/64KB	2MB	24MB	90	1720 (1.2 bil in cache)	580
Intel Pentium4 (Northwood)	2.0	8KB/8KB	512KB	None	130	55	146
Intel Pentium4 (Northwood)	3.2	8KB/8KB	512KB	2MB	130	190	237
Alpha 21364 Ev-79	1.6	64KB/64KB	1.75MB	None	130	100	300
HP PA-8700	.875	2.25MB	None	None	180	186	304
IBM Power5	2.5	32KB/32KB	1.9MB	36MB (Off-Chip)	130	276	389
Alpha 21464	1.2-2	64KB/64KB	1.75-3MB	None	175	250	420

Rivers et. al. [75] proposed a multibanking scheme as a cost-effective alternative to multi-porting caches. In this scheme, n independent single ported cache banks are employed to get an n ported data-cache memory. The cache-line addresses are interleaved through the banks and the banks are accessed in parallel. Also, the authors proposed a combined technique, to avoid bank conflict when simultaneous accesses are going to the same cache line.

2.6 Techniques and Evaluation Methodology

In this section, we will discuss some techniques such as chip area estimation and the simulation environment that is employed in our work to evaluate the microthreaded CMP support structures.

2.6.1 Chip Estimate Area Model

Current technology continues to follow Moore's law and provides a doubling of the number of transistors integrated on a single chip every two years. To fulfill Moore's law's predictions, silicon chip designers continue to shrink the feature size of the silicon chip to increase number of transistors. However, as described earlier, existing microarchitecture designs are reaching a limit in performance and it is now very hard for these designs to scale properly.

Generally, the die size of the processor refers to its area size on the wafer, which measured in square millimeters *(mm^2)*. Table 2.1 shows a summary of selected die size of some existing and upcoming microprocessor chips. The table also shows the memory hierarchy storage sizes and the transistors count in each processor chip.

Unfortunately, an efficient analysis of area requires an accurate analytical model to predict the costs of the various architecture parameters. Work published by [97] described a chip area model for register files and caches. Also, Gupta et. al. [98] derived a set of technology-independent area models, by measuring die photographs of commercial microprocessors and normalising the results. In addition, Standard Cell Datasheets [99] provide an estimate of area for digital combinational logic circuits.

Moreover, the Cache Access and Cycle Time model *(CACTI)* [100] includes an area model for different cache configurations along with their process technology. The

analysis includes the area of decoders, bit cells, input and output circuits and routing tracks. Also, a technology-independent area model in [98] identifies and summarised the areas and sizes for various instruction and data caches. Our work therefore, uses these techniques to provided area estimates of the microthreaded support structures and the processor core.

2.6.2 Simulation Environment

The validation of the hardware design requires functional simulation, in order to assess the overall system performance and to check the correctness of its behaviour. One popular approach to simulation for hardware components is to use the hardware description language VHDL. This language has become widely accepted, is commonly used in industry and can be used to target FPGA-like logic devices. It provides several advantages such as good verification for the behaviour of the system components, easy code construction, and modification and accurate pre-layout simulation results for the system hardware components.

The base MIPS processor is already modelled using VHDL as a hardware description language and synthesis tool [101]. Thus, our implementation results in this thesis are described in VHDL. In particular, we used Symphony EDA VHDL compiler and simulator [102] to model and implement the microthreaded CMP components described throughout this thesis. This simulator is a leader in HDL simulation technology and produces verification solutions for the hardware system components. It is important to note that our design strategy in writing the VHDL code is that it is important first to define the logic design of the mechanism that must be employed, and then the corresponding high-level language coding methodology can be described.

2.7 Summary

The chapter reviewed several existent processor architectures, and highlights the complexity and challenges that limits existing approaches from being scalable. We also discussed distributed memory architecture organisation. Also, techniques such as the chip estimate area model and simulation environment that are employed in our work are presented.

The characteristics of advanced integrated circuits (ICs) will in future require powerful and scalable CMP architectures. However, current techniques like wide-issue, superscalar processors suffer from complexity in instruction issue and in the large multi-ported register file required. The complexity of these components grows at least quadratically with increasing issue width; also, execution of instructions using these techniques must proceed speculatively, which does not always result in efficient power consumption. In addition, more on-chip memory is required in order to ameliorate the effects of the so called "memory wall" [103]. These obstacles limit the processor's performance, by constraining parallelism or through having large and slow structures. In short this approach does not provide scalability in a processor's performance, on-chip area and power dissipation.

An alternative solution, which eliminates this complexity in instruction issue and the global register file, and also avoids speculation, is presented in this thesis. This model supports concurrent threads all drawn from a single context and exploits instruction level parallelism across loop bodies using a variety of loop structures, including static, for loops and dynamic while loops. The model is based on decomposing a sequential program into small fragments of code called microthreads, which are scheduled dynamically and which can communicate and synchronise with each other

very efficiently.

The concurrency controls used in this approach provide latency hiding in a microthreaded processor pipeline and also support a pre-fetching mechanism that avoids any instruction cache misses. An important feature of the model is its support for a fully distributed register file where the latency tolerance decouples register access from pipeline operation. Microthreaded chip multiprocessors add a means of exploiting legacy code in such systems. Using this model, compilers generate parametric concurrency from sequential source code, which can be used to optimise a range of operational parameters such as power and performance over many orders of magnitude, given a scalable implementation. The next chapter reviews this model and describes its concurrency controls in more detail. It also highlights the problems that the work in this thesis resolves.

Chapter 3

Microthreaded Microprocessor Model

3.1 Chapter Overview

In the previous chapter, it was shown that most existing approaches have multiple limitations and systems based on these approaches do not scale well. An alternative approach that supports a scalable CMP design is the microthreaded microprocessor model. In this chapter we consider the microthreaded concurrency model, describe its features that support the implementation of a scalable CMP, and highlight the problems that will be resolved in chapter 4.

The chapter is organised as follows. In the next section, an overview of the microthreaded model is presented. The microthreaded ISA and microthreaded in-order execution pipeline are described in sections 3.3 and 3.4 respectively. A detailed description of the concurrency controls provided by this model is presented in section 3.5. Scalability of the instruction issue and thread state are described in section 3.6 and 3.7 respectively. In section 3.8, microthreaded register file partitioning and distribution is documented. The register allocation unit (RAU) and the method of dynamically

allocating registers are presented in section 3.9. A prefetching and replacement mechanism that avoids any instruction cache misses provided by microthreaded model is described in section 3.10. The summary of the chapter is presented in section 3.11.

3.2 The Microthreaded Model

Microthreads are small sequences of code (as short as a single, executable instruction) that are created dynamically and execute concurrently. Creation is by an instruction added to the ISA for that purpose. A family of microthreads can be distributed to more than one processor and both the number of processors used and the number of microthreads created is parametric and not bound by the resources available on those processors. The create instruction specifies a family of related microthreads, which are created as resources become available and at the same time as previously created microthreads are being executed. All microthreads follow an execution path which ends in the execution of an instruction which terminates that thread, at which point its state is lost and its resources are released.

Microthreads describe parametric concurrency where resources are managed dynamically through the concept of microcontext. Microcontext refers to the private state associated with a microthread. This includes a microthread's program counter and an offset into the register file, which locates its private register variables. The contents and synchronisation state of the registers are also a part of its microcontext. The microcontext is stored in two structures, the local register file and the local scheduler of the processor on which the microthread is executing. Using a 5-bit register specifier, this state is bounded above by 32 register variables and one slot in the scheduler's tables. The microthread and its microcontext are identified uniquely

by its address in the scheduler's tables and this is called its slot number. It should be noted that a family of microthreads will share all memory variables in the scope of a given higher-level context and may also share a number of register variables.

Microthreaded code is not backward compatible. It must be recompiled from the original source code, although this can be legacy, sequential source code. The parallelisation of the source is primarily, but not exclusively, obtained by translating loops into families of microthreads that execute concurrently. Techniques have been used to parallelise both for and while loops, as well as loops with and without loop-carried dependencies. The type of dependency supported is a function of the detailed implementation of the processor and network.

Microthreads created on a processor are queued in its scheduler for execution and a microthread is removed from this queue and passed to the instruction fetch stage of the pipeline on a context switch. The active microthread will continue to execute until either it completes or is itself context switched, because of a blocking read to a register. This may occur on instructions dependent on memory loads or data produced by other microthreads. These are recognised by the compiler and flagged as context-switch points. These instructions may or may not suspend on reading their operands and the explicit context switch merely enables the scheduler to eliminate bubbles in the pipeline in the event that the instruction does block. In this case, the slot number of the suspended thread is stored in the empty register until the data arrives, at which point that thread is rescheduled and added to the scheduler's active queue again. Swapping execution between threads when data is unavailable keeps the processor's utilisation high and hides communication or memory-access latency.

During execution, any data exchange between concurrent microthreads is achieved

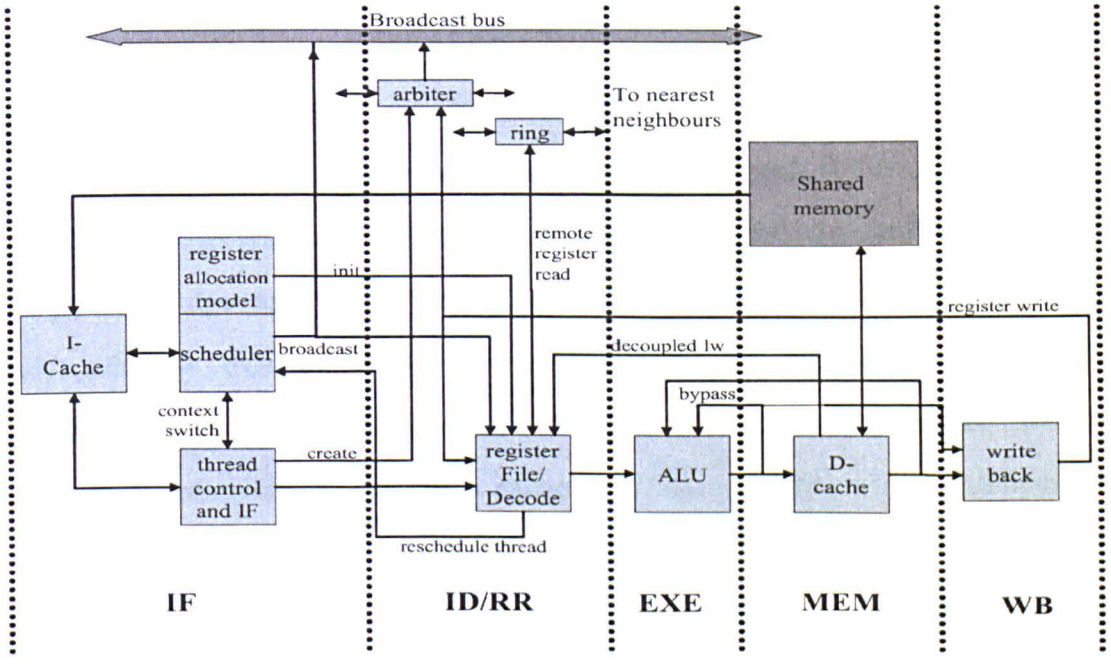


Figure 3.1: Microthreaded microprocessor pipeline.

using register variables. Concurrent microthreads may not communicate using shared memory, as no guarantee can be made about their order of execution. Memory consistency is achieved therefore using bulk synchronisation, either using knowledge of the termination of a dependency chain in a dependent family of microthreads or by the use of a barrier synchronisation in an independent family of microthreads.

3.3 The Microthreaded In-order Pipeline

The microthread model is a generic one, as it can be applied to any ISA, so long as its instructions are executed in-order. In addition, the model can be designed to maintain full backward compatibility, allowing existing binary code to run without speedup [8] on a microthreaded pipeline. Binary compatibility with speedup can also

be obtained using binary-to-binary translation to identify loops and dependencies and adding instructions to support the concurrent execution of those loops and/or the concurrency within the basic blocks.

Figure 3.1 shows a microthreaded, in-order pipeline with its five stages and the communication interfaces required to implement this model in a distributed manner. The pipeline stages are: thread control/instruction fetch, instruction decode/register read/reschedule, execute, memory (if implemented) and write back. Notice that instructions normally complete in order but that in circumstances where the execution time is non-deterministic, such as a cache miss, data is written asynchronously to the register file on a port dedicated to this purpose. In this situation, instruction issue stops in a thread as soon as an instruction attempts to read a register that is empty. Note that all registers have synchronisation bits associated with them defining their state: full, empty, waiting local, waiting remote. No additional pipeline stages are required for instruction issue, retiring instructions, or for routing data between different processors' register files. Short pipelines provide low latency for global operations but a short pipeline can be super-pipelined if required, to increase clock frequency.

Context switching is determined explicitly by the Swch instruction, which can follow/precede any executable instruction and causes control to be transferred to another microthread on the fetching of that instruction. Whether it follows or precedes the instruction it flags is an implementation detail. In this thesis we assume it follows at no loss of generality. In this case a Swch instruction is fetched concurrently with an executing instruction and causes a context switch in the same cycle.

As well as managing data dependencies, context switching is also used to manage

control dependencies in the pipeline, as all transfers of control are also flagged to context switch and only rescheduled when the execution path has been determined. A context switch is also used as a pre-fetching mechanism in the instruction cache. A context switch is forced when the PC increments over a cache-line boundary. This makes a potential cache miss become part of the scheduling process rather than the instruction-execution process. Indeed, it provides a unified mechanism for cache pre-fetching as any thread will not be scheduled for execution unless its current PC is guaranteed to be in the I-cache.

Contexts switches or Kills must be planted by the compiler on all branches of control and on instructions that might stall on reading data. The latter occurs when communicating with other threads, or following a load or long operation. Note that a Swch instruction will always update the value of the PC in the thread's state, and this update occurs after the register-read stage. This is obvious in the case of a branch but not so obvious following a data dependency, where the state of the register will determine whether the instruction will be re-executed or not. If a register reads fails, the instruction reading the register must be re-issued, when the data is available. On the other hand, if the register read succeeds, the next instruction must be executed, which may be the next executable instruction or the one at the branch target location, thus the action at the register read stage determines the value of the thread's PC for all programmed context switches.

Each register in a microthreaded CMP therefore acts as a synchroniser, which can control the issue of instructions from the thread or threads that access it. A reference to the thread's slot number is stored in the register on a synchronisation failure and that thread is rescheduled only when data is written to the register. This mechanism

Table 3.1: Concurrency-control instructions.

Instruction	Instruction Behaviour
Cre	Creates a new family of threads
Swch	Causes a context switch to occur
Kill	Terminates the thread being executed
Bsync	Waits for all other threads to terminate
Brk	Terminates all other threads

is distributed and scalable, requiring only two additional bits per register together with state machines on each of the registers file's ports. This is in stark contrast to an OOO processor's instruction window.

The mechanism of thread suspension and activation provides latency hiding during long or non-deterministic delays when obtaining data. The maximum latency tolerated is related to the size of the scheduler queue (called the continuation queue - CQ, throughout the thesis) or the size of the register file, which can both restrict the number of local threads active at any time. Of course, the latency is also related to the average number of statically scheduled instructions between context switches. Only if a processor has no active threads, will the pipeline stall on attempting to read an empty register.

This means of scheduling instructions is similar in complexity to that of a conventional, single-issue, in-order processor. The only additional overhead is the larger than normal register file, the maintenance of the CQ and the RAU, which are investigated in detail in this thesis. However, as they are both scalable with local concurrency they can both be tuned in size at design time in order to provide a given amount of latency tolerance.

3.4 Concurrency Controls

Table 3.1 shows the five instructions required to support this model on an existing ISA. The model provides concurrency-control instructions to create families of threads (Cre), to explicitly context switch between threads (Swch) and to kill a thread (Kill). Two global synchronisation instructions are also provided, one is a barrier synchronisation (Bsync), the other is a form of a break instruction (Brk), which forces a break from a loop executed concurrently. Note that all of these instructions can be completed in the first stage of a pipeline as they only control the action of the scheduler. Because of this, these additional instructions do not require a pipeline cycle so long as they are fetched concurrently with executable instructions. This allows concurrency controls in the model to be very efficiently implemented. Each instruction will now be described in more detail.

3.4.1 Thread Creation

The microthreaded model defines explicit and parametric concurrency using the *Cre* instruction. This instruction broadcasts a pointer to the TCB to all processors assigned to the current context; see [24] for details of dynamic processor allocation. The TCB contains parameters that define a family of threads, e.g. thread pointer and the start and limit of the loop. It also defines the dynamic resources required by each thread (its microcontext) in terms of local, global and shared registers. For loops which carry a dependency, the dependency distance between loop iterations is assumed to be constant i.e. $= 1$. A family of threads can be created without requiring a pipeline slot, as the create instruction is executed concurrently with a regular instruction in the Instruction fetch *(IF)* stage of the pipeline. The TCB for our current

Table 3.2: Thread control block containing parameters that describe a family of microthreads.

Name	Description Behaviour	Size (Byte)
Main Pointer	One pointer per thread for main loop-body code	4
Start	Start of loop index value	4
Limit	Limit of loop index value	4
Locals	Number of local registers dynamically allocated/thread	1
Globals	Number of global registers dynamically allocated/thread	1
Shareds	Number of shared registers dynamically allocated/thread	1

work on implementation overheads is defined in table 3.2.

The concurrency described by this instruction is therefore parametric and may exceed the resources available in terms of registers and thread slots in the CQ. The RAU in each local scheduler maintains the allocation state of all registers in each register file and this controls the creation of threads at a rate of one per pipeline cycle. Once allocated to a processor a thread runs to completion, i.e. until it encounters a Kill instruction and then terminates. A terminated thread releases it resources so long as any dependent thread has also terminated. To do so before this may destroy data that has not yet been read by the dependent thread. Note that microthreads are usually (but not exclusively) very short sequences of instructions without internal loops.

3.4.2 Context-Switching

The microthreaded context switching mechanism is achieved using the *Swch* instruction, which is acted upon in the IF stage of the pipeline, giving cycle-by-cycle interleaving if necessary. When a *Swch* instruction is executed, the IF stage reads the next instruction from another ready thread, whose state is passed to the IF stage as a

result of the context switch. As this action only requires the IF stage of the pipeline, it can be performed concurrently with an instruction from the base ISA, so long as the *Swch* instruction is prefetched with it.

The context switching mechanism is used to manage both control and data dependencies. It is used to eliminate control dependencies by context switching following every transfer of control, in order to keep the pipeline full without any branch prediction. This has the advantage that no instruction is executed speculatively and consequently, power is neither dissipated in making a prediction nor in executing instructions on the wrong dynamic path. Context switching also eliminates bubbles in the pipeline on data dependencies that have non-deterministic timing, such as loads from memory or thread-to-thread communication. Context switching provides an arbitrary large tolerance to latency, determined by the size of the local register file.

3.4.3 Thread Synchronisation

The only synchronising memory in the microthreaded model is provided by the registers and this gives an efficient and scalable mechanism for synchronising data dependencies. The synchronisation is performed using two synchronisation bits associate with every register, which differentiate between the following states: *full*, *empty*, *waiting-local* and *waiting-remote*.

Registers are allocated to microcontexts in the *empty* state and a read to an empty register will fail, resulting in a reference to the microthread that issued the instruction being stored in that register. This reference passes down the pipeline with each instruction executed. Using the CQ in the scheduler, lists of continuations may be suspended on a register, which is required when multiple threads are dependent on

the value to be stored there. All registers therefore implement i-structures [105] in a microthreaded microprocessor. In the *full* state, registers operate normally, providing data upon a register read and, if no synchronisation is required, a register can be repeatedly written to without changing its synchronisation state to provide backward compatibility. The compiler can easily recognise the potential for a synchronisation failure if a schedule for the dependency is not known at compile time. If so, it inserts a context switch on the dependent instruction. Examples include instructions dependent on a prior load word, produced in another thread, or produced in iterative CPU operations.

The register is set to one of the waiting states when it holds a continuation. Two kinds of continuation are distinguished *waiting-local*, when the register holds the head of a list of continuations to local microthreads and *waiting-remote*, when the register holds a remote request for data from another processor. The latter enables the microcontext for one iteration to be stored for read-only access on a remote processor when managing loop-carried dependencies. This implements a scalable and distributed shared-register model between processors without using a single, multi-ported register file, which is known to be unscalable.

The use of dataflow synchronisation between threads enables a policy of conservative instruction execution to be applied. When no microthreads are active because all are waiting external events, such as load word requests, the pipeline will stall and, if the pipeline is flushed completely, the scheduler will stop clocks and power down the processor going into a standby mode, in which it consumes minimal power. This is a major advantage of data-driven models. Conservative instruction execution policies conserve power in contrast to the eager policies used in OOO issue pipelines, which

have no mechanisms to recognise such a conjunction of schedules. This will have a major impact on power conservation and efficiency.

Context switching and successful synchronisation have no overhead in terms of additional pipeline cycles. The context switch interleaves threads in the first stage of the pipeline, if necessary on a cycle- by-cycle basis. Synchronisation occurs at register-read stage and only if it fails will any exceptional action be triggered. On a synchronisation failure, control for the instruction is mutated to store a reference to the microthread in the register being read. This means that the only overhead in supporting these explicit concurrency controls in is the additional cycle required to reissue the failed instruction when the suspended thread is reactivated by the arrival of the data. Of course there are overheads in hardware but this is true for any model.

The model also provides a barrier synchronisation *(Bsync)* instruction, which suspends the issuing thread until all other threads have completed and a *Brk* instruction, which explicitly kills all other threads leaving only the main thread. These instructions are required to provide bulk synchronisation for memory consistency. There is no synchronisation on main memory, only the registers are synchronising. This means that two different microthreads in the same family may not read after write to the same location in memory because the ordering of those operations can not be guaranteed. It also means that any loop-carried dependencies must be compiled to use register variables. A partitioning of the microcontext supports this mechanism efficiently.

3.4.4 Thread Termination

Thread termination in the microthreaded model is achieved through a *Kill* instruction, which of course causes a context switch as well as updating the microthread's state to killed. The resources of the killed threads are released at this stage, unless there is another thread dependent upon it, in which case its resources will not be released until the dependent thread has also been killed. (Note that this is the most conservative policy and more efficient policies may be implemented that detect when all loop-carried dependencies have been satisfied).

3.5 Scalable Instruction Issue

Current microprocessors attempt to extract high levels of ILP by issuing independent instructions out of sequence. They do this most successfully by predicting loop branches and unrolling multiple iterations of a loop within the instruction window. The problem with this approach has already been described; a large instruction window is required in order to find sufficient independent instructions and the logic associated with it grows at least with the square of the issue width.

If we compare this with what is happening in the microthreaded model, we see that almost exactly the same mechanism is being used to extract ILP, with one major difference, a microthreaded microprocessor execute fragments of the sequential programs OOO. These fragments (the microthreads) are identified at compile time from loop bodies and conventional ILP and may execute in any order subject only to dataflow constraints. Instructions within fragments however, issue and complete in-order. We have already seen that a context switch suspends a fragment at instructions

whose operands have non-deterministic timing. The dependent instruction is issued and stores a pointer to its fragment if a register operand is found to be empty. Any suspended fragments are rescheduled when data is written to the waiting register. Thus only instructions up to the first dependency in each fragment (loop body) are issued and only that instruction will be waiting for the dependency to be resolved, all subsequent instructions in that pipeline will come from other fragments. In an OOO issue model the instruction window is filled with all instructions from each loop unrolled by branch prediction because it knows nothing prior about the instruction schedules.

Consider a computation that only ever contains one independent instruction per loop of l instructions, then to get n-way issue n loops must be unrolled and the instruction window will contain $n*l$ instructions for each n instructions issued. In comparison, the microthreaded model would issue the first n independent instructions from n threads (iterations), then it would issue the first dependent instructions from the same n threads before context switching. The next n instructions would then come from the next n iterations (threads). Synchronisation, instead of taking place in a global structure with $O(n^2)$ complexity, is distributed to n registers and has linear complexity. Each thread waits for the dependency to be resolved before being able to issue any new instructions. In effect the instruction window in a microthreaded model is distributed to the whole of the architectural register set and only one link in the dependency graph for each fragment of code is ever exposed simultaneously. Moreover, no speculation is ever required and consequently, if the schedules are such that all processors would become inactive, then this state can be recognised and used to power-down the processors to conserve energy.

Compare this to the execution in an OOO processor, where instructions are executed speculatively regardless of whether they are on the correct execution path. Although predictions are generally accurate in determining the execution path in loops, if the code within a loop contains unpredictable, data-dependent branches, this can result in a lot of energy being consumed for no useful work. Researchers now talk about "breaking the dependency barrier" [104] using data in addition to control speculation, but what does this mean? Indices can be predicted readily but these are not true dependencies and do not constrain the microthreaded model. Addresses, based on those indices can also be predicted with a reasonable amount of accuracy but again these do not constrain the microthreaded model. This leaves true computational data dependencies, which can only be predicted under very extraordinary circumstances. It seems therefore that there is no justification for consuming power in attempting data speculation.

OOO issue has no global knowledge of concurrency or synchronisation. Microthreading, on the other hand, is able to execute conservatively as it does have that global knowledge. Real dependencies are flagged by context switching, concurrency is exposed by dynamically executing parametric *Cre* instructions and the name-space for synchronisation spans an entire loop as registers are allocated dynamically. At any instant the physical namespace is determined by the registers that have been allocated to threads.

3.6 Thread State

When a thread is assigned resources by the scheduler, it is initially set to the *waiting* state in the local scheduler queue, as it must wait for its code to be loaded into the I-cache before it can be considered active. A thread will go into a *suspended* state when it has been context switched until either the register synchronisation has completed or the branch target has been defined, when it again goes into the waiting state. The scheduler generates a request to the I-cache to pre-fetch the required code for any thread that enters the waiting state. If the required code is available, then the I-cache acknowledges the scheduler immediately, otherwise not until the required code in the cache. The thread's state becomes *ready* at this stage. A *killed* state is also required to indicate those threads that have completed but whose data may still be in use. At any time there is just one thread per processor, which is in the *running* state, on start up this will be the main thread.

On a context switch or kill, the instruction fetch stage is provided with the state of a new thread if any are active, otherwise the pipeline stalls for a few cycles to resolve the synchronisation and if it fails, the pipeline simply stops. This action is simple, requires no additional flush or clean-up logic and most importantly, is conservative in its use of power. Note that by definition, when no local threads are active, the synchronisation event has to be asynchronous and hence does not require any local clocks.

The state of a thread also includes its program counter *(PC)*, the base address of its microcontext and the base address and location of any microcontexts that it is dependent upon. The state also includes an implicit slot number, which is the address of the entry in the CQ and which uniquely identifies the thread on a given processor.

The last field required is a link field, which holds a slot number for building linked lists of threads to identify empty slots, ready queues and an arbitrary number of CQs that support multiple continuations on different registers. The slot reference is used as a tag to the I-cache and is also passed through the pipeline and stored in the relevant operand register if a register read fails, where it forms the head of that CQ. Chapters 5 and 7 discuss the implementation of the CQ with its required connections in more detail.

3.7 Register File Partitioning and Distribution

We have already seen that Rixner et. al. [22] have shown that a distributed register file architecture achieved a better performance compared with a global solution and it also provides superior scaling properties. Their work was based on streaming applications, where register sources and destinations are compiled statically. We will show that such a distributed organisation can also be based on extensions to a general-purpose ISA with dynamic scheduling. The concept of a dynamic microcontext associated with parallelising different iterations has already been introduced and is required in order to manage communications between microcontexts in a scalable manner. It is necessary for the compiler to partition the microcontext into different windows representing different types of communication and for the hardware to recognise these windows to emulate a shared register multiprocessor using distributed register files and a communication network.

A microthreaded compiler must recognise and identify four different types of communication patterns. There are a number of ways in which this partitioning can be encoded and here, we describe a simple and efficient scheme that supports a fully

distributed register file based on a conventional RISC ISA, assuming a 5-bit register specifier and hence a 32-register address space per microcontext (although, not the same 32 registers for each thread).

The first register window is the global window (represented by $Gi). These registers are used to store loop invariants or any other data that is shared by all threads. In other models of concurrency these would represent broadcast data, which is written by one and read by many processes. Their access patterns have the characteristics that they are written to infrequently but read from frequently. The address space in a conventional RISC ISA is partitioned so that the lower 16 registers form this global window. These are statically allocated for a given context and every thread can read and/or write to them. Note that the main thread has 32 statically allocated registers, 16 of which are visible to all microthreads as globals and 16 of which are visible only to the main thread. Each thread sees 32 registers. The lower 16 of these are the globals and these are shared by all threads and the upper half are local to a given thread.

The upper 16 registers are used to address the microcontext of each iteration in a family of threads. As each iteration shares common code, the address of each microcontext in the register file must be unique to that iteration. As we have seen, the base address of a thread's micro-context forms a part of its state. This immediately gives a means of implementing a distributed, shared-register model. We need to know the processor on which a thread is running and the base address of its microcontext in order to share its data. However, we can further partition the microcontext into a local and shared part to avoid too much additional complexity in implementing the pipeline.

Three register windows are mapped to the upper or dynamic half of the address space for each microcontext. These are the local window ($Li), the shared window ($Si) and the dependent window ($Di). Thus the sum of the size of these three windows must be less than or equal to 16. The local window stores values that are local to a given thread. For example they store values from indexed arrays used only in a single iteration. Reads and writes to the local window are all local to the processor that a thread is running on and no distribution of the L window is therefore required. The S and D register windows provide the means of sharing a part of a microcontext between threads. The S-window is written by one thread and is read by another thread using its D-window.

It should be noted that many different models can be supported by this basic mechanism. In this thesis, a simple model is described but different models of communication with different constraints and solutions to resource deadlock can be implemented. The mechanism would even support a hierarchy of microcontexts by allowing an iteration in one family of threads to create a subordinate family where the dynamic part of the address space in the creating family became the static part in the subordinate family. This would support nested multi-dimensional loops as well as breadth first recursion. There are difficulties however, in resolving resource deadlock problems in all but the simplest models and this requires further research to resolve.

In this thesis we describe a simple model that supports a single level of loop with communication between iterations being allowed only between iterations that differ by a create-time constant. An example of this type of communication can be found in loop-carried dependencies, where one iteration produces a value, which is used by another iteration. For example, A[i] := ... A[i-k]... where k is an invariant of the loop.

Such dependencies normally act as a deterrent to loop vectorisation or parallelisation but this is not so in this model, as the independent instructions in each loop can execute concurrently. This is the same ILP as is extracted from an OOO model.

Consider now the implementation of this basic model. It is straightforward to distribute the global register window and its charactistics suggest a broadcast bus as being an appropriate implementation. This requires that all processors executing a family of microthreads be defined prior to any loop invariants being written (or re-written) to the global window. The hardware then traps any writes to the global window and replicates the values using the broadcast bus to the corresponding location in all processors' global windows. As multiple threads may read the values written to the global register window, registers must support arbitrarily large CQs, bounded above only by the number of threads that can be active at any time on one processor.

The write to the global window can be from any processor and thus can be used to return a value from an interaction to the global state of a context. The write is also asynchronous and independent of pipeline operation, provided there is local buffering for the data in the event of a conflict on the bus. Contention for this bus should not occur regularly, as writes to globals are generally much less frequent than reads (by a factor proportional to the concurrency of the code). This issue is analysed and evaluated in the next chapter.

The distribution of S and D-windows is a little more complex than the global window. Normally, a producer thread writes to its S-window and the consumer reads from its D-window, which maps in some sense onto the S-window of the producer; we will return to this. However, there is no restriction on a thread reading a register

from its S-window so long as data has already been written to it (it would deadlock otherwise). There is also no physical restriction on multiple writes to the S-window, although this may introduce non-determinism if a synchronisation is pending on it. As far as the hardware is concerned therefore, the S-window is identical to the L-window, as all reads and writes to it are local and are mapped to the dynamic half of the register-address space. On the other hand, a thread will never write to its D-window, which is strictly read-only. The hardware need only recognise reads to the D-window in order to implement sharing between two different threads. In order to perform a read from a D-window, a processor needs the location (processor id) and base address of the S-window of the producer thread. There are two cases to consider in supporting the distribution of register files in the base-level model we have described.

The first and easiest case is when the consumer iteration is scheduled to the same processor as the producer. In this case a read to the D-window can be implemented as a normal pipeline read by mapping the D-window of the consumer microcontext onto the S-window of the producer microcontext. The thread's state must therefore contain the base address of its own microcontext for local reads and also the base address of any microcontext it is dependent upon. In the base-level model we present, only one other microcontext is accessed, at a constant offset in the index space.

In the second case, the producer and consumer iterations are scheduled to different processors. Now, the consumer's read to the D-window will generate a remote request to the processor on which the producer iteration is running. Whereas in the first case a microcontext's D-window is not physically allocated, in this second case it must be. It is used to cache a local copy of the remote microcontext's S-window. It is also used

to store the thread continuation locally. The communication is again asynchronous and independent of the pipeline operation. The consumer thread is suspended on its read to the D-window location until the data arrives from the remote processor. For this constant strided communication, iteration schedules exist that require only nearest neighbour communication in a ring network to implement the distributed shared-register scheme. Note that a request from the consumer thread may find an empty register, in which case the request gets suspended in the producer's S-window until the required data has been produced. Thus a shared-register transaction may involve two continuations, a thread suspended in the D-window of the consumer *(waiting-local)* and a remote request suspended in the S-window of the producer *(waiting-remote)*. As these states are mutually exclusive, the compiler must ensure that the producer thread does not suspend on one of its own S-window locations. This can happen if a load from memory to an S location is also used in the local thread. However, as dependencies are passed via register variables, this can only happen in the initialisation of a dependency chain. This case can be avoided by loading to a location in the L-window when the value is required locally and then copying it to the S-window with a deterministic schedule.

The additional complexity required in this distributed register file implementation is two bits in each register to encode the four synchronisation states: full, empty, waiting-local, waiting-remote; a small amount of additional logic to address the dynamically allocated registers using base-displacement addressing; and a simple state machine on each register port to implement the required action based on the synchronisation state.

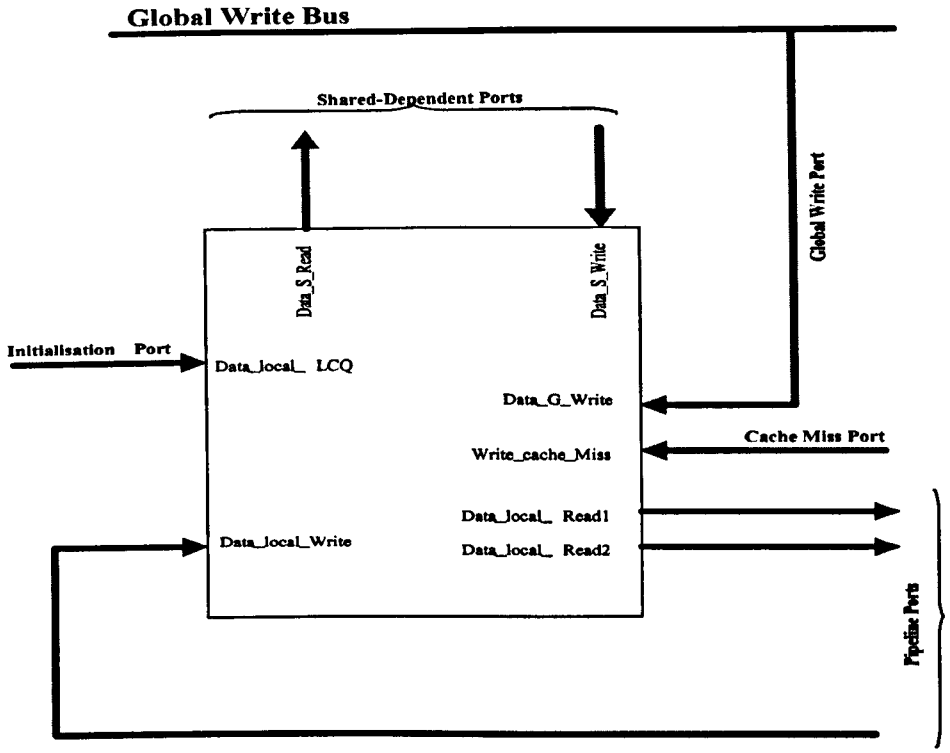


Figure 3.2: Microthreaded register-file ports.

A method now has been described to distribute all classes of communication required in the base-level model. However, we must ensure that this distribution does not require us to implement register files locally that are not scalable. This requires the number of local ports in the register file to be constant. Accesses to L, S and local D-windows requires at most two read and one write port for a single-issue pipeline. The G-window requires an additional write port independent of the pipeline ports. Finally reads to a remote D-window require one read port and one write port per processor. Contention for this port will depend on the pattern of dependencies, which for the model described is regular and hence evenly distributed with appropriate scheduling. Each iteration is allocated a separate microcontext in the dynamic half of the

register-address space and the first local register ($L0) is initialised by the scheduler to the loop index for that iteration, so this also requires a write port. Finally, a write port is required to support decoupled access to the memory on a cache miss in order to avoid a bubble in the pipeline, when data becomes available. The next chapter provides a full detail analysis to these ports in term of frequency accesses to each port.

Figure 3.2 shows a block diagram of the microthreaded register file illustrating its required ports. As shown, it has a maximum of 8 local ports per processor. The register file could be implemented with just three ports by stalling the pipeline whenever one of the asynchronous reads or writes occurs but this would degrade its performance significantly. In fact, the requirements of the register file in terms of exact number logical ports is still not clear. Therefore, an analysis to the requirements of the microthreaded register file in terms of number of read/write ports is required. The next chapter provides a detail analysis and evaluation to this problem.

3.8 Registers Allocation Unit

To create a namespace that includes all iterations of the loop and to create a binding between variables in two iterations in the case of loop-carried dependencies, Jesshope [25] proposed a dynamic allocation of registers to thread before thread scheduling.

As stated previously, the TCB defines information about a family of threads with the loop's parameters. It also defines the dynamic resources required by each thread in terms of local, global and shared registers. Local registers are allocated from the processor's local register file, while global registers can be accessed by all threads.

Shared registers can be accessed by two threads, one thread to produce data and another to consume it.

The RAU in each local scheduler maintains the allocation state of all registers in each register file. When a thread is killed a release signal is returned to the RAU to free that thread's resources. Indeed, dynamically allocating registers prior to thread scheduling and releasing them when the thread terminates provides an efficient and effective utilisation of registers. Also, it is important to note that, the process of dynamically allocating registers to microthreads is fully decoupled from the pipeline execution, allowing the pipeline to work at full utilisation without any extra pipeline stages for allocating registers.

However, the mechanism of dynamically allocating registers still requires an efficient hardware scheme to implement this process. The next chapter provides the solution and discusses the design and implementation of an efficient scheme for doing this.

3.9 Cache Prefetching and Data Locality

As the gap speed between processors and the memories becomes very large, where processor speeds are increasing 60% a year compared with memory speeds at only 7% a year [106], techniques such as *cache locality optimisations* and *cache line prefetching* are become increasingly important. For example, instruction cache misses stall pipelined processors for many cycles and are a major source of performance degradation in modern processors. To overcome this limitation, instruction prefetching schemes can be used to minimise instruction cache miss latency. Also, cache locality optimisations use compiler or run-time transformations to change the computation

order and/or data layout of a program to increase the probability of accessing data already in the cache [107].

Several papers have already investigated instruction cache prefetching either by compiler driven or hardware schemes [108, 109, 110, 111]. Compiler driven schemes have been used in some recent commercial machines such as [108, 109], where the compiler's analysis of the program code and provides a hint to the hardware through a prefetching instructions. Thus, when the explicit prefetch instructions are executed, the data is loaded from memory to cache.

Generally, hardware prefetching schemes can be classified into two main categories; *sequential* and *non-sequential* instruction cache prefetching schemes. In sequential prefetching schemes like [110], a simple mechanism is used, by prefetching the next cache line when a cache line fetched (next-line always). Other previous work [111] has also described sequential prefetching, which uses a next N-line prefetching scheme to prefetch the next N sequential lines following the line currently being fetched by the processor. However, increasing the value of N, results in increasing the prefetching distance, which causes increased pollution of the cache with useless prefetch [112, 113]. The pollution occurs from the useless speculative memory references by moving out the correct memory block, while this block may be used by correct-path execution.

Two main styles have been used in a non-sequential hardware prefetching schemes, *history-based* and *execution-based.* An example of history-based is found in [110], where the authors proposed a target-line prefetching scheme, which uses a history prediction table to maintain information about the address of the cache lines most recently fetched by the processor. Thus, if the target line address is in the table, then that line is candidate for prefetching. While, if there is a miss in the prefetch

table, there is no prefetch request. Execution-based [114, 115] schemes uses a branch predictor to prefetch the cache lines. For example work done by [115], proposed a fetch directed prefetching scheme. This architecture uses a branch predictor and an instruction cache, so the branch predictor can run ahead of the instruction cache fetch.

Recently, Spracklen et. al. [113] analysed and summarised the problems in existing sequential and non-sequential instruction prefetcher hardware schemes. They also showed how the aggressive instruction prefetching in CMPs can cause pollution in the shared L2 cache and increase the L2 cache miss rate. Generally, existing instruction cache prefetching schemes attempt to reduce cache miss rate rather than eliminate this limitation. Also, these schemes still employ heuristic prediction, which may result in extra penalties and insufficient use of the prefetching. Generally, the aggressive speculation and prefetching techniques used in modern processors cause speculative memory references, which result in loading the data into the caches that are not needed by correct-path execution [116].

The microthreaded microprocessor model supports a pre-fetching and replacement mechanism that avoids any instruction cache misses [17]. The mechanism is deterministic and very simple, where each line in the I-cache requires a counter of the number of active threads that are using that cache line. As soon as the thread's resources are allocated after being created, the scheduler generates a request through the thread pointer associated with its slot number to the I-cache to pre-fetch the required code for that thread. If the code is available, then the I-cache acknowledges the local scheduler immediately, the requested cache line counter is incremented and the thread's state becomes active.

While, if the thread pointer misses the required code in the I-cache, then the required memory block fetches into any line with a count of zero. Until this happens, the thread remains in a suspended state. It is important to note that the thread is not made active until the I-cache block along the new path of execution has been fetched. Also, when a thread is rescheduled after being suspended, the same process is followed. Finally, when a thread is killed, its resources are released and the I-cache line counter is decremented. Therefore, there is no need to insert special instructions to perform prefetching, only the flexible and efficient thread scheduling mechanism provided by the microthreaded model detects and predicts what thread will be submitted to the execution in the future. This is hidden from the compiler/programmer. It is important to note that this mechanism is also fully decoupled from the pipeline execution.

3.10 Summary

This chapter reviewed the microthreaded microprocessor approach and discussed its features that make it a very promising solution for scalable CMP with large numbers of processors. The approach allows concurrency to be extracted from sequential code, exploiting different types of parallelism. ILP and LLP are both detected by the recompilation of legacy, sequential source code or indeed, could be obtained from the translation of existing legacy binary code. The approach also supports TLP by assigning application threads to groups of processors in the CMP. The concurrency controls provided by this approach not only provides a considerable level of concurrency, but also optimises the scheduling process and supports scalability. Also, the

approach supports a pre-fetching and replacement mechanism that avoids any instruction cache misses. This mechanism is fully decoupled from the processor pipeline and avoids any stalls during instruction misses.

The distributed configuration of instruction issue and a fully scalable register file, which implements a distributed, shared-register model of communication and synchronisation between multiple processors on a single chip, are two distinct features in this model. However, it is not yet clear what the requirements of the microthreaded register file are in terms of number of read and write ports to keep it compact and scalable. The next chapter provides an evaluation and analysis for this issue in more detail.

The disadvantage of the microthreaded approach is that registers must be allocated dynamically and state, in addition to its PC, must be maintained for each microthread. To allocate registers dynamically requires additional logic and with many concurrent threads, any additional thread state can lead to significant storage in the scheduler. In the next chapter, we proposed a novel design and implementation of a hardware support for dynamically allocating and de-allocating registers for microthreaded CMP.

Chapter 4

Microthreaded Distributed Register File

4.1 Chapter Overview

In the previous chapter, it was shown that the requirement in terms of the number of logical read and write ports for a microthreaded register file is not clear. Also, because the model supports dynamic register allocation, an efficient hardware scheme is required to handle registers allocation. This chapter provides a solution to both problems with analysis, implementation and evaluation.

The outline of this chapter is as follows. The next section summarises selected work on modern register files. In section 4.3, a method of sharing registers in the microthreaded model is presented. An analysis and evaluation of the requirements of the microthreaded register file in terms of the frequency of accesses to each logical port is given in section 4.4. Section 4.5 discusses the centralised and distribution organisation for the RAU. The section also compares an alternative implementation

for register allocation. The design and implementation of a scalable allocation scheme for dynamically allocating and de-allocating registers for the microthreaded CMP is presented in section 4.6. A summary of the chapter is provided in section 4.7.

4.2 Modern Register Files

All systems that implement concurrency require some form of synchronisation memory. In dataflow architecture, this is the matching store, in an OOO issue processor it is the register file, supported by the instruction window, reservation stations and reorder buffer. To implement more concurrency and higher levels of latency tolerance, this synchronising memory must be increased in size. This would not be a problem except that in centralised architectures, as issue width increases, the number of ports to this synchronising memory must also increase. The problem is that the register cell size grows quadratically with the number of ports or issue width. As mentioned previously, if N instructions can be issued in one cycle, then a central register file requires $2N$ read ports and N write ports to handle the worst case scenario. This means that the register cell size grows quadratically with N. Moreover, as the number of registers also increases with the issue width, a typical scaling of register file area is as the cube of N.

Several projects have investigated the register file problem, in terms of reducing the number of registers, or minimising the number of read or write ports [60, 22, 117, 118]. As described in chapter 2, the register file in the proposed Alpha 8-way issue 21464 occupied an area of some 5 times the size of the L1 D-cache of 64KB. Also, in the Motorola's M. CORE architecture, the register file energy consumption can be 16% of the total processor's power and 42% of the data path power [67]. It is clear therefore

that multi-ported register files in modern microprocessors consume significant power and die area.

Work done in [117] describes a bypass scheme to reduce the number of register file read ports by avoiding unnecessary register file reads for the cases where values are bypassed. In this scheme an extra bypass hint bit is added to each operand of instructions waiting in the issue window and a wake-up mechanism is issued to reduce register file read ports. As described in [119], this technique has two main problems. First, the scheme is only a prediction, which can be incorrect, requiring several additional repair cycles for recovery on miss-prediction. Secondly, because the bypass hint is not reset on every cycle, the hint is optimistic and can be incorrect if the source instruction has written back to register file before the dependent instruction is issued. Furthermore, an extra pipeline stage is required to determine whether to read data operands from the bypass network or from the register file.

Other approaches include a delayed write back scheme [118], where a memory structure is used to delay the write-back results for a few cycles to reduce register file ports. The disadvantage of this scheme is that it is necessary to write the results both to the register file and the write-back queue concurrently to avoid consistency problems during register renaming. The authors propose an extension to this scheme to reduce the number of register write ports. However, this extension suffers from an IPC penalty and it degrades the pipeline performance. Furthermore, in this model, any branch miss-predictions cause a pipeline stall and insufficient use of the delay write back queue. In fact most previous schemes for minimising the multi-ported register file have required changes in the pipeline design and do not enable full scalability. At best they provide a constant remission in the scalability of the register

file.

Rixner et. al. [22] suggested several partitioning schemes for the register file from the perspective of streaming applications, including designs spanning a central register file through to a distributed register file organisation. Their results, not surprisingly, show that a centralised register file is costly and scales as $O(N^3)$, while in the distributed scheme, each ALU has its own port to connect to the local register files and another port to access other register files via a fast crossbar switch network. This partioning proved to use less area, power and delay compared with the purely global scheme and was also shown to provide a scalable solution. The distributed configuration also has a smaller access time compared with the centralised organisation. Bunchua et. al. [120] also compared the register file access time for central and local register files configuration. In this work, a 128 32-bit register file with 16 read ports and 8 write ports is used as a central register file and is compared to a local register file with 32 32-bit registers, 2 read ports, 1 write port, and 1 read/write port. The result from their cache access and cycle time model *(CACTI)* showed a 47.8% reduction in access time for the distributed register file organisation across all technologies.

It is not clear from this work, whether the programming model for the distributed register file model is sufficiently general for most computations. With a distributed register file and explicitly routed network, operations must be scheduled by the compiler and routing information must also be generated with code for each processor in order to route results from one processor's register file to another. Although it may be possible to program streaming applications using such a model, in general, concurrency and scheduling can not be defined statically.

Other previous work has described a distributed register file configuration [120] where a fully distributed register file organisation is used in a superscalar processor. The architecture exploits a local register mapping table and a dedicated register transfer network to implement this configuration. This architecture requires an extra hardware recopy unit to handle the register file dispatch operations. Also, this architecture suffers from a delay penalty as the execution unit of an instruction that requires a value from a remote register file must stall until it is available. The authors have proposed an eager transfer mechanism to reduce this penalty but this still suffers from an IPC penalty and requires both central issue logic and global renaming.

In our research, it seems that only the microthreaded model provides sufficient information to implement a penalty free distributed register file organisation. Such a proposal is given in [8] where each processor in a CMP has its own register file in a shared register model. Accesses to remote data is described in the binary code and does not require speculative execution or routing. The decoupling is provided by a synchronisation mechanism on registers and the routing is decoupled from the operation of the microthreaded pipeline operation, exploiting the same latency tolerance mechanisms as used for main memory access.

4.3 Analysis and Evaluation of Microthreaded Register File Ports

It is shown in the previous chapter that microthreaded register file uses a 32-register address space per microcontext. Half of these addresses are shared by all threads by replicating writes to all processors using the broadcast bus. This can be considered the

top-level context. The remaining 16 addresses refer to unique locations in the register file for each value of the loop index and represent the different microcontexts. In some ways this is similar to register windowing in the scalable processor architecture (SPARC) architecture [121]. However, in a microthreaded processor multiple base addresses are maintained in the CQ for each active thread.

In this section, an analysis of microthreaded register file ports (see figure 3.2 for register-file ports analysed) is made in terms of the average number of accesses to each port of the register file in every pipeline cycle. This analysis is based on hand compilation of a variety of loop kernels (see appendix A for loop kernels analysed). The loops considered included a number of livermore kernels, some that are independent and some that contain loop-carried dependencies. It also includes both affine and non-affine loops, vector and matrix problems, and a recursive doubling algorithm. We have used loop kernels at this stage as we currently have no compiler to compile complete benchmarks. However, as the model only gains speedup via loops, we have chosen a broad set of representative loops from scientific and other applications. Analysis of complete programs and other standard benchmarks will be undertaken when a compiler we are developing is able to generate microthreaded code.

The results are based on a static analysis of the accesses to various register windows and investigate the average traffic on the microthreaded register file ports. The five types of register file ports are shown in figure 3.2 and include, pipeline ports (read-R and write-W), the initialisation port (I), the shared-dependent ports (S_d), the broadcast port (B_r) and the write port that is required in the case of a cache miss (W_m). The goal of this analysis is to guide the implementation parameters of such a system. We aim to show that all accesses other than the synchronous pipeline ports

Table 4.1: Average number of accesses to each class of register file port over a range of loop kernels, m= problem size.

Loop	Ne	R	W	I	Br	Sd
A: Partial Products	3m	$\frac{4m-3}{Ne}$	$\frac{2m-1}{Ne}$	0.3333	0	$\frac{m-1}{M*Ne}$
B:2-D SOR	5m -2	$\frac{8m-15}{Ne}$	$\frac{4m-4}{Ne}$	0.2	0	$\frac{m-2}{M*Ne}$
L3 :Inner Product	4m +4	$\frac{5m+3}{Ne}$	$\frac{4m+1}{Ne}$	0.25	0	$\frac{m}{M*Ne}$
L4 :Banded Linear Equation	3m +34	$\frac{2.4m+37.4}{Ne}$	$\frac{3m+22}{Ne}$	0.2	$\frac{4n}{Ne}$	$\frac{1.8m+1.8}{M*Ne}$
L5 :Tri -Diagonal Elimination	5m +3	$\frac{7m}{Ne}$	$\frac{4m}{Ne}$	0.25	0	$\frac{m-1}{M*Ne}$
L6 :General Linear Recurrence	$2.5m+6.5m^{1/2}-5$	$\frac{5.5m+2.5m^{1/2}-5}{Ne}$	$\frac{3m+m^{1/2}2}{Ne}$	0.1429	$\frac{(m^{1/2}-1)\ n}{Ne}$	$\frac{0.5m-0.5m^{1/2}}{M*Ne}$
C :Pointer Chasing	14m +5	$\frac{9m+3}{Ne}$	$\frac{6m+2}{Ne}$	0.0714	$\frac{n}{Ne}$	$\frac{m}{M*Ne}$
L1 :Hydro Fragment	9m +5	$\frac{15m}{Ne}$	$\frac{8m+3}{Ne}$	0.1111	$\frac{3n}{Ne}$	0
L2 :ICCG	11m +2logm -21	$\frac{17m-5logm-27}{Ne}$	$\frac{10m-5logm-12}{Ne}$	0.0909	$\frac{(logm-1)\ n}{Ne}$	0
L7 :Equation of State Fragment	26m +5	$\frac{43m+3}{Ne}$	$\frac{25m+3}{Ne}$	0.0385	$\frac{3n}{Ne}$	0

can be implemented by a pair of read and write ports, with arbitration between the different sources. In this case a register file with five fixed ports would be sufficient for each of the processors in our CMP design.

The microthreaded pipeline uses three synchronous ports. These ports are used to access three classes of register windows i.e. the \$L, \$S and \$G register windows. If we assume that the average number of reads to the pipeline ports in each cycle is R and the average number of writes to the pipeline port in each cycle is W, then these values are defined by the following equations, where N_e is the total number of instructions executed.

$$R = \frac{\sum_{inst.} (Read(\$L) + Read(\$S) + Read(\$G))}{N_e} \tag{4.3.1}$$

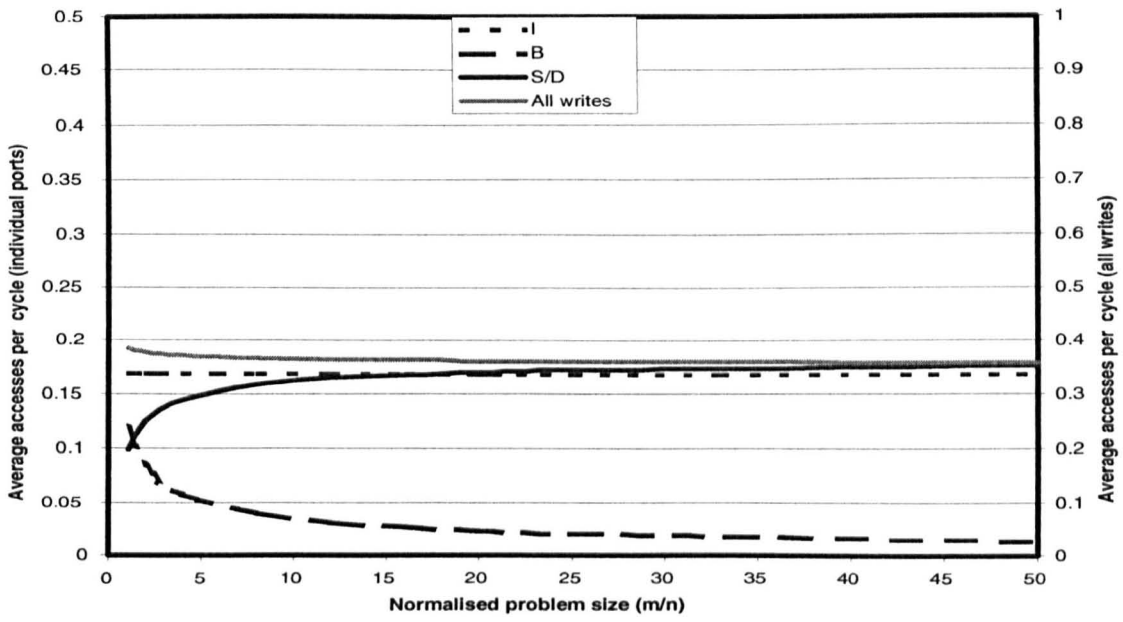


Figure 4.1: Average accesses per cycle on additional ports, n=4 processors.

$$W = \frac{\sum_{inst.} (Write(\$L) + Write(\$S) + Write(\$G))}{N_e} \tag{4.3.2}$$

The initialisation port on other hand is used in register allocation to initialise the \$L0 to the loop index. This port is accessed once when each iteration is allocated to a processor and so the average number of accesses to this port is constant and equal to the inverse of the number of instructions executed by the thread before it is killed, n_o. Therefore if I is the average number of accesses to the initialisation port per cycle, we can say that:

$$I = \frac{1}{n_o} \tag{4.3.3}$$

A dependent read to a remote processor uses a read port on the remote processor and a write port on the local processor, as well as a read to the synchronous pipeline

port on the local processor. The average number of accesses to these ports per cycle is dependent on the type of scheduling algorithm used. If we use modulo scheduling, where M consecutive iterations are scheduled to one processor, then interprocessor communication is minimised. An equation for dependent reads and writes is given based on modulo scheduling although we consider only at the worst case scenario. The average number of accesses per cycle to the dependent window is given below by S_d using the following equation, where M is the number of consecutive threads scheduling to one processor and N_e is the total number of instructions executed. It is clear that the worst case is where $M = 1$, i.e. iterations are distributed one per processor in a modulo manner.

$$S_d = \frac{\sum_{inst.} Read(\$D)}{M * N_e} \tag{4.3.4}$$

The global write port is used to store data from the broadcast bus to the global window in every processor's local register file. If we assume that the average number of accesses per cycle to this port is B_r, then B_r can be obtained from the following equation, where N_e is the total number of instructions executed and n is the number of processors in the system. The result is proportional to the number of processors, as one write instruction will cause a write to every processor in the system.

$$B_r = \frac{\sum_{inst.} Write(\$G) * n}{N_e} \tag{4.3.5}$$

Finally, the frequency of accesses to the port that is required for the deferred register write in the case of a cache miss can also be obtained. It is parameterised by cache miss rate in this static analysis and again we look at the worst case (100% miss rate). The average number of writes per cycle to the cache-miss port is given by Wm,

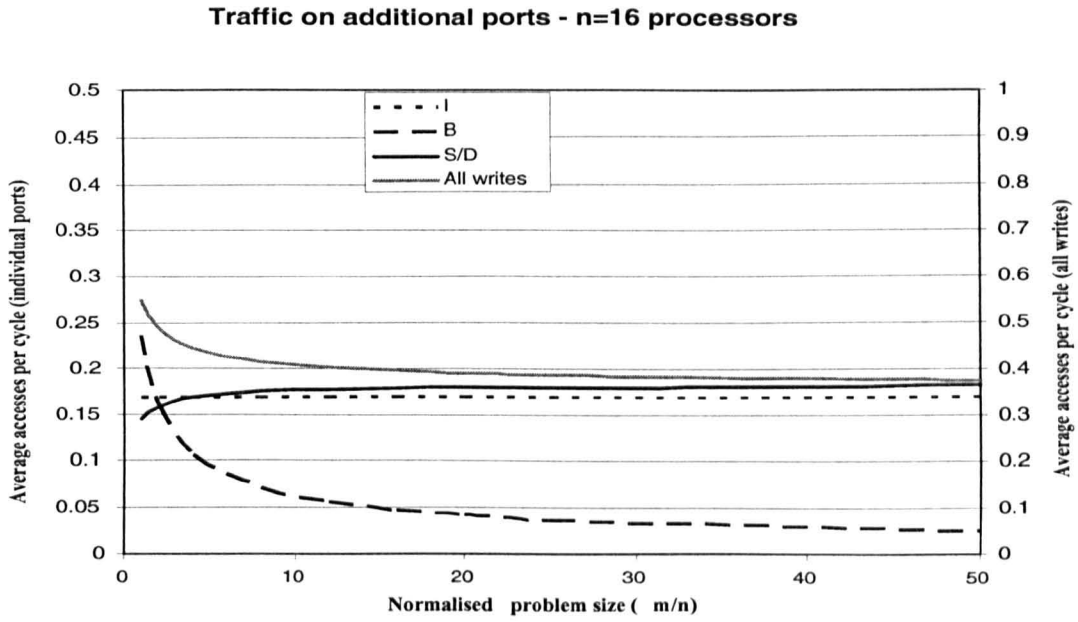


Figure 4.2: Average accesses per cycle on additional ports, n=16 processors.

which is given by the formula below where Lw is the number of load instructions in each thread body, n_o is the number of instructions executed per thread body, and C_m is the cache miss rate. Again the average access to this port is constant for a given miss rate.

$$W_m = \frac{\sum_{inst.}(Lw)}{n_o} * C_m \tag{4.3.6}$$

Table 4.1 shows the average number of accesses to each class of register file port over a range of loop kernels using the above formula. The first seven kernels are dependent loops, where the dependencies are carried between iterations using registers. The last three are independent loops, where all iterations of the loop are independent of each other.

As described previously, each of the distributed register files has four sources for

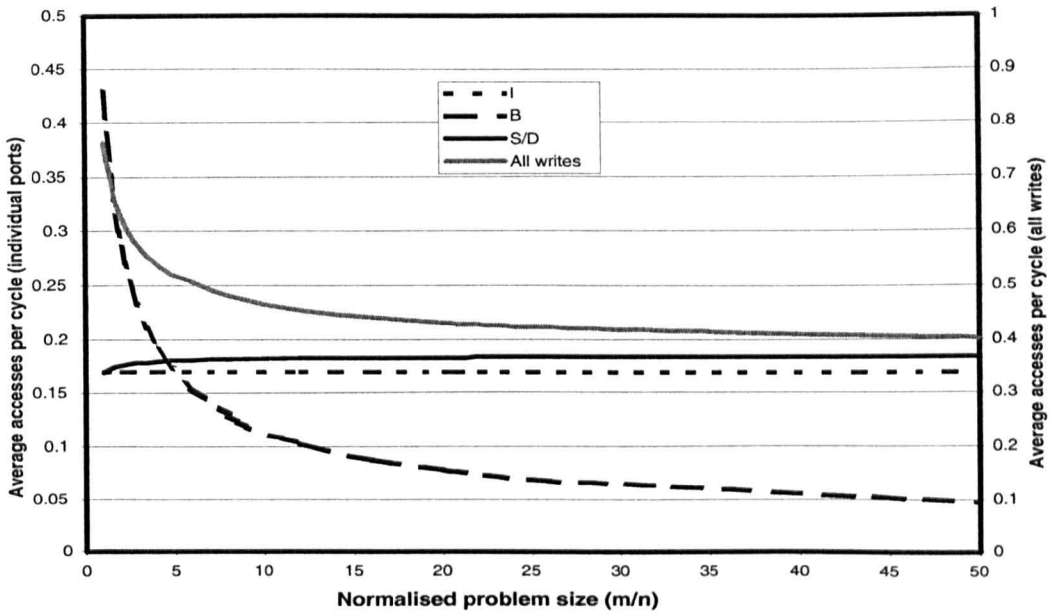


Figure 4.3: Average accesses per cycle on additional ports, n=64 processors.

write accesses in addition to the pipeline ports. These are for $G write, the initialisation write, the $D return data and the write to the port that supports decoupled access to memory on a cache miss. Our analysis shows that the average accesses from these sources is much less than one access per cycle over all analysed loop kernels. This is shown in figures 4.1 to 4.4 where accesses to initialisation (I), broadcast(B_r) and the network ports (S_d, shown as S/D) are given. The four figures illustrate the scalability of the results (from n=4 to n=256 processors). Results are plotted against normalised problem size, where m is the size of the problem in terms of the number of iterations, although not all iterations are executed concurrently in all codes (for example the recursive doubling algorithm has a sequence of concurrent loops varying by powers of 2 from 2 to m/2). Normalised problem size is therefore a measure of the number of iterations executed per processor.

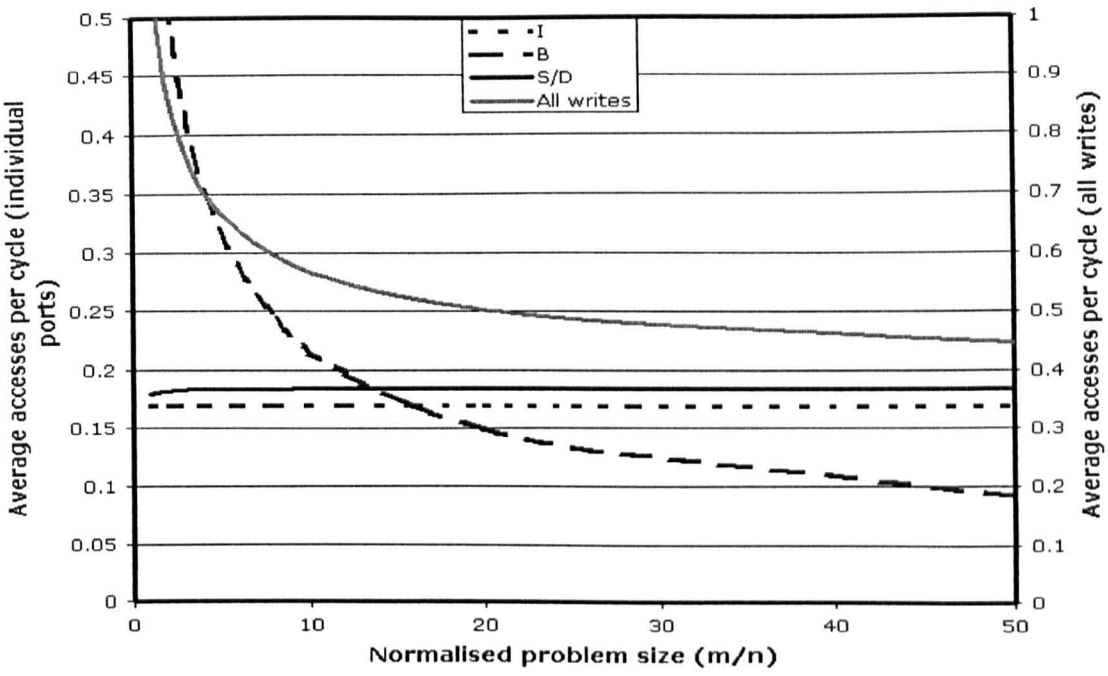


Figure 4.4: Average accesses per cycle on additional ports, n=256 processors.

It can be seen that only accesses from the broadcast bus increase with the number of processors and even this is only significant where few iterations are mapped to each processor. Even in the case of 256 processors, providing we schedule more than a few iterations to each processor, the overall number of writes is less than 50%. Note that a register file of 512 registers supports at least 32 microcontexts per processor. To put this in perspective, this means that on average a single port sharing all I, B_r and S_d writes would be busy only 50% of the time. There may be peaks in the distribution of writes per cycle, however all of these accesses are asynchronous and they can be queued without stalling the operation of any of the pipelines. This still leaves capacity to include writes from the decoupled-memory accesses.

The analysis of the decoupled-memory port also shows that the average number of accesses per cycle is small. If we assume a cache miss rate of 50%, then the average

Table 4.2: Average number of accesses to all additional write ports for different number of processors, m/n=8.

Miss Rate (R %)	Average Accesses to write miss port (Wm)	Average Accesses (All Write Ports) n = 4	Average Accesses (All Write Ports) n = 16	Average Accesses (All Write Ports) n = 64	Average Accesses (All Write Ports) n = 256
10%	0.038	0.405	0.453	0.517	0.634
20%	0.076	0.443	0.491	0.555	0.672
30%	0.113	0.480	0.528	0.592	0.709
40%	0.151	0.518	0.566	0.630	0.747
50%	0.189	0.556	0.604	0.668	0.785
60%	0.227	0.594	0.642	0.706	0.823
70%	0.265	0.632	0.680	0.744	0.861
80%	0.302	0.669	0.717	0.781	0.898
90%	0.340	0.707	0.755	0.819	0.936
100%	0.378	0.744	0.785	0.857	0.974

number of accesses is less than 20% over all loop kernels. Thus, a single write port would not be fully utilised at this miss rate. For completeness, table 4.2 and shows the average number of accesses per cycle to all write ports including the W_m port with a variable cache miss rate. This table is compiled for a normalised problem size where the m/n=8, which corresponds to fully utilising a small register file (see also figure 4.5).

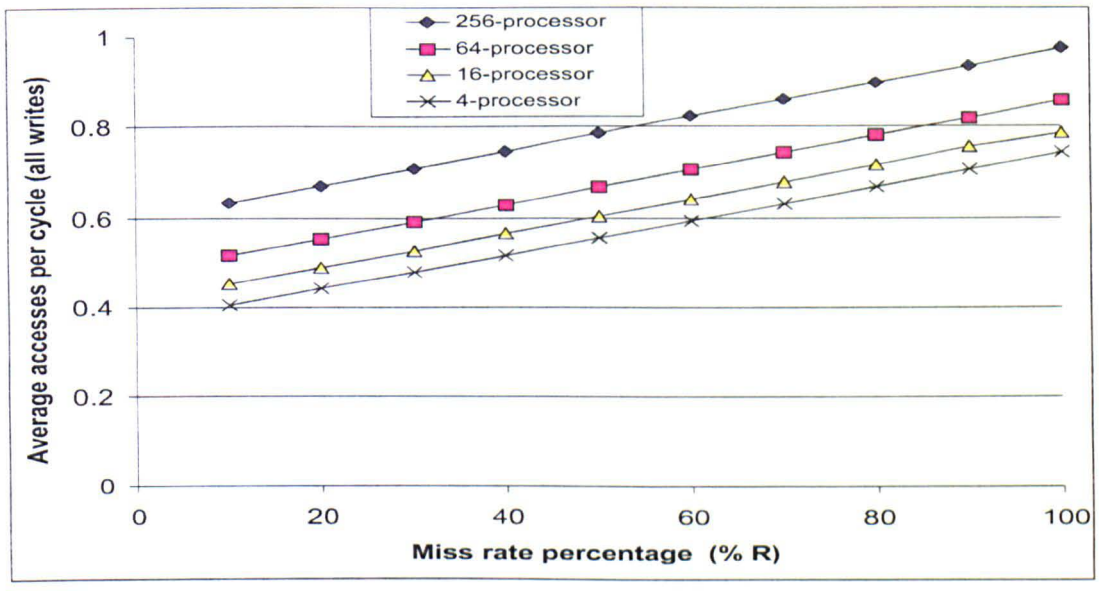


Figure 4.5: Average number of accesses to all additional write ports for different number of processors, m/n=8.

4.4 Registers Allocation

4.4.1 Background

Register allocation and instruction scheduling are two important issues for processors aiming to extract a high level of ILP. Separate instruction scheduling and register allocation leads to poor register utilisation and degrades the amount of parallelism [122]. The complex hardware and mechanisms used in superscalar processors such as register renaming to remove artificial data dependencies between dependent instructions, speculative execution to handle control dependencies and recovery from misspredic-tion, limit the amount of parallelism that can be readily achieved. These approaches also have limited knowledge of when a physical register can be reallocated. In contrast, microthreading provides a dynamic scheduling with dynamic register allocation,

and there is a full transparent interaction between the thread scheduling and threads allocation.

Several previous projects have investigated register allocation such as [123, 124] attempting to exploit more ILP or to allocate more physical registers at runtime. In [123] a mechanism for managing multiple register sets in a register file is proposed. The mechanism includes both compiler and hardware support to reallocate registers. The compiler tries to identify the required number of registers for each thread, and generate code using that number. At execution time, a special instruction is executed (with hardware support) which tries to dynamically group the register sets from all active threads into the register file. This mechanism requires one or more cycles in the execution pipeline, affecting the instruction decode stage. Also it constrains the context sizes, which must be a power-of-two.

Other proposals such as [124] use a hardware mechanism together with an enhanced compiler to allocate more physical registers (not addressable through the ISA) at runtime in order to reduce the spill code problem. The spilling problem is very well known, and most existing processors have a mechanism to handle the situation when the number of physical registers exceeds the register file size. This allocation scheme complicates the hardware design and is not precise in the way registers are allocated - allocation information is conveyed to the hardware through offsets of spills [125].

In the microthreaded model, the distribution of threads to pipelines is deterministic and is based on a simple scheduling algorithm. It is dynamic as it is determined by resource allocation and release (the concurrency exposed is parametric and not limited by the hardware resources). Register allocation in the microthreaded model is performed prior to thread creation, where the RAU in each local scheduler maintains

the allocation state of the local and shared registers required by each thread created. The registers must be allocated in a contiguous block, and the block is defined for a given family of threads. The allocation is based on a base-offset addressing mechanism for each thread, where the base address is the first address of the contiguous registers. The offset is obtained from the register address in the binary code. Indeed, the process of allocating registers before thread creation and releasing these registers when the thread completes provides an efficient and flexible solution to this problem. It also provides an accurate utilisation of the registers in the register file.

4.4.2 Comparing Registers Allocation Design Alternative

In this section, we will discuss in brief one possible implementation choice for allocating registers in microthreaded CMP, that we have already investigated throughout our research. The implementation is based on a set of components, which are required to perform the allocation and de-allocation processes. The components comprise *Free-block table*, *Allocated-block table*, and *Slice table*. The free block table contains information about a contiguous block of registers. Initially, there is one free block and it contains all registers in the register file above register 31, note that 0-31 are the "architectural registers", which are addressed by the main thread and which provide compatibility with any given instruction set architecture. Also, two registers *fstart*, which is a pointer to the first free block in the free block table and a register *ffree*, which is a pointer to a list of free entries in the table. Note that the philosophy of this algorithm is to deal with slices of registers, which is a function of a given family of threads, and not individual registers. In particular, the block may contains a set of slices, and each slice is equal (in size) to number of local (L) and shared (S) registers

per thread i.e. *L+2S*.

The allocated-block table contains information about the register blocks that have been allocated. Initially, there are no allocated blocks. Again these represent contiguous blocks of registers allocated to one family of microthreads. Also, in this table two pointers are required. The first is *astart*, which is a pointer to the first allocated block in the allocated-block table. The second pointer is *afree*, which is a pointer to a list of free entries in the table. Finally, there is the slice table, which contains information about slices that have been allocated; its main use is to monitor slice reuse. It uses a slice counter (breleased) to count the number of released slices (sreleaesed). Thus, when the slice counter becomes equal to the size of the allocated block *(fs)*, then the allocated block is released and becomes free again. Note that, the slice table index is equivalent to the CQ slot number.

The allocation and release processes work concurrently, and a brief pseudo-code algorithm description of each of these processes is shown in figure 4.6. In effect, we investigated this implementation and we found that it has multiple limitations such as:

- Three memory structures (tables) with pointers and extra logic are required to perform the allocation process. This is costly, and does not scale well.
- The strategy of allocating blocks and then deallocating these blocks when they are released results in fragmentation the register files, and inefficient used for the allocation scheme.
- A merging technique is required to merge two contiguous blocks when this becomes possible, but the problem is that the merging process requires extra

```
Allocate : Process Allocation(Thread parameters);
begin
For each thread family:
 Get thread family parameters ;
 Loop: While family not complete do:
      Get next free block parameters;
      Find a free allocated block;
      While block not full:
          Allocate slice;
          Update free-block parameters;
          Update allocated-block parameters;
          Update slice parameters;
 End loop;
end process Allocate;


Release : Process (Rlease Parameters);
bgin
 sreleased = sreleased - number(releases for slice);
 When any slice's sreleased reached zero;
    breleased = slice + breleased;
    When any allocated-block brealeased counter = fs;
        Create new free block;
        Merge free blocks if possible;
        Return allocated block to free list;
        Merge free allocated blocks is possible;
end process Release;
```

Figure 4.6: An alternative algorithm for allocation scheme.

cycles, which in terms increase the allocation overhead.

- To perform the allocation process, more than one cycle is required, which is necessary to update the required pointers, and parameters.

Finally, this implementation choice is complex, costly, and has multiple limitations. Therefore, in the next section we will discuss an alternative implementation, which has multiple advantages. The design is largely in combinational logic and it can perform the allocation and de-allocation processes very quickly. It provides a simple and scalable solution for dynamically allocating and de-allocating registers.

4.5 Dynamic Register Allocation Scheme For Microthreaded CMPs

4.5.1 Description of the Allocation Scheme

In this section, we describe the hardware mechanism for dynamically allocating and de-allocating registers to families of microthreads. The hardware uses information provided by the compiler through the TCB to define the allocation requirements, and a set of 1-bit flags to model the allocation state of the registers. The free registers are split into a block of the appropriate size and the remaining registers (if any) continue to be flagged as free. Initially there is one free block and it contains all registers in the register file above register 31.

It is important to perform the allocation process in a minimum number of cycles. Our scheme allocates one microcontext per cycle, which is the fastest rate and corresponds to a single thread per microcontext; the allocation may be amortised over a number of threads if there is ILP defined in the TCB. Design tradeoffs can be made that allocate in units of a few registers, rather than single registers. The scheme has an area proportional to the product of number of allocation units in the register file and the number of bits in the register specifier. For a given ISA, or number of bits in the register specifier, the allocation has a constant (worse-case) time delay. An analysis of 10 Livermore loop kernels, including both independent and dependent loops, gave an average number of registers required per microcontext of 6 as shown in table 4.3, and a minimum number of 3 (one for the loop index). The area can be saved by allocating in units of greater than one, as allocating a unit of n-registers reduces the complexity of the allocation scheme by a factor of n.

Table 4.3: Number of required registers per allocation over a range of loop kernels.

Loop Name	Number of $L	Number of $S	Total $
A: Partial Product	2	1	3
B: 2-D SOR	3	1	4
L3: inner Product	3	1	4
L4: Banded Linear Equation	7	2	9
L5: Tri-Diagonal Elimination	3	1	4
L6: General Linear Recurrence	7	1	8
C:Pointer Chasing	3	1	4
L1: Hydro Fragment	6	0	6
L2: ICCG	12	0	12
L7: Equation of State Fragment	10	0	10
Average Registers per thread			**6**

Figure 4.7 shows the Top-level design of the RAU and its interaction with the thread-create process and hence, the rest of the scheduler. It shows that the RAU comprises an iterative array of allocation slices, one slice per n-register block. Information on the action required (no op., allocate or release), the required block size (for allocation), and the base address (for release) is supplied to each slice from the scheduler. Each slice maintains a flag, which indicates whether the corresponding section of the register file is free or not. Note that in cycles when no action (allocate or release) is being performed, the RAU still calculates the next base address ready for allocation, so that it is available before an allocation is actually required.

In figure 4.7 data ripples through the allocation slices from bottom to top, corresponding to increasing register-file addresses. The output from the final slice identifies the base address in the register file of the first free contiguous block that meets the current block size requirement for allocation (if one exists). The scheduler uses this to determine whether the current allocation round can proceed and to set the base address in the CQ of any threads associated with this microcontext.

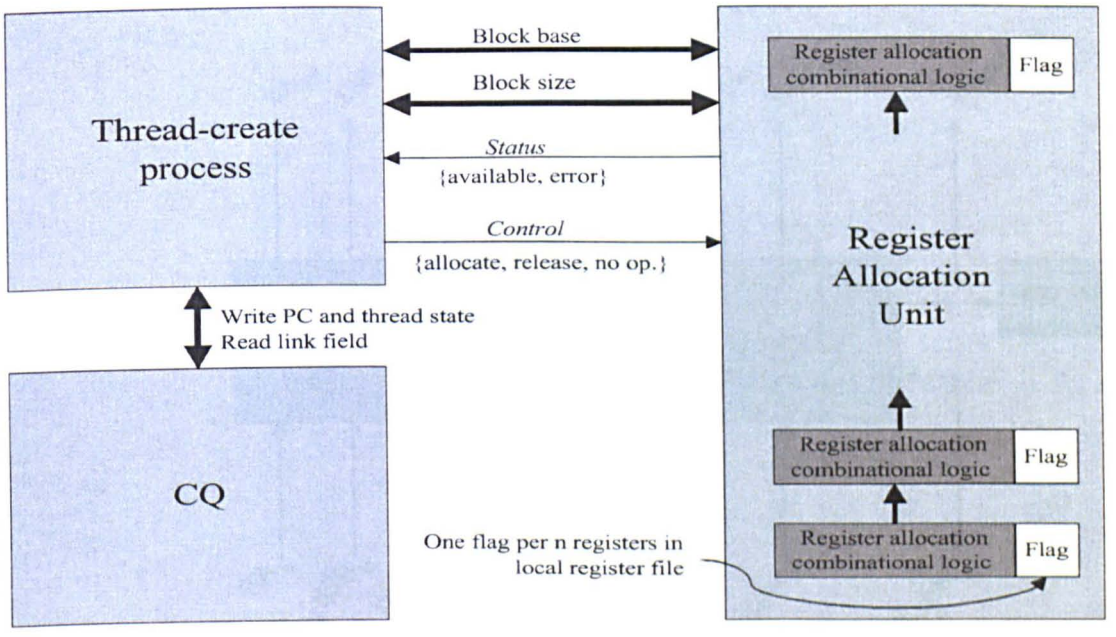


Figure 4.7: Block diagram of the RAU and its interaction with the thread-create process.

The ripple inputs to the first slice are not hardwired, but held in a register to facilitate test and adaptability. The base address input of the first slice is held at the address of the first register that can be allocated in the register file, which in the scheme described above would be address 32. This register would remain constant using the simple models adopted in our work to date but could support recursive microcontexts, where a microcontext in one family could become the global context for a subordinate family. This would support concurrent nested loops for example.

Information propagated from slice to slice includes whether a free block has been found, the base address of the largest free block, the size of the largest free block, the base address and size of the current free block. An error flag is also propagated which indicates if inappropriate inputs have been applied to the RAU. The data manipulated and propagated between slices is listed in full in table 4.4 and is illustrated

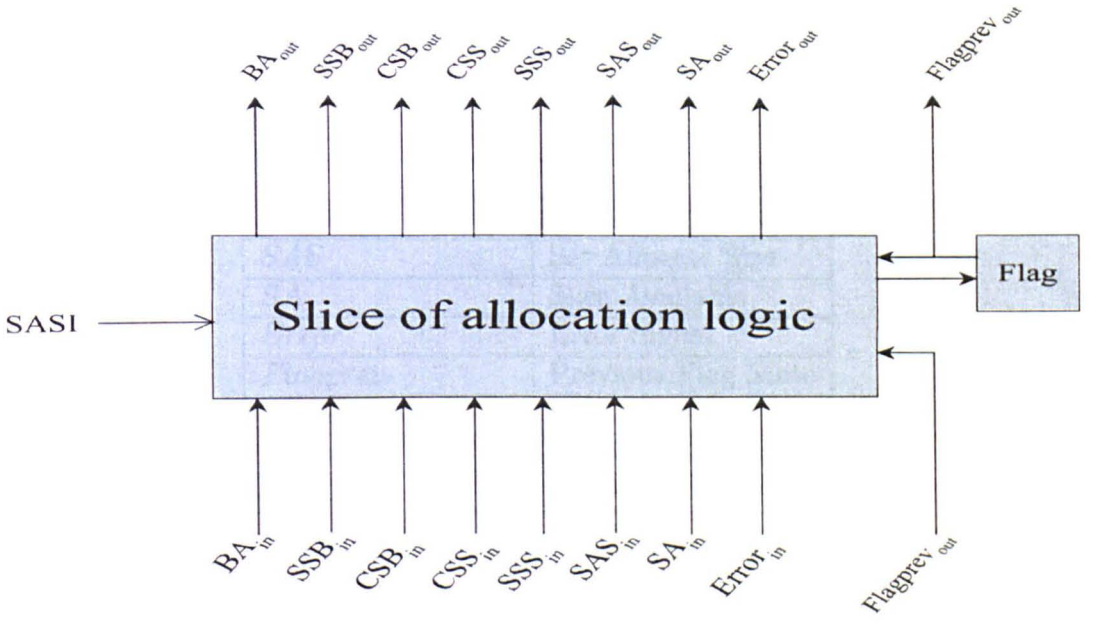


Figure 4.8: Register allocation unit's combinational logic slice.

in figure 4.8. As already explained, the number of registers per microcontext is at least one and less than or equal to 16, so it is possible to limit the size fields to 4 bits, which can significantly reduce the propagation time within a slice and hence the time to perform the allocation update process. Note that each slice performs an increment on this field. The state of the allocator is held entirely in a set of flags, one per slice, which indicates if the associated n-register block is available for allocation or not.

If the RAU is ready, the scheduler can initiate an allocation immediately, which completes in a single cycle, after which the state of the allocator updates. The update process, may take longer than one cycle, depending on the size of the register file and allocation unit size (n), which defines the RAU's ripple-through time. The rate of performing allocations will, on average, be less than one per cycle, as each microcontext may have many threads associated with it. Also the recovery time is

Table 4.4: Allocation logic parameters.

Abbreviation name	Description
BA	Base Address
SSB	Selected Slice Base
CSB	Current Slice Base
CSS	Current Slice Size
SSS	Selected Slice Base
SAS	Set Allocate Size
SA	Slice Available
Error	Error Signal
Flagprev	Previous Flag State
Flagout	New Flag state
Flagin	Current Flag State
SASI	Set Allocate Size In
Reg	Register

less important than the latency of allocation from request to allocation completion.

When a thread or group of threads associated with a microcontext are killed, then the scheduler also causes the allocation model in the RAU to be updated to reflect this, by providing the base address of the microcontext being released and its size. This information propagates through the slices to determine which flags to reset. The algorithm implemented by the RAU is described qualitatively below:

- Find the start address (base address) and the size of the first and largest free chunk in the register file (or determine that no space is available).
- If space is available and the size of the available block is greater than or equal to the required size, identify the portion of the free block required for the allocation starting at its base address.
- Flip the corresponding flags of that chunk in the register-use model.
- When a release occurs, the base address provides a pointer to the beginning

```
For each thread family:
If (event= allocate and block_size > required_size and space_available) then
  Allocate_thread( )
Else if (event= release) then
  Release_thread( )
```

Figure 4.9: General action of the allocation scheme.

of the corresponding chunk to flip the corresponding flags in the register use model.

The corresponding general action of the allocation scheme is shown in figure 4.9.

The allocation scheme is straightforward and allocating registers in units n provides both area and propagation-delay reduction in the scheme. If we assume that the size of the register file is R, and the number of registers allocated in a unit of allocation is n, then the complexity of the allocation scheme is proportional to $O(\frac{R}{n})$.

4.5.2 Implementation and Simulation Results for the Allocation Scheme

This section provides the implementation methodology and simulation results for the allocation scheme described in the previous section. As described earlier, VHDL simulation is used for initial system component verification. VHDL allows each component in the model to be described independently with its required internal behaviour by defining how its outputs behaves when certain conditions are applied to its inputs.

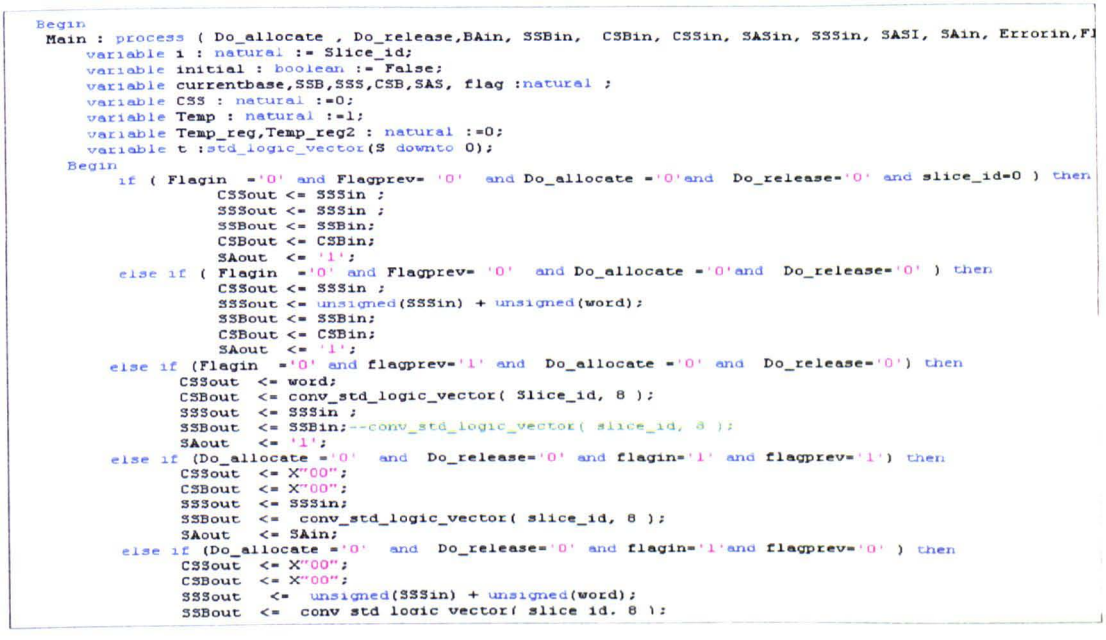
```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use STD.TEXTIO.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_arith.all;
Entity Slice is
    Generic (
              M              : integer :=31;                    -- Max slice size
              S              : integer :=7;                     -- Number of bits
              Slice_id       : integer;                         -- Slice Number
            );
    port    (
              BAin            :in  std_logic_vector(S downto 0); -- Base address in
              SSBin           :in  std_logic_vector(S downto 0); -- Slice Size Base in
              CSBin           :in  std_logic_vector(S downto 0); -- Current SLice Base in
              CSSin           :in  std_logic_vector(S downto 0); -- Current Slice Size in
              SASin           :in  std_logic_vector(S downto 0); -- Set allocate Size in
              SSSin           :in  std_logic_vector(S downto 0); -- Selected Slice Size in
              SASI            :in  std_logic_vector(S downto 0); -- Set allocate size
              SAin            :in  std_logic;                    -- Slice Available
              Errorin         :in  std_logic;                    -- Error Signal in
              Flagin          :in  std_logic;                    -- Flag Input
              Flagprev        :in  std_logic;                    -- Previous value
              Do_allocate     :in  std_logic;                    -- Do allocate
              Do_release      :in  std_logic;                    -- Do release
              BAout           :out std_logic_vector(S downto 0); -- Base address out
              SSBout          :out std_logic_vector(S downto 0); -- Slice Size Base out
              CSBout          :out std_logic_vector(S downto 0); -- Current Slice Base out
              CSSout          :out std_logic_vector(S downto 0); -- Current Slice Size out
              SASout          :out std_logic_vector(S downto 0); -- Set allocate Size out
              SSSout          :out std_logic_vector(S downto 0); -- Selected Slice Size out
              SAout           :out std_logic;                    -- Slice Available
              Errorout        :out std_logic;                    -- Error signal out
              Flagout         :out std_logic                     -- Flag state
            );
  End Slice;
Architecture Combination_Alloc of Slice is
constant ZeroWord           : std_logic_vector(S downto 0) := (others => '0');
```

Figure 4.10: Allocation scheme entity description source code.

To verify and test the behaviour of the allocation scheme, a higher-level model has been written for the allocation scheme.

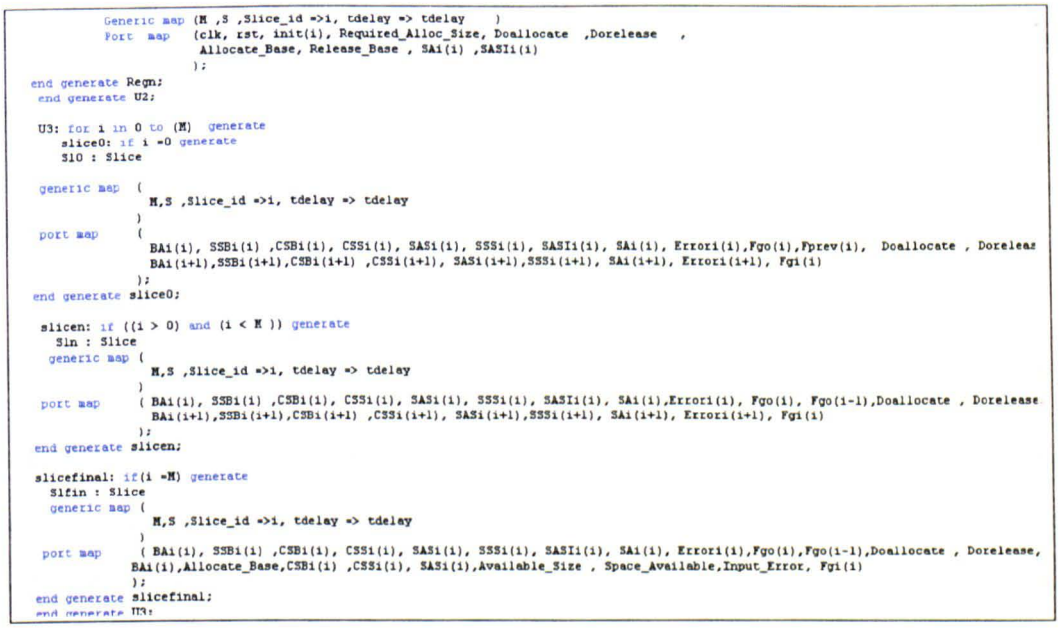
We have modeled the behaviour of the allocation scheme in VHDL language, exploiting the *generate* statement provided by this language to create the allocation unit. In particular, the model includes the allocation entity, its architecture behaviour and the test bench as shown in figures 4.10 to 4.12 respectively.

A set of public and *generic* definitions are used as a default value to pass structural information to system components that are referenced by this definition. This is important to maintaining the modularity of the allocation scheme, where the actual parameters can be changed without changing any component of the model. Note that in this implementation, we consider that all signals and variables have a corresponding value assigned in any branch of the *IF* statement.



```
Begin
 Main : process ( Do_allocate , Do_release,BAin, SSBin,  CSBin, CSSin, SASin, SSSin, SASI, SAin, Errorin,Fl
     variable i : natural := Slice_id;
     variable initial : boolean := False;
     variable currentbase,SSB,SSS,CSB,SAS, flag :natural ;
     variable CSS : natural :=0;
     variable Temp : natural :=1;
     variable Temp_reg,Temp_reg2 : natural :=0;
     variable t :std_logic_vector(3 downto 0);
   Begin
        if ( Flagin  ='0' and Flagprev= '0'  and Do_allocate ='0'and  Do_release='0' and slice_id=0 ) then
                  CSSout <= SSSin ;
                  SSSout <= SSSin ;
                  SSBout <= SSBin;
                  CSBout <= CSBin;
                  SAout  <= '1';
        else if ( Flagin  ='0' and Flagprev= '0'  and Do_allocate ='0'and  Do_release='0' ) then
                  CSSout <= SSSin ;
                  SSSout <= unsigned(SSSin) + unsigned(word);
                  SSBout <= SSBin;
                  CSBout <= CSBin;
                  SAout  <= '1';
      else if (Flagin  ='0' and flagprev='1' and  Do_allocate ='0' and  Do_release='0') then
             CSSout  <= word;
             CSBout  <= conv_std_logic_vector( Slice_id, 8 );
             SSSout  <= SSSin ;
             SSBout  <= SSBin;--conv_std_logic_vector( slice_id, 8 );
             SAout   <= '1';
      else if (Do_allocate ='0'  and  Do_release='0' and flagin='1' and flagprev='1') then
             CSSout  <= X"00";
             CSBout  <= X"00";
             SSSout  <= SSSin;
             SSBout  <=  conv_std_logic_vector( slice_id, 8 );
             SAout   <= SAin;
       else if (Do_allocate ='0'  and  Do_release='0' and flagin='1'and flagprev='0' ) then
             CSSout  <= X"00";
             CSBout  <= X"00";
             SSSout   <=  unsigned(SSSin) + unsigned(word);
             SSBout  <=  conv_std_logic_vector( slice_id, 8 );
```

Figure 4.11: Allocation scheme architecture behaviour source code.



```
          Generic map (M ,S ,Slice_id =>i, tdelay => tdelay    )
          Port  map   (clk, rst, init(i), Required_Alloc_Size, Doallocate  ,Dorelease   ,
                       Allocate_Base, Release_Base , SAi(i) ,SASIi(i)
                      );
  end generate Regn;
   end generate U2;

   U3: for i in 0 to (M)  generate
      slice0: if i =0 generate
      S10 : Slice

   generic map  (
                 M,S ,Slice_id =>i, tdelay => tdelay
               )
   port map     (
                 BAi(i), SSBi(i) ,CSBi(i), CSSi(i), SASi(i), SSSi(i), SASIi(i), SAi(i), Errori(i),Fgo(i),Fprev(i),  Doallocate , Dorelea
                 BAi(i+1),SSBi(i+1),CSBi(i+1) ,CSSi(i+1), SASi(i+1),SSSi(i+1), SAi(i+1), Errori(i+1), Fgi(i)
               );
  end generate slice0;

   slicen: if ((i > 0) and (i < M )) generate
     Sin : Slice
    generic map (
                 M,S ,Slice_id =>i, tdelay => tdelay
               )
   port map     ( BAi(i), SSBi(i) ,CSBi(i), CSSi(i), SASi(i), SSSi(i), SASIi(i), SAi(i),Errori(i), Fgo(i), Fgo(i-1),Doallocate , Dorelease
                 BAi(i+1),SSBi(i+1),CSBi(i+1) ,CSSi(i+1), SASi(i+1),SSSi(i+1), SAi(i+1), Errori(i+1), Fgi(i)
               );
  end generate slicen;

  slicefinal: if(i =M) generate
    Slfin : Slice
    generic map (
                 M,S ,Slice_id =>i, tdelay => tdelay
               )
   port map     ( BAi(i), SSBi(i) ,CSBi(i), CSSi(i), SASi(i), SSSi(i), SASIi(i), SAi(i), Errori(i),Fgo(i),Fgo(i-1),Doallocate , Dorelease,
              BAi(i),Allocate_Base,CSBi(i) ,CSSi(i), SASi(i),Available_Size , Space_Available,Input_Error, Fgi(i)
              );
  end generate slicefinal;
  end generate U3;
```

Figure 4.12: Allocation scheme test bench source code.

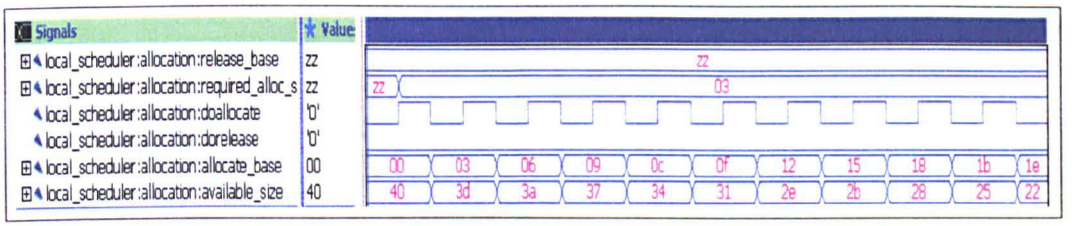
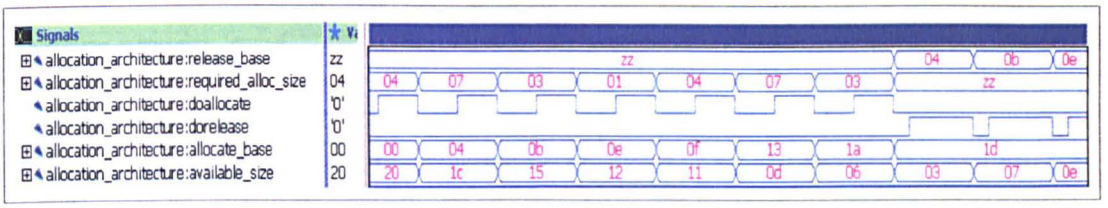


Figure 4.13: Simulation waveforms for allocation three registers per thread (Register file size is 64-registers).

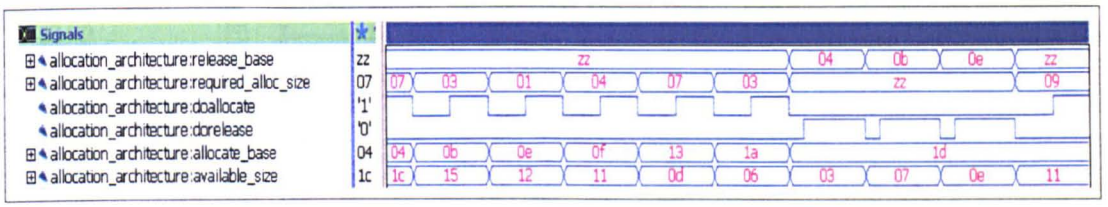
The VHDL model was simulated using various compile scenarios and with different thread allocation size implementations. In effect, a variety of test cases were simulated and the output results compared with the required behaviour of the allocation scheme. For example; figures 4.13 and 4.14 shows a snapshot of simulation waveforms for the allocation scheme. In figure 4.13 we assume that the given family of microthreads required a slice with three registers in each allocation cycle. While figure 4.14 shows allocation and de-allocation for a different slice sizes. Also figure 4.15 shows waveforms for the allocation logic parameters values. We will also show the behaviour of our allocation simulation together with the microthreaded in-order pipeline in chapter 7. Note that the allocation scheme VHDL source code and more waveforms simulation results are described in appendix B.

4.6 Summary

This chapter presented two contributions. The first includes an analysis of the requirements of the microthreaded register file in term of frequency of access to asynchronous (non-pipeline) ports in the synchronising memory. The result of analysing a range



a) Wavforms Sample one



b) Wavforms Sample two

Figure 4.14: Simulation waveforms for allocation and de-allocating different slice sizes per thread (Register file size is 32-registers).

of different code kernels shows that a distributed shared register file could be implemented with only 5-ports per processor, where three ports provide single instruction issue per cycle and the other two asynchronous ports were able to manage all other demands on the local register file. In fact, the decoupled approach to register-file design avoids a centralised register file organisation and, as we have shown, requires a small, fixed number of ports to each processor's register file, regardless of the number of processors in the system.

The analysis involved different types of dependent and independent loop kernels. The analysis illustrates a number of interesting issues, which can be summarised as follows:

- A single write port with arbitration between different sources is sufficient to support all non-pipeline writes. This port has an average access rate of less than 100% over normal operating conditions. This is true even in the case of a

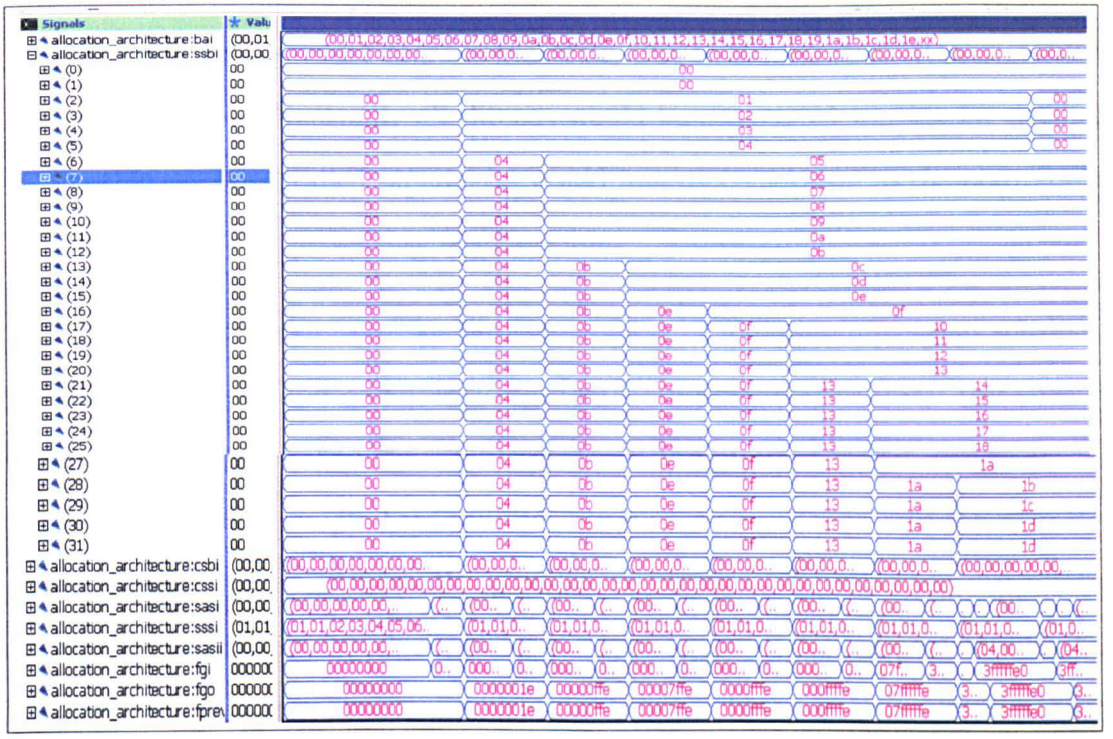


Figure 4.15: Simulation waveforms showing slice parameters values (three registers per thread).

100% cache-miss rate.

- A second port is required to handle reads to the $D-window. The analysis shows that the average access to this port is less than 10% over all analysed loop kernels.

- As a consequence, the distributed register files require only five ports per processor and these ports are fixed regardless of the number of processors in the system. This provides a scalable and efficient solution for large numbers of processors on-chip.

- Finally, the average accesses to all write ports does not exceed 100% even in the

case of n= 256-processor. However, to deal with a large number of processors, the performance would degrade gracefully due to the inherent latency tolerance of the model. Eventually all threads would be suspended waiting for data and in this case the stalled pipeline(s) would free up contention to the non-pipeline write port.

In the second contribution an implementation of a simple allocation scheme to dynamically allocate and de-allocate registers for microthreaded CMP has been described. The scheme behaviour was verified and tested using VHDL language. The scheme employs very simple hardware combinational logic design and the allocation process is fully decoupled from the pipeline execution. The allocator can allocate registers in fixed blocks, which simplifies the logic and reduces the area significantly. Chapter 7 addresses the area scalability of this issue, where an area estimate to the allocation scheme compared with the actual register file area is given.

Chapter 5

Microgrid Chip Multiprocessor Architecture Model

5.1 Chapter Overview

The term microgrid refers to a CMP where all processors have a microthreaded scheduler and a synchronising, distributed register file. This chapter introduces the microgrid CMP architectural model, discusses its components, and highlights the problems that will be resolved in chapter 6 and chapter 7.

The chapter is organised as follows. The next section discuses a top-level architecture model of the microgrid CMP. The chip communication buses and its main uses are presented in section 5.3. Microgrid CMP supports Globally Asynchronous Locally Synchronous *(GALS)* communication and this issue and its features are presented in section 5.4. Microthread scheduling and thread distribution to support microgrid operation is presented in section 5.5. In section 5.6, an overview about microgrid input/output routines is presented. Microgrid CMP Scalability is discusses in section 5.7. Finally, we present the summary of the chapter in section 5.8.

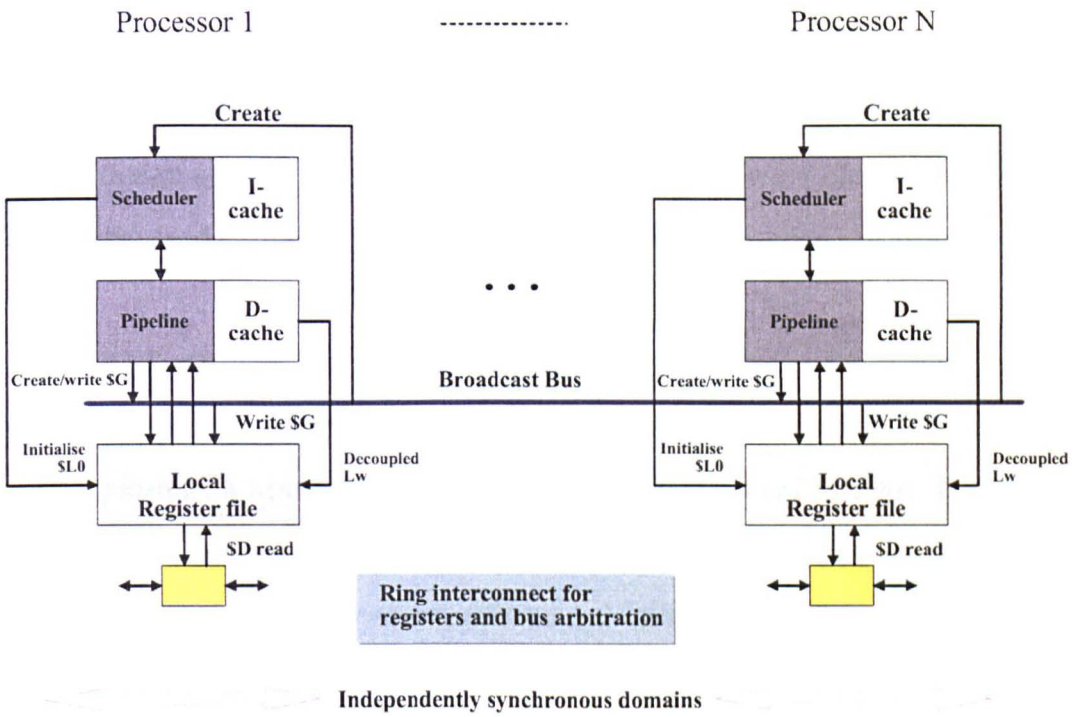


Figure 5.1: Microthreaded CMP architecture, showing communication structures and clocking domains.

5.2 Microgrid CMP Top-level Architecture Model

Microgrid refers to a CMP where all processors have a microthreaded scheduler and a synchronising, distributed shared register file. A microgrid will have an interprocessor network to support the sharing of microcontexts between microthreads in a family of microthreads, mapped to different processors. The network also supports the broadcast of shared-register variables and the parameters defining the creation of a family of microthreads. A microgrid may also have a systems environment processor that manages the allocation of processors to families of threads dynamically and configures the network accordingly. A long-term vision is considered in the design and

component organisation of the microgrid CMP architecture model. This vision comes from the fact that most existing CMP designs suffer from hardware and software implementation problems. Thus microgrid CMP avoids global clocking by supporting a *GALS* design approach, where each microthreaded processor has its own local clock domain and accesses global resources asynchronous. Microgrids are described further in [24].

Figure 5.1 gives an overview of such a microgrid, showing the networks required and the datapaths between the major components within a processor. These are an *in-order pipeline*, a *scheduler*, a large *register file* and a *local I-cache*. The processor may also have a *local D-cache* but latency tolerant access to data means this is not a necessity. In a profile of processors, a subset of the microgrid, any processor can create a family of microthreads for execution on that subset. This requires the distribution to each processor of the address of a data block in memory. This is the previously described TCB, which contains all of the parameters that define the family of microthreads. This is the only global communication required in the execution of a family of microthreads, apart from those defined by memory accesses in the code. Each processor receiving the address of the TCB will execute a deterministic subset of that family, based on the parameters in the TCB, the number of processors in the profile and its position in the profile.

A microgrid has two main buses, the *Broadcast Bus* and the *Shared-register Ring Network*. The broadcast bus allows the register file to be fully distributed between multiple processors. The shared-register ring network is used by the processors to communicate results along a dependency chain. For the model described, this requires

only local connectivity between independently clocked processors. All global communication systems are decoupled from the operation of the microthreaded pipeline and thread scheduling provides latency hiding during remote access. This technique gives a microgrid CMP a serious advantage as a long-term solution to silicon scaling. The next section describes microgrid buses in more detail.

5.3 Microgrid CMP Communication Buses

5.3.1 Broadcast Bus

The Broadcast bus enables one processor to create a family of identical threads. This bus arbitrates between multiple processors and in each cycle one processor can access this bus to create a descriptor of a new family of microthreads. The descriptor identified in the create process is distributed to each scheduler, which uses that information to determine the subset of the family of threads it will execute. It will probably also be used by the same processor to distribute any loop invariants and finally, if there is a scalar result, one processor may write values back to global locations. This situation occurs when searching an iteration space, it is the only situation where contention might be required, as a number of processors might find a solution simultaneously and attempt to write to the bus. In this case a break instruction acquires the bus and terminates all other threads allowing the winner to write its result back to the global state of the main context.

The rate of accessing the broadcast bus depends on the behaviour of the create instruction. It is inversely proportional to the number of threads and cycles required by each thread. If we assume that N_t is the number of threads scheduled to one

Table 5.1: Relative frequency of create instruction over a range of loop kernels.

Loop Name	**Create instruction rate** compared to other instructions
A: Partial Product	0.3333
B: 2-D SOR	0.2
L3: inner Product	0.25
L4: Banded Linear Equation	0.2
L5: Tri-Diagonal Elimination	0.25
L6: General Linear Recurrence	0.1429
C:Pointer Chasing	0.0714
L1: Hydro Fragment	0.1111
L2: ICCG	0.0909
L7: Equation of State Fragment	0.0385

processor and the number of cycles required by each thread is C_t, then the average number of accesses to the create port (Cre_{freq}) per cycle can be given by the following formula:

$$Cre_{freq} = \frac{1}{C_t * N_t} \tag{5.3.1}$$

The frequency of executing this instruction over a range of loop kernels as shown in table 5.1 is very low. The loops considered included a number of livemore kernels, some that are independent and some that contain loop carried-dependencies. Figure 5.2 also shows the frequency of executing create instruction over a range of loop kernels against the normalised problem size, where m is the size of the problem in terms of the number of iterations, and n is the number of processors. As shown, the frequency of executing this instruction is very low, and the percentage of executing this instruction is less than 17% over all loop kernels considered in this analysis. It is important to note that microthreaded processors are tolerant to latency when they

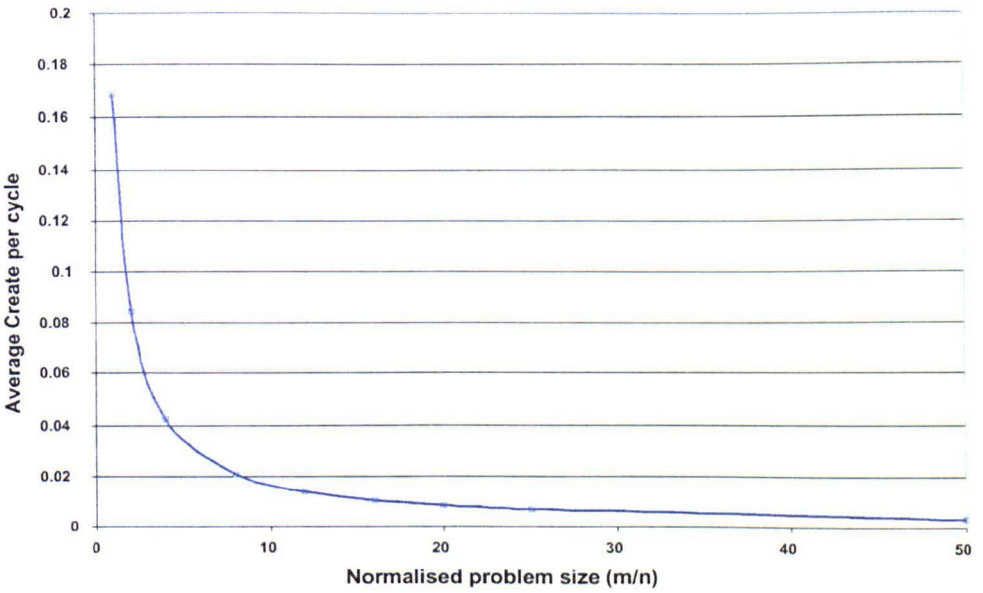


Figure 5.2: Frequency of executing create instruction over a range of loop kernels, m= problem size.

have created threads.

Even if the access to the broadcast bus is at a low frequency, a form of arbitration mechanism is required to avoid contention and to provide fairness in communication between processors. Also, it is necessary to investigate the implementation of the microgrid CMP bus interface with its required signals. The next chapter discusses these issues and introduces a novel asynchronous arbiter optimised for this application.

5.3.2 Point-to-Point Ring Interconnection Network

One of the most important issues in designing a CMP which effects and limits the scalability is the interconnection network. The network allows the processors to share

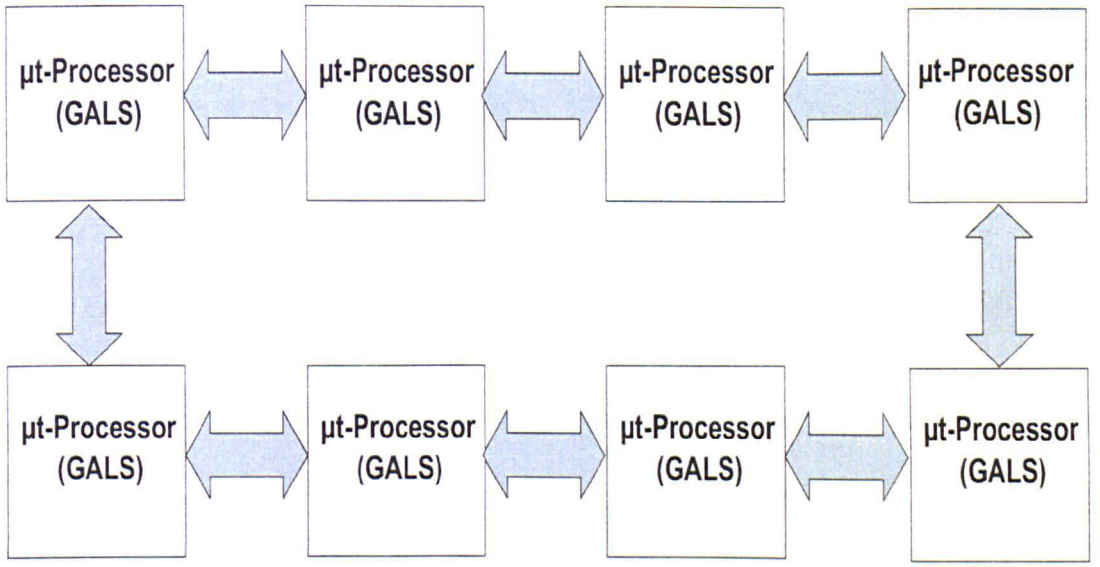


Figure 5.3: Point-to-point communication between microthreaded processors.

data. In fact, low latency, enough bandwidth and scalability are all important requirements in choosing the interconnection network topology. For this reason we use a ring interconnection network in the microgrid CMP that it is scalable and, given sufficient resources, can adopt a schedule which ensures that any constant-strided, loop-carried dependency be mapped to a neighbouring processor. The ring does not suffer from bus bottlenecks and provides a point-to-point connection between nearest-neighbour processors. Moreover, wire complexity in a ring network is low compared with other network topologies. Concerning throughput, latency, and area requirement, the ring occupies a position somewhere in between the shared bus and the switch solution [126].

Use of a shared-register ring network in a microgrid CMP allows communications between pairs of threads, one of which produces data and the other which consumes it. This communication, as shown in figure 5.3, is between the *shared* and *dependent*

threads and will be performed by the ring network if the threads are allocated to different processors. Note that schedules can be defined to minimise inter-processor communication, more importantly; this communication is totally decoupled from the pipeline's operation through the use of explicit context switching.

5.4 Globally Asynchronous Locally Synchronous (GALS) Design Approach

Modern synchronous CMP architectures are based on single clock domain with global synchronisation and control signals. The control signal distribution must be very carefully designed in order to meet the operation rate on each component used and the larger the chip, the more is the power that is required to distribute these signals. In fact, clock skew, and the large power consumption required to eliminate it, is one of the most significant problems in modern synchronous processor design.

Full asynchronous design is difficult but one promising technique is to use a Globally-Asynchronous, Locally- Synchronous (GALS) clocking scheme [127]. This approach promises to eliminate the global clocking problem and provides a significant power reduction over globally synchronous designs. It divides the system into multiple independent domains, which are independently clocked but which communicate in an asynchronous manner. A GALS system not only mitigates against the clock distribution problem, the problem of clock skew and the resulting power consumption, it can also simplify the reuse of modules as they have asynchronous interfaces that do not require redesign for timing issues when composed [128].

In CMP design, global communication is one of the most significant problems in

both current and future systems [8], yet not every system can be decomposed into asynchronously communicating synchronous blocks easily, there must be a clear decoupling of local and remote activity. To achieve this, the local activity should not be overly dependent on a remote communication. The model we have described has just this property; each processor is independent and when it does need to communicate with other processors, that communication occurs independently without limiting the local activity. In short, the local processor is tolerant of any latency involved in global communication, as in most circumstances it will have many other independent instructions it can process and, if this is not the case, it will simply switch off its clocks, reduce its voltage levels and wait until it has work to accomplish while dissipating minimal power.

The size of the synchronous block in a microthreaded CMP can be from a single processor upwards. The size of this block is a low-level design decision. The issue is that as technology continues to scale this block size will scale down with the problems of signal propagation. Thus the model provides solutions to the end of scaling in silicon CMOS. Compare this with the current approach, which seeks to gain performance by clock speed in a single large wide-issue processor where all strategies are working against the technology.

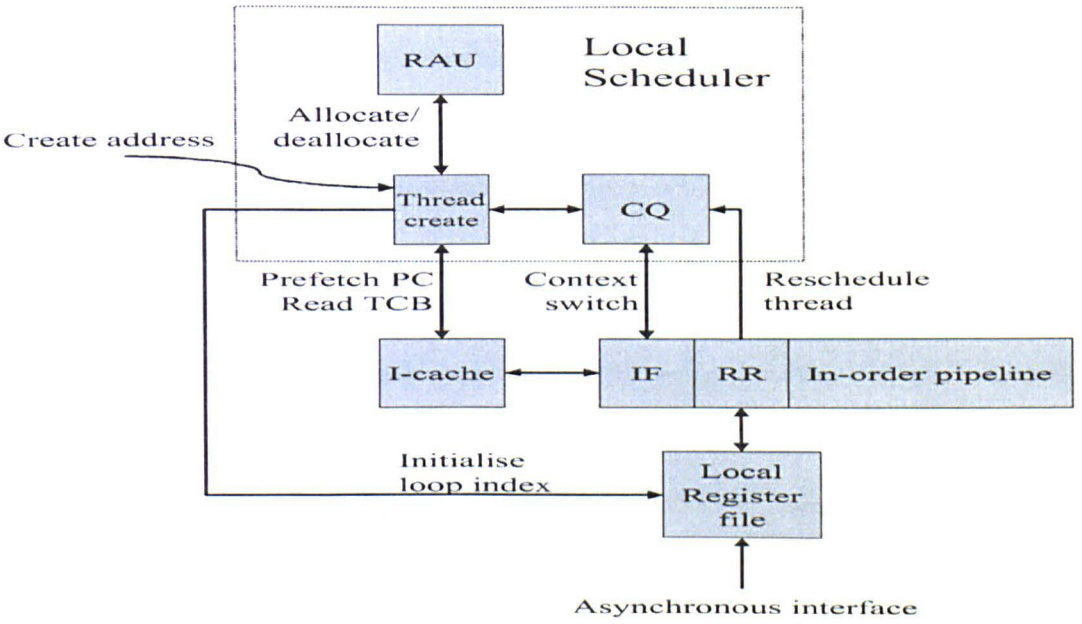


Figure 5.4: Detail of the local scheduler showing its main components and the data paths between it and other stages of the pipeline.

5.5 Thread Scheduling and Distribution to Support Microgrid CMP

5.5.1 The scheduler

A global scheduling algorithm determines the order in which a group of related threads is distributed to the processor array. This algorithm is built into the local schedulers and is controlled by the parameters from the TCB and the number of processors used to execute the family, which may both be dynamic. Within each processor, the local scheduler manages the execution of all microthreads currently allocated to that processor. The schedulers in different processors are independent and each manages a local model of its resource utilisation for the subset of the family of threads that it

must execute. This is based on the global scheduling algorithm and requires minimal communication between the processors (each processor must know the number of processors used and its location within that set).

As already described, microthreading exploits LLP by executing the same loop body for multiple instances of an index variable. It is also able to capture ILP within basic blocks. LLP is specified parametrically using loop bounds, with multiple iterations sharing the same code but using different microcontexts; this is SPMD concurrency. MIMD concurrency can also be specified using pointers to multiple code blocks but is static in extent, as the compiler must make the partition of the basic block and generate code fragments accordingly. Both are captured through the control instruction *Cre*, which initiates the creation of threads on all processors defined in a given profile. Each scheduler will continue to create threads, until its distribution of iterations has been exhausted. It may then continue to create threads from other families, whose *Cre* instructions may have been queued in the scheduler.

The process of thread creation requires the following actions (see figure 5.4):

- A slot number is obtained to address the scheduler's tables (CQ, in figure 5.4) from the scheduler's empty queue and the empty queue is updated.
- The RAU reserves the required number of registers for the microthread's context and returns a base address in the register file.
- The code pointer, the base address of the microcontext, and the base address and slot number of the microcontext it is dependent upon are all stored in the CQ slot.
- Finally, the index value associated with the microthread is written into the first

local register variable of its microcontext and all other variables are initialised to empty.

- The slot number is then passed to the I-cache to prefetch the first instruction, only after it has been prefetched, will the slot number be added to the active queue of the scheduler, where it is available for execution.

Figure 5.4 shows more detail of a local scheduler and its connections with the I-cache and the processor pipeline. The RAU within each scheduler models the allocation of micro-contexts to the local register file and determines when new microthreads may be allocated. If registers are available it will allocate a microcontext and then create entries in the CQ for each thread associated with that microcontext. When all the threads associated with a microcontext have been killed, its registers will be relinquished and the RAU will update its allocation model. In this way a scheduler can manage concurrency that is parametric and which exceeds the statically available resources.

5.5.2 Thread Distribution

One algorithm that can be used to distribute an iteration space to the array of processors is to use a block-cyclic distribution, which can be defined by the following equation, where iteration i, is mapped to processor q, using a profile of P processors and a block of b consecutive iterations allocated to each processor:

$$q = \left|\frac{i}{b}\right| mod P \tag{5.5.1}$$

In this schedule, b can be chosen to minimise inter-processor communication and

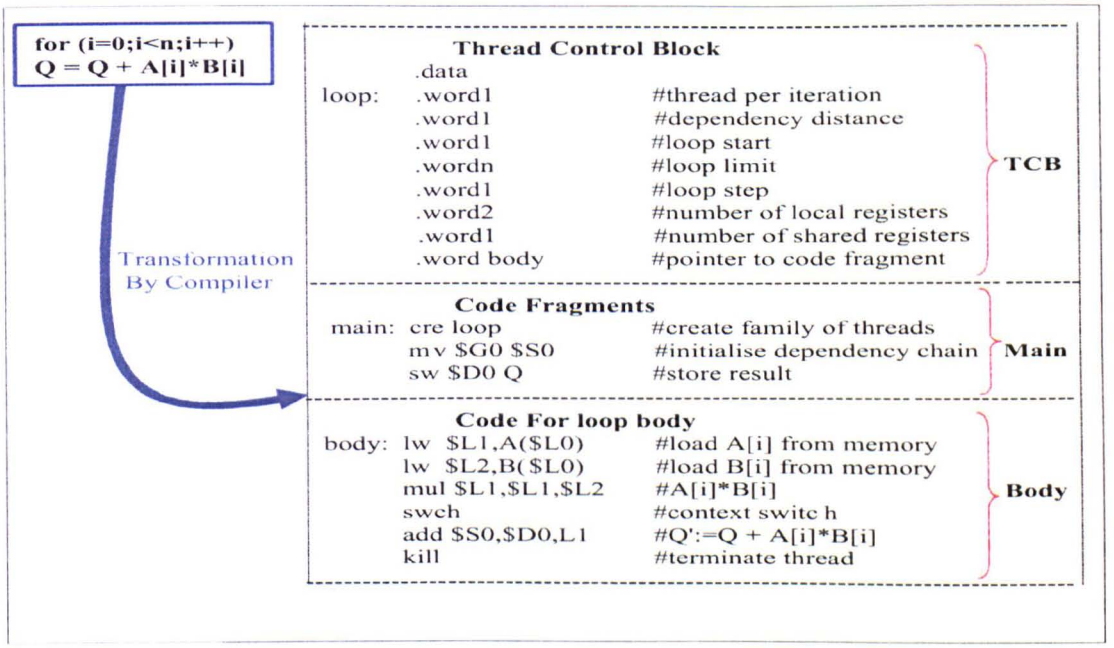


Figure 5.5: Transformation of the for loop to microthreaded assembly code.

ensure that regular inter-microcontext communication can be mapped to a point-to-point network, more specifically a ring network.

The process of thread creation and code generation will be illustrated using the following dependent loop:

```
for (i=0;i<n;i++)
  Q = Q + A[i]*B[i]
```

The loop has a dependency in the add operation between Q in the current iteration and Q′ from the previous iteration. The compiler generates code to carry this dependency between iterations using a register shared between two microthreads. This is specified by a dependency distance of 1 in the TCB, which is used to link dependent threads in the scheduler. The registers in a microcontext are divided into a local

part $Li, a shared part, $Si, and a dependent part, $Di, where the shared part of one microcontext maps to the dependent part of the iteration that is dependent upon it. Thus the assembly code shown in figure 5.5 uses $S0/$D0 to carry this dependency between iterations, where $S0 is written by the producer thread and $D0 is read by the consumer thread. The dependency chain is initialised and terminated in the main thread. In the assembly code, three parts can be identified. The first is the TCB, the second is the code for the main thread, which creates and synchronises this family, and the third part is the code for the loop body. Note that n iterations of this body execute concurrently between the *mv* and *sw* instructions in the main thread.

Looking at the concurrency in this code, it can be seen that all loads and multiplications can proceed concurrently but that the accumulation of Q in $S0/$D0 is constrained to execute in sequence and may be mapped to different processors. During the execution of this dependency chain, only one processor will be active while the result is accumulated. This constraint will limit speedup, but during the execution of the dependency chain, only the processor currently executing will be active and consuming power as all other processors will recognise an empty active queue.

This situation is easily detected and can be used for power management. When executing multiple iterations on one processor, the chain can be executed at one addition operation per cycle using the bypass network as a mechanism has been developed to reschedule threads in dependency chains predictively, i.e. where we know one thread must reschedule the next, e.g. on the add instruction for all threads. Information to detect this situation is given by the compiler in parameters to the switch and kill instructions. Those parameters signal the number of non-deterministic operands and whether to predictively reschedule the next interation in a dependency

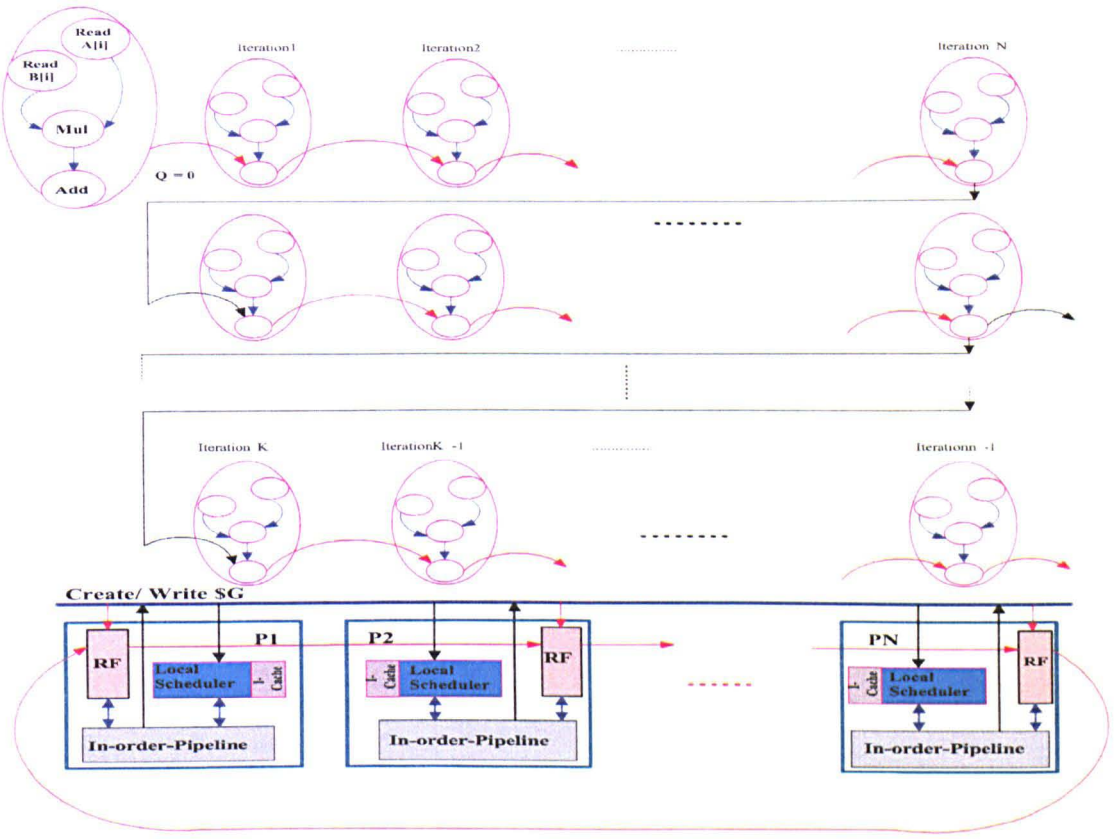


Figure 5.6: Modulo schedule of one iteration per processor for the example code in the text. This illustrates the mapping of a dependency chain to the ring network connecting processors in the Microgrid.

chain.

Figure 5.6 shows how the body of the loop is mapped to multiple processors connected with a ring network. It also shows the dependency chain between microcontexts mapped to different processors. Note that for simplicity, this particular schedule maps only one iteration per processor before moving to the next. Mapping multiple iterations per processor would minimise communication and maximise cache line locality.

5.6 I/O Service Routines

Input or output (I/O) routines on a microgrid CMP are managed by microthreads from a microcontext [24] running on dedicated processors. A collection of microcontexts that has low frequency tasks can be scheduled on a single processor, which provides a good utilisation of system resources and responds efficiently to external I/O events. All I/O routines in a microgrid CMP are directed to the registers, and wait there to be served by a microthreaded service routine. A very responsive fast and user-programmable on-chip network is also possible using this approach.

5.7 Microgrid CMP Scalability

The major advantage of the microgrid CMP is its scalability in terms of performance and power dissipation [24, 26]. Indeed, the method of decomposing a sequential program into microthreads, scheduling and allocating these microthreads dynamically and the efficient communication and synchronisation mechanisms are all factors in achieving scalability. The reduction in power comes from the hardware partitioning of the chip cores and from the distribution of the workload across a large portion of the chip. Processors with no active threads are aware that instructions can not be scheduled and can therefore go into standby mode dissipating minimal power. This power usage can be scaled with IPC rather than issue width. This conservative scheduling also provides an insight into asynchronous partitioning of a CMP. This modelled reduction in power dissipation is realistic and is further evidence of the scalability of the microgrid CMP.

However, scalability in terms of silicon area for microthreaded support structures

has not been considered before. In addition, it is necessary to model the top-level nature of the CQ and scheduling system, in order to verify and test the correctness of its behaviour. These issues, are demonstrated in chapter 7 in full detail, which shows a scalable implementation of microthreaded support structures, the feasibility of large-scale CMPs using emerging technology, and full simulation results for the top-level model of the CQ and scheduling system in VHDL.

5.8 Summary

The chapter presented the microgrid CMP architecture model and its buses. The distributed implementation of a microthreaded CMP includes two forms of asynchronous communication. The first is the broadcast bus, used for creating threads and distributing invariants. The second is the shared-register ring network used to perform communication between the register files in the producer and consumer threads. It is important to note that this action is totally decoupled from the pipeline operation through the use of explicit context switching. However, to avoid processor contention during bus access time, and to provide fairness in communication between processors, we need some form of arbiter. Also, the implementation of the bus interface between processors still not clear. In the next chapter, we discuss the implementation of these issues and we introduce a novel asynchronous arbiter optimised for this application.

It is shown that microthreaded CMPs use hardware scheduling and synchronisation and have structures to support this that are distributed, and have locality in communication wherever possible. This is achieved with distributed schedulers that jointly manage large parametric families of threads and a distributed register file that provides synchronisation and sharing of data between them. These structures

provide support for a shared-register, instruction-level model of concurrency in which synchronisation occurs between instructions and in the registers. However, the scheduler's structure is a most significant challenge, and the scalability of this structure is not yet clear and the area-performance has not been considered prior to this work. Therefore, chapter 7 provides an implementation and evaluation of microthreaded support structures and the feasibility of large-scale CMPs is investigated by giving a detailed area estimate of these structures. Moreover, the chapter provides full simulation results in VHDL of the CQ and scheduling system in order to verify their correct behaviour.

Chapter 6

Scalable and Partitionable Asynchronous Arbiter for Microgrid Chip Multiprocessor

6.1 Chapter Overview

In the previous chapter, it was shown that, when more than one processor requires access to the broadcast bus, an arbiter mechanism is required to determine either the order of request arrival or a request priority. In this chapter we discuss the design and implementation of a novel asynchronous arbiter optimised for this application. The arbiter has the advantage of asynchronous communication and uses a point-to-point connection between arbiter modules. A delay-insensitive methodology is used for our arbiter, allowing unbounded delays to both wires and logic gates.

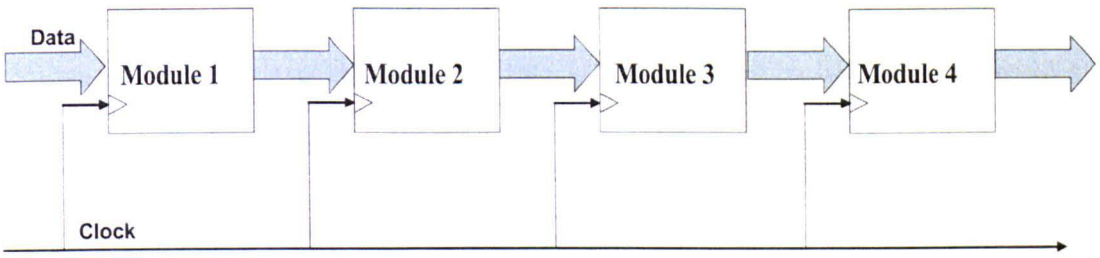
The outline of this chapter is as follows. In the next section, an overview of the asynchronous design methodology is given. Section 6.3 presents modern arbitration systems. The organisation, operations and design of the arbiter are presented in section 6.4. Implementation and simulation results for the proposed arbiter are given in section 6.5. Finally, a summary of the chapter is provided in section 6.6.

6.2 Asynchronous Design Methodology

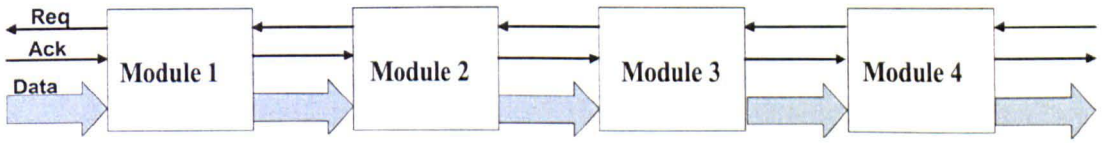
6.2.1 Asynchronous Design Procedures

Generally, circuit design styles can be classified either as *synchronous*, where the whole system is globally synchronous, or *asynchronous* with whole system being globally asynchronous. Figure 6.1 shows synchronous and asynchronous pipeline implementations, where the clock in figure 6.1a, is replaced by handshaking signals in figure 6.1b. Synchronous circuits may be simply defined under the control of a central clock. However, the centralised clock is one of the most significant challenges in modern synchronous systems. It restricts the system scalability and consumes a lot of power. An asynchronous approach eliminates the use of clock, has the advantage of better design modularity and opens the door wide for system scalability and functional partitioning, both of which are the requirements for future powerful and scalable designs. The Semiconductor Industry Association *(SIA)* Roadmap recognises that by 2007 asynchronous techniques will be used in many designs [129]. However, in simple asynchronous circuits the absence of a clock may result in hazards, for which one popular solution is the *Muller C-element* [130].

The design procedures for asynchronous design can be similar to that employed for synchronous machines. However, state diagrams for asynchronous circuits differ from those for synchronous circuits in that each stable state of the circuit must be include by a *sling* [131]. This means that the transition path originates and terminates at the same stable state. When a new input arrives, it changes the current state to the next state. The next state now becomes the current state, and a new input can arrive. Asynchronous circuit to be synthesised must expressed as a flow table [132], a form



a) Synchronous Connection



b) Asynchronous Connection

Figure 6.1: (a) Synchronous circuit. (b) Asynchronous circuit.

similar to a truth-table in synchronous design.

A general design procedure for asynchronous state machines can be summarised as follows:

- Create a state transition diagram or state table for the state machine that describes the required functional behaviour.
- Derive a primitive flow table from the state transition diagram. Note that only one stable state occurs in each row.
- Minimise the flow table reduction by merging rows (Remove any redundant states).
- Simplify the excitation table and obtain the output expressions using Karnaugh maps or other logic minimisation.

- Construct and simulate a circuit that implements these expressions.

Our asynchronous arbiter was synthesised using these design procedures and the details are explained later in this chapter.

6.2.2 Delay-insensitive Circuits

Delay modelling is one of the most significant elements of validating asynchronous design. One popular well-known approach that gives unbounded delays to both wire and gate elements is the delay-insensitive design approach. This design style avoids the need for the timing analysis, giving designs that operate correctly whatever the delay in the interconnecting wires [133]. It also has some benefits over bounded-delay methodologies in that the former delay model forces the designs to use conventions such as completion signals and transition signaling which are both important for good asynchronous circuit structures [134]. Furthermore, the delay-insensitive model allow the possibility of exploiting the average case delay rather than the worst case, which provides a significant saving with long interconnections [133]. There have been some processors that used a delay-insensitive technique such as described in [135, 136].

6.3 Modern Arbitration Systems

It is very well-known that accessing a shared resource with two or more processors requires an arbitration mechanism to prevent contentions and to insure that only one processor can access the shared resource at a time. Many arbitration schemes have been proposed [137, 138, 139, 140, 141] with different characteristics. Arbiters can be

centralised, decentralised, daisy chained, tree, round robin with fixed or dynamic priority, ring structure, etc. In fact, the degree of comparison between these mechanisms depends on a set of factors, such as: reusability, modularity, fairness in accessing the shared resource, avoiding starvation and minimising both power consumption and logic area. That is, most of the arbitration mechanisms are only suitable for some cases and none of them is optimal for all cases.

One popular arbitration priority scheme for distributed arbitration is the daisy chain mechanism [142]. In this mechanism all processors share the same bus request and bus busy lines, but a grant signal's propagated through all the processors (daisy chained). The priority in this mechanism is fixed and depends on the physical position of the processors within the chain [143]. Macii and Poncino [143] described a design of a scalable bus arbiter for a multiprocessor system using a ring architecture. This arbiter is synchronous in design and the priority level of each processor is reduced by one at every arbitration cycle to satisfy a rotating priority between the processors. Also, two signals (Bus_Busy and Token_Out) must be propagated through the ring network to circulate the token. Our arbiter also uses a ring structure but is a fully asynchronous design. It exploits the concurrency control instruction (Brk) provided by the microthreaded microprocessor model to hide the token circulation time and to set a priority processor based on the processor that has succeed in executing this instruction. Also one grant signal (Gout) rather than two is propagated to circulate the grant token around the ring.

Valencia et. al. [139] presents a modular asynchronous design for an n-user linear array arbiter; see figure 6.2. In this design a centralised control signal (C_0) is used to drive all the modules (M) in the array. When this control signal is 0, the arbitration

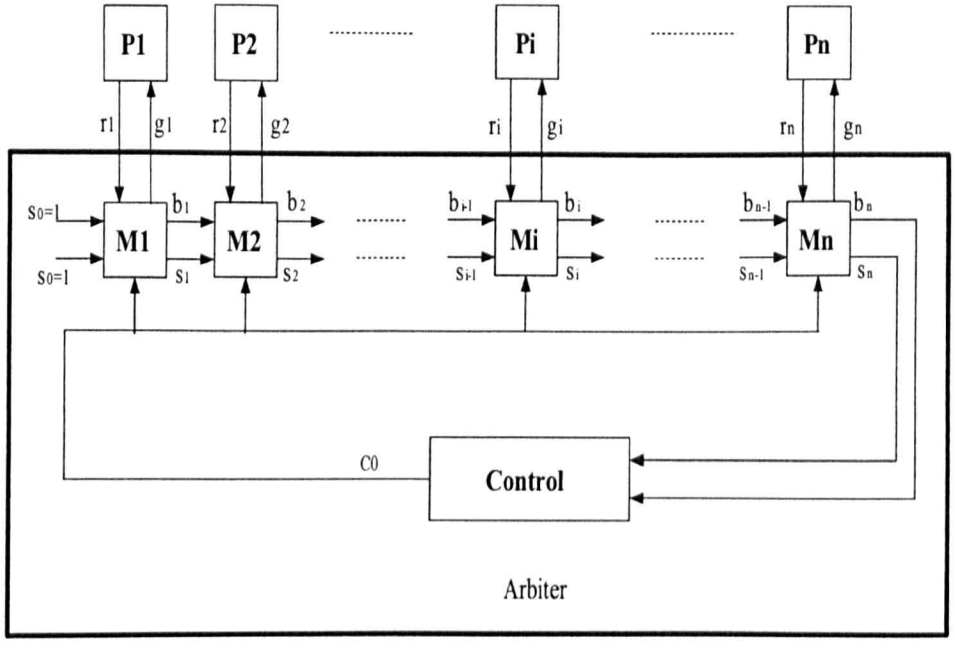


Figure 6.2: Organisation and signaling conventions for the arbiter (proposed in [139]).

process takes place in such a way that this signal is not 1 until the requests (r) have been granted (g) in the same order of the module in the array. Also, the priority policy in this arbiter is dependent on the relative position of the component modules. This arbitration mechanism is not fair and leads to a starvation situation if a large number of modules are used. Our arbiter has the advantage of being partitioned, where each arbiter can decide locally to access the global create bus or to wait. So, there is no need to propagate the control through all modules. Also, the priority policy described in our arbiter provides fair communication and avoids processor starvation. It also, hides the token circulation time by moving the token to the most likely processor to issue the create instruction, which enable one processor to create a new family of microthreads.

Moore et.al. [144] proposed an asynchronous-synchronous interface design for

point-to-point channel communication with independent clock domains. The authors suggested a new scheme by adding an asynchronous FIFO between the producer and consumer modules to hide the waiting time during the request and acknowledge synchronisation. From a hardware point of view, adding extra components means increased complexity. This mechanism requires a complex control scheme, and in some cases, if the FIFO is deeper, the performance will be significantly degraded.

Work done in [141] describes the design of asynchronous arbiters for on-chip communication systems. The authors proposed both fixed and dynamic priority arbiter configurations. In the fixed priority design, three blocks are used to handle the arbitration mechanism. These blocks are the loop control block to reactivate the arbiter after serving requests, the synchroniser block to sample the input requests and the fixed-priority block to determine the priority value based on a hardware coded priority mechanism. The dynamic priority design also has the same complexity of blocks, where n request-analyser blocks and n priority-comparator blocks are required to handle n requests. This arbiter has a complex arbitration design with a centralised structure which prevents partitioning. Also, many comparisons may be required to determine the priority values if the previous comparison failed in determining the priority value.

In contrast, our arbiter has less complexity and provides a simple arbitration mechanism for a large number of processors. It also provides a simple mechanism to pre-detect the priority through a concurrency control instruction provided by the microthreaded microprocessor model to move the token to the most likely processor to issue the create instruction. Also, the arbiter we describe has the advantage of a partitioned design and this issue will be explained later in this chapter.

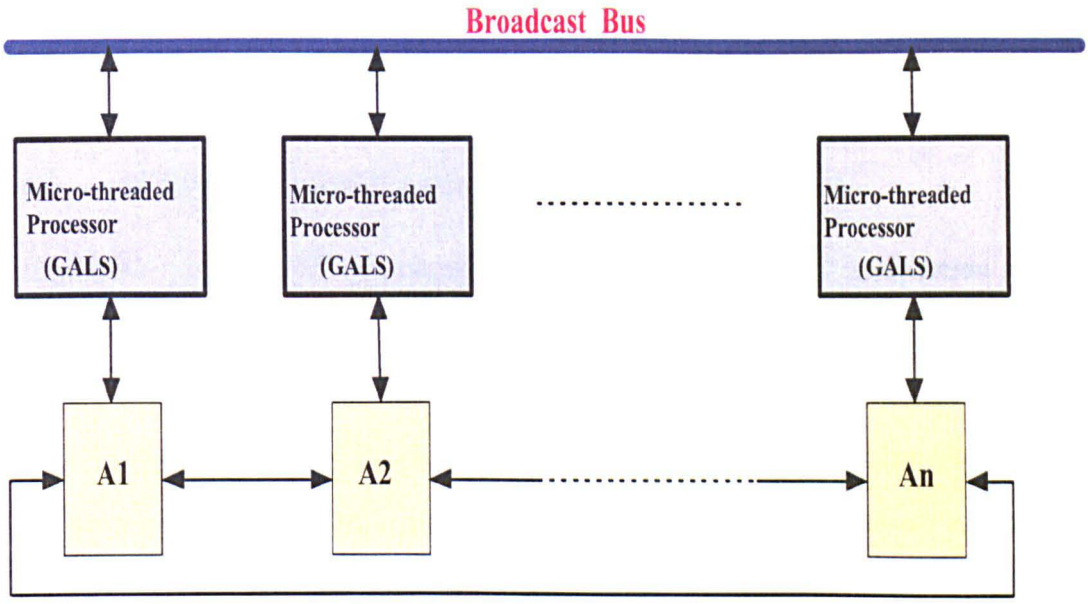


Figure 6.3: Asynchronous arbiter block diagram.

Villiger et.al. [126] proposed a mechanism for transferring data between GALS modules using a self-timed ring topology. This configuration provides a point-to-point communication between two adjacent GALS modules and provides a modular connectivity, which has full scalability in both bandwidth and area with an increasing number of GALS modules. The design we described in this chapter has the advantage of a ring organisation that connects GALS microthreaded processors with the broadcast bus in a circular fashion.

6.4 Asynchronous Arbiter for Microgrid Chip Multiprocessor

6.4.1 Arbiter Organisation and Bus Interface

As described in the previous chapter, microgrid CMP has two subsystems requiring global communication i.e. the broadcast bus, and the arbiter ring network, and both use asynchronous signals, creating independent clocking domains for each processor. The arbiter exploits the advantage of a concurrency control instruction *(Brk)* provided by the microthreaded microprocessor model to set the priority processor and move the circulated arbitration token to the most likely processor to issue the create instruction. This mechanism provides a latency hiding of the token circulation time by decoupling the microthreaded processor from the ring's timing.

Figure 6.3 shows the novel arbiter organisation. Each processor has its own local control and a separate arbiter module in order to allow processor partitioning. Each arbiter module is linked to the next one in a ring arrangement and the processors are arranged in a grid layout as shown in figure 6.4a. Thus each arbiter can be linked to two physically adjacent ones to reduce propagation delays. Our arbiter has the optional capability of being usable in a dynamically partitionable processor array, assuming a suitable routing architecture is available. For example, a possible reconfiguration of the processors in figure 6.4a onto two independent groups is also shown in figure 6.4b.

Figure 6.5 shows the arbiter input and output signals. As shown, the arbiters are linked by four lines comprising the request high (RH_i), which is the highest priority request, request low (RL_i), which is the lowest priority request, an acknowledgement

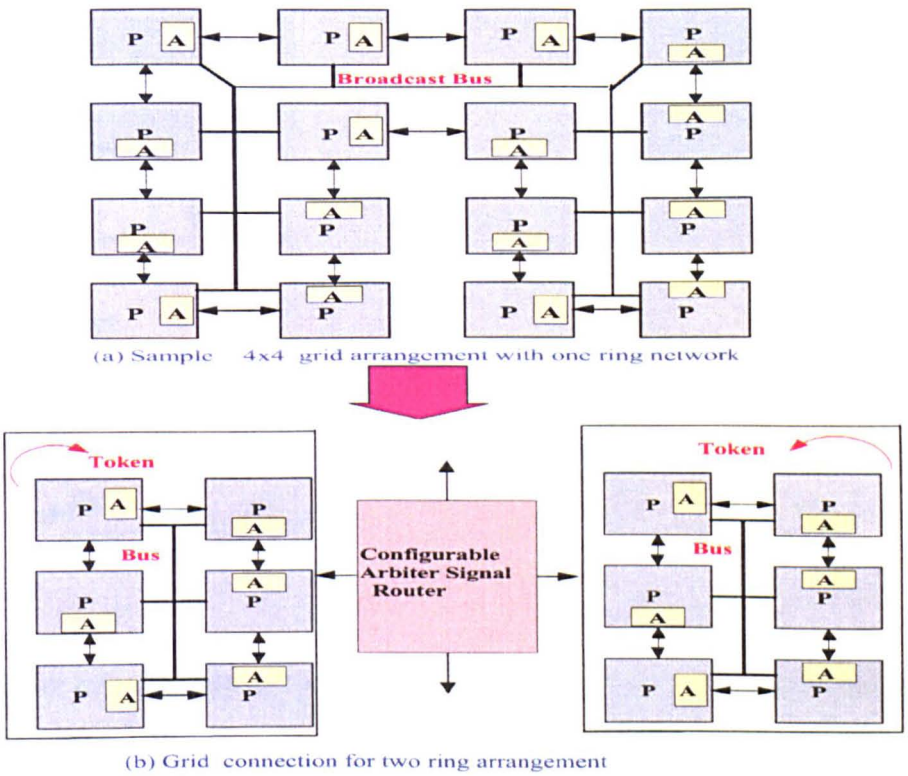


Figure 6.4: Asynchronous arbiters with different partitioning. a) Grid organisation. b) Independent group organisation.

signal (Ack_i) to release the bus, and the grant line (G_i) to grant requests and move the grant token towards the requesting module. The request and grant signals propagate in opposite directions around the ring. Also, one output wire ($Wout_i$) is required from each arbiter module to give processor P_i permission to access the broadcast bus.

There are three signals from each processor to its arbiter. The first is to inform the arbiter that the current processor has succeeded in executing the Brk_i instruction, the next signal (D_i) is used to assert a demand request. The third is the local acknowledgement ($Ackl_i$) signal to inform the arbiter that a receiving processor has finished reading the data from the bus. Note that within the arbiter the Brk_i signal

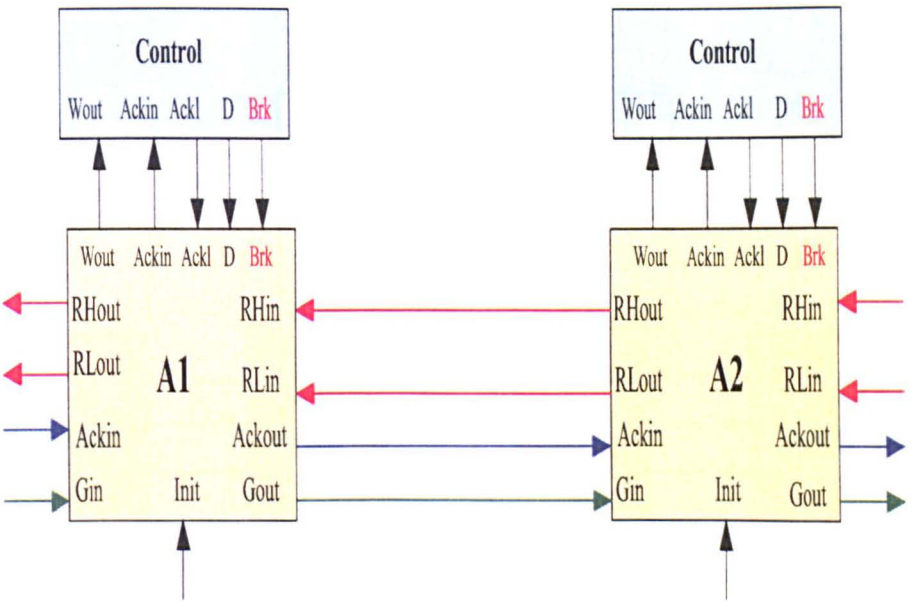


Figure 6.5: Asynchronous arbiter with require input and output signals.

wire is assigned to the RH_i signal line with highest priority and the D_i signal assigned to RL_i line with low priority. Note that an initial (init) signal is also required to determine the initial location of the token. One arbiter is initialised with the token, the others without.

In order to release the bus a processor must receive an acknowledgement signal. To get that, every processor has to signal it has read the data, therefore we can return the acknowledgement signal back to the grantee by using the same ring connectivity to propagate the acknowledgement back until it reaches the processor that has currently reserved the broadcast bus. The required acknowledgment control circuit is shown in figure 6.6, where each processor asserts a high signal through its local acknowledgment ($ACKl_i$) line when that processor has read the data from the bus. A write (WR) signal is also required to control the propagation of the acknowledgment signal

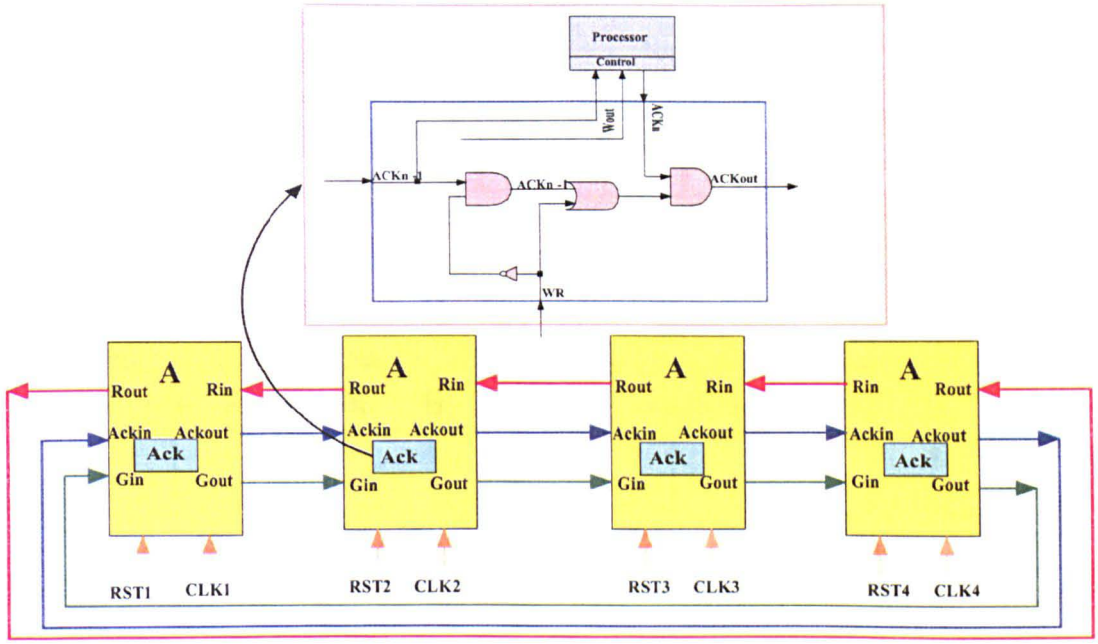


Figure 6.6: Released control circuit.

through the arbiter chain. Thus, the acknowledgement signal is propagated from one module to another until it reaches the processor that has reserved the broadcast bus. When that processor receives an input acknowledgment ($Ackin_{i-1}$) signal from the previous arbiter module the processor releases the token and the arbiter responds by deasserting Wout.

6.4.2 The Proposed Arbitration Mechanism

The arbiter provides a very simple arbitration mechanism, where each module has a few wires connecting to the next one and the last is linked to the first module in a circular fashion; see figure 6.7. Thus, as soon as the $Wout_i$ signal arrives at the corresponding processor, the processor sends its data through the broadcast bus and waits for the acknowledgement signal. This signal informs that the data is arrived at

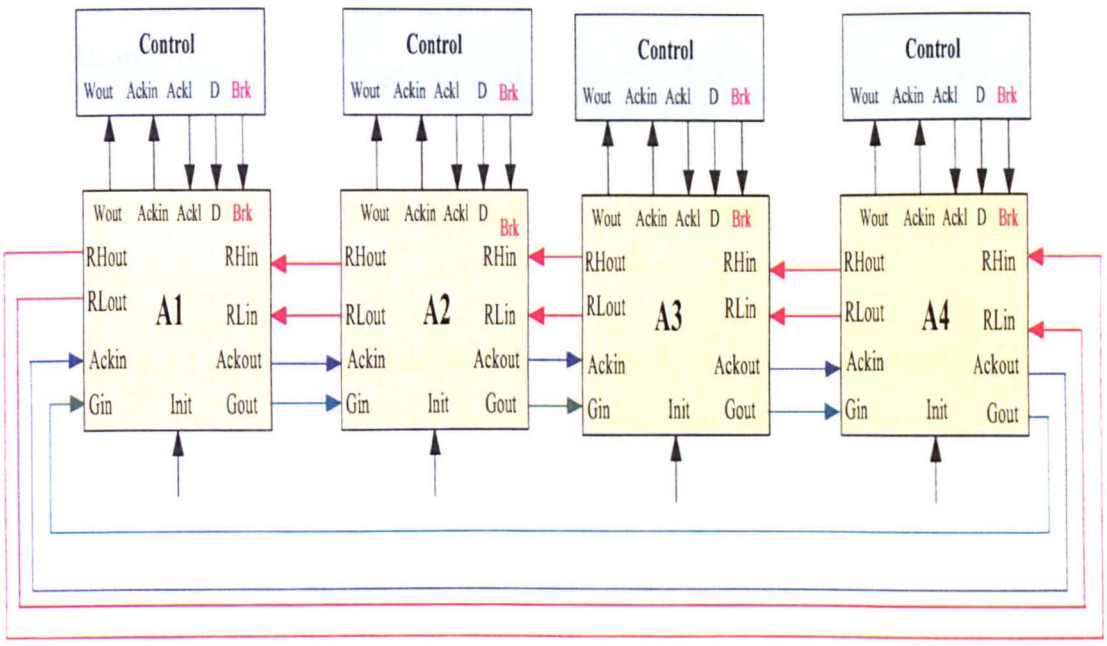


Figure 6.7: Arbiter modules with required signals connected as a ring configuration.

the required destination successfully.

The arbiters operations can be described as follows, where we have N arbiter modules and only one processor can succeed in executing the Brk instruction at a given time.

- The arbiter is labelled using modulo arithmetic so for M arbiters A_{i+1} is A_0 for $i = M - 1$ and A_{i-1} is A_{m-1} for M=1.

- Note that $init1 = 1$ and $init2$ to $init_m = 0$. This means that processor 1 would have a request acknowledged immediately after system initialisation (reset) but other processors must wait for the grant to propagate ($A_1 to A_2 ..... to A_m$).

- If Brk_i =1, A_i outputs a high request to the next arbiter via $RHout_i$. The rest of the modules can also generate a demand request via $RLout_k$ where k can be

any number from 1..N except i ($k \neq i$). If all Brk=0 any module can assert RLout.

- If Brk_i=0 and D_i=0, A_i propagates $RHin_i$ to $RHout_i$, $RLin_i$ to $RLout_i$ and Gin_i to $Gout_i$. This propagates RH_i and RL_i from A_i to A_{i-1} and G_i from A_i to A_{i+1}.

- If Brk_i=1 and Gin_i=1, and $Ackin_i$=0 then A_i asserts $Wout_i$ (read), which gives the processor permission to access the broadcast bus.

- When a receiving processor has completed the bus transaction it asserts a local acknowledge signal $Ackl_i$=1, which is also propagated through the ring until it reaches the module that has currently reserved the bus. Thus, when $Ackin_i$=1 and $Wout_i$=1, the token is released and the arbiter responds by deasserting Wout.

- If Brk_i=0, and the input line $RHin_i$=1 , then forward the grant to the next module irrespective of D. If D_i=1 assert $RLout_i$=1, else propagate $RLin_i$ to $RLout_i$.

- If Brk_i=0, and input line $RHin_i$=0, and demand request D_i=1 and $Ackin_i$=0, then activate the $Wout_i$, which gives the processor permission to access the broadcast bus.

- If Brk_i=0, and Gin_i=1 and $RHin_i$=0, and demand request D_i=0, and $RLin_i$=1, then forward the grant to the next module.

- When there is no request from any processor, then the RH_i, RL_i, G_i, Ack_i, and $Wout_i$ will all be 0.

6.4.3 Priority Policy

As described above, the arbiter exploits the concurrency control instruction provided by the microthreaded microprocessor model to set a priority policy based on the processor that it has succeeded in executing the Brk instruction, instead of just assigning the priority based on the position of the processor in the chain as described in [139]. Note that the microthreaded pipeline executes the Brk instruction before executing the Cre instruction, which provides latency hiding during grant token circulation time.

It is important to know the average number of arbiter modules in one ring, that will keep the latency of the token movement hidden. To do this, it is useful first to know the worst and best case for the number of arbiter modules. If we assume that the request time between two adjacent arbiter modules is given by t_m, and the time interval between executing Brk and Cre instructions is given by t_{brk}, then the best case for the number of arbiter modules in one ring that keeps the latency hidden (constant time) during token movement is 2 and the worst case (N_w) is given by the following equation.

$$N_w = \frac{t_{brk}}{2t_m} \tag{6.4.1}$$

If we take the average for the best case and the worst case, then the average number of arbiter modules (N_{av}) in one ring can be given by the following equation.

$$N_{av} = \frac{t_{brk}}{4t_m} + 1 \tag{6.4.2}$$

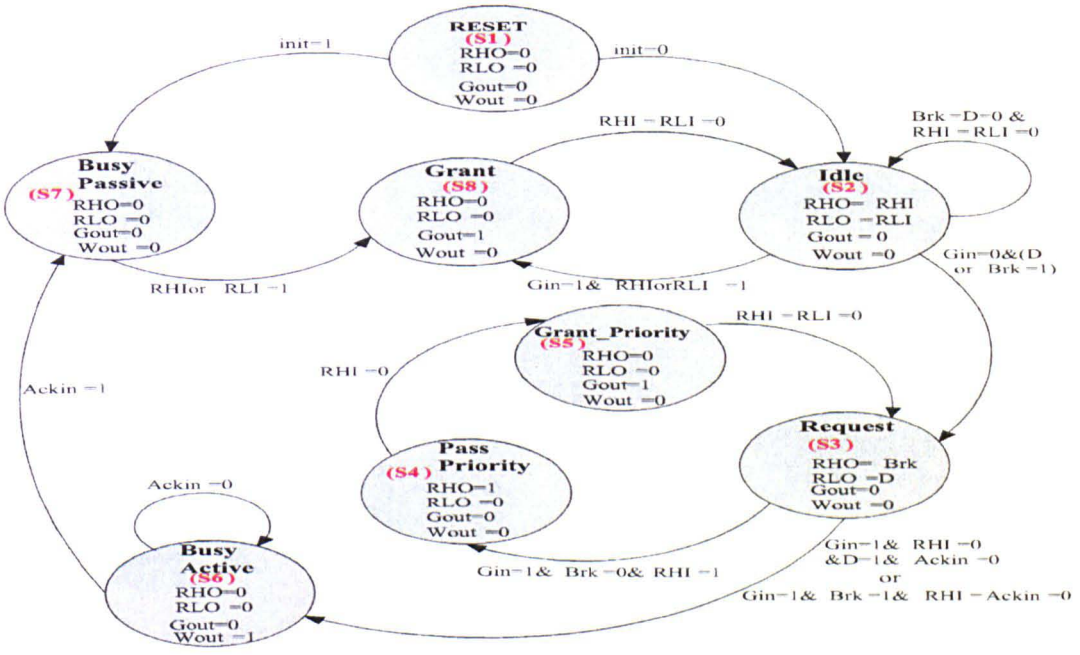


Figure 6.8: Arbiter state transition diagram.

The cycle time of our arbiter configuration, T_c, is the time required to move the token around the ring. This time is linear if t_{brk} time is greater than or equal to the time require to move the token around the ring. Otherwise, the time start increases. This issue can be expressed as follows.

$$T_c = \left\{ \begin{array}{ll} Constant & \text{if } t_{brk} \geq 2N_{av}t_m, \\ 2N_{av}t_m & \text{if } t_{brk} < 2N_{av}t_m. \end{array} \right\}$$

Two levels of priority have been introduced in this design, high and low priority. The high priority is given to the processor that has succeeded in executing the Brk instruction, while the low priority is assigned to a processor that has activated a demand request. Note that with the current microthreaded CMP model; only one processor can succeed in executing the Brk instruction at a given time, which means there is no need for many levels of priority. However, the mechanism we described

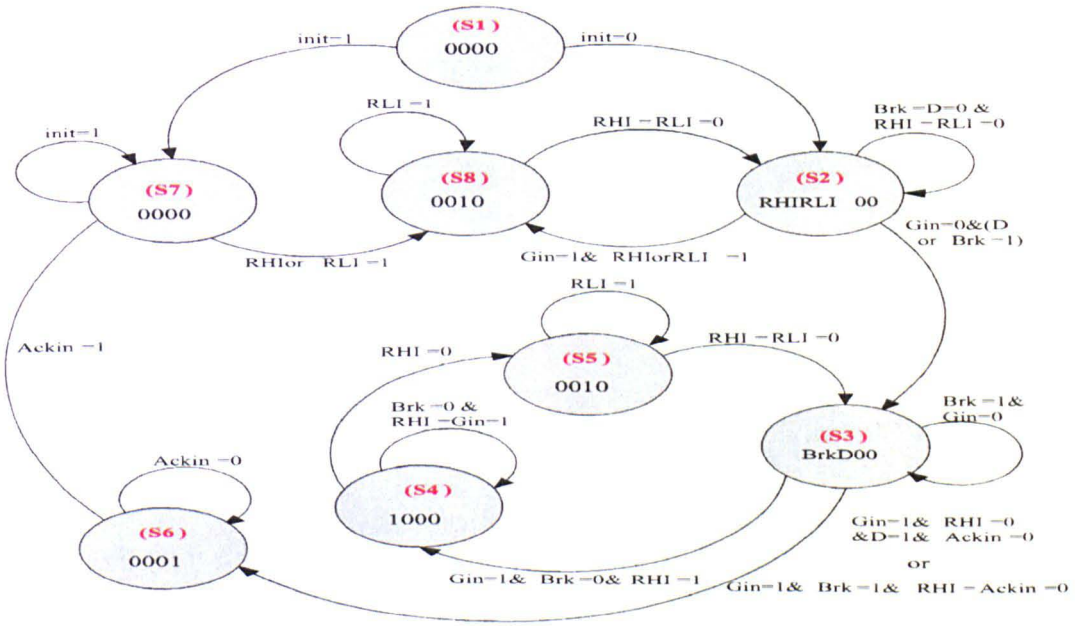


Figure 6.9: Asynchronous version from the arbiter state transition diagram showing sling on each stable state.

can be easily extended for many levels of priority and can be used to support any CMP arbitration model.

Thus, as described above the high priority is given first to the module that has succeeded in executing the Brk instruction, then the rest of the modules that have requested the bus are served based on their position in the ring and in sequence order. This mechanism provides fairness and is starvation free. As soon as the processor releases the bus the next module will be served directly.

6.4.4 Arbiter Design Methodology

The state machine diagram for the arbiter module is shown in figure 6.8. There are eight states; however an asynchronous version of this machine can be minimised. Two

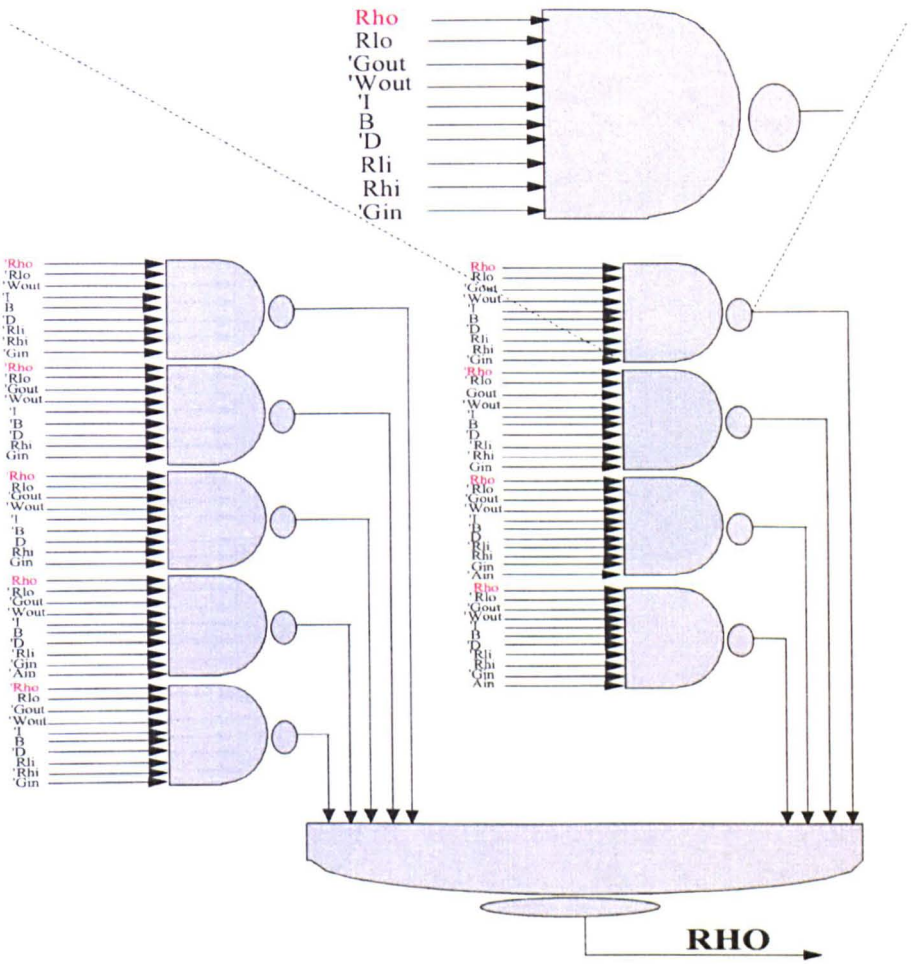


Figure 6.10: Arbiter level gate design (request high output (RHO)).

states, reset and grant priority, can be eliminated by merging states; see appendix C1 for details. It is important to note that each stable state of the state transition diagram must be represented by a *sling* i.e. a transition path originating and terminating at the same stable state; see figure 6.9.

The idle state receives the input requests from $RHin_i or RLin_i$ and if there is no input grant $Gin_i = 0$, it propagates the input requests to the next arbiter module via output request lines $RHout_i or RLout_i$. The request must be propagated until it

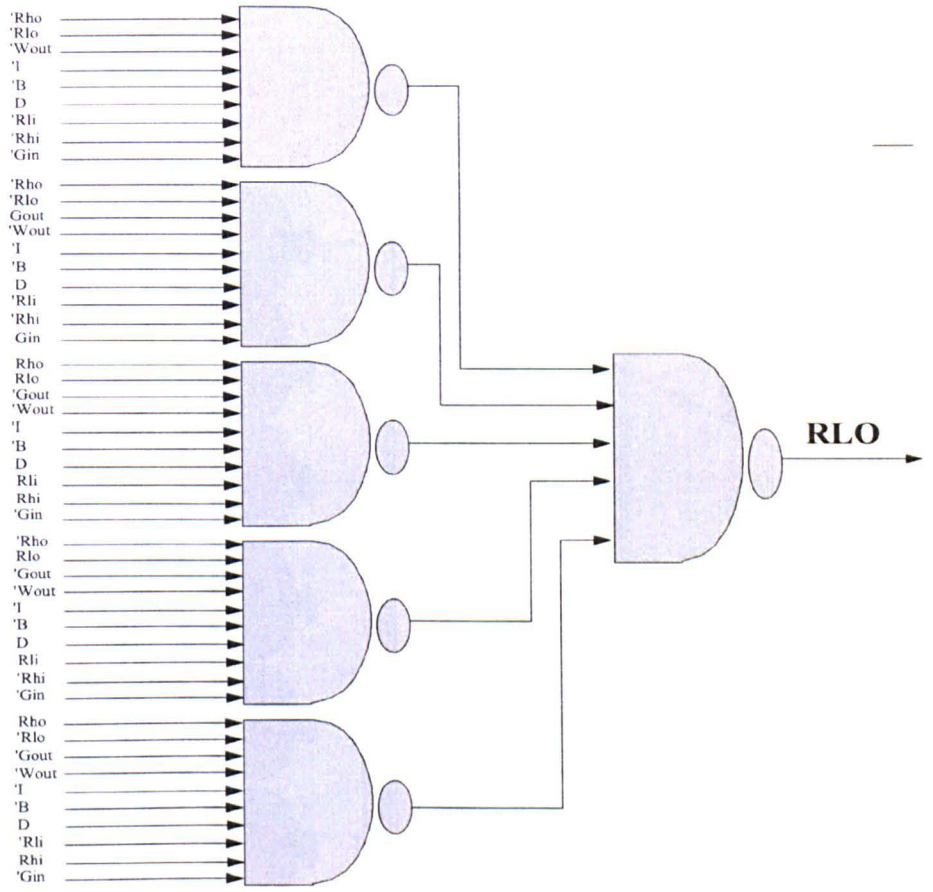


Figure 6.11: Arbiter level gate design (request low output (RLO)).

reaches the module that currently holds the token. The token is stored in the busy passive state, from which a high input request from RHi or RLi cause a change to the grant state. In the grant state the machine waits for removal of the incoming request before returning to the idle state.

From the idle state an incoming bus demand from the processor (D=1 or Brk=1) causes a change to the request state. In the request state, if the input grant $Gin_i = 1$, and $Brk_i = 1$, and $(Ackin_{i-1} = 0)$, then the state changes to busy active, which gives the processor permission to access the broadcast bus by activating the $Wout_i$ line.

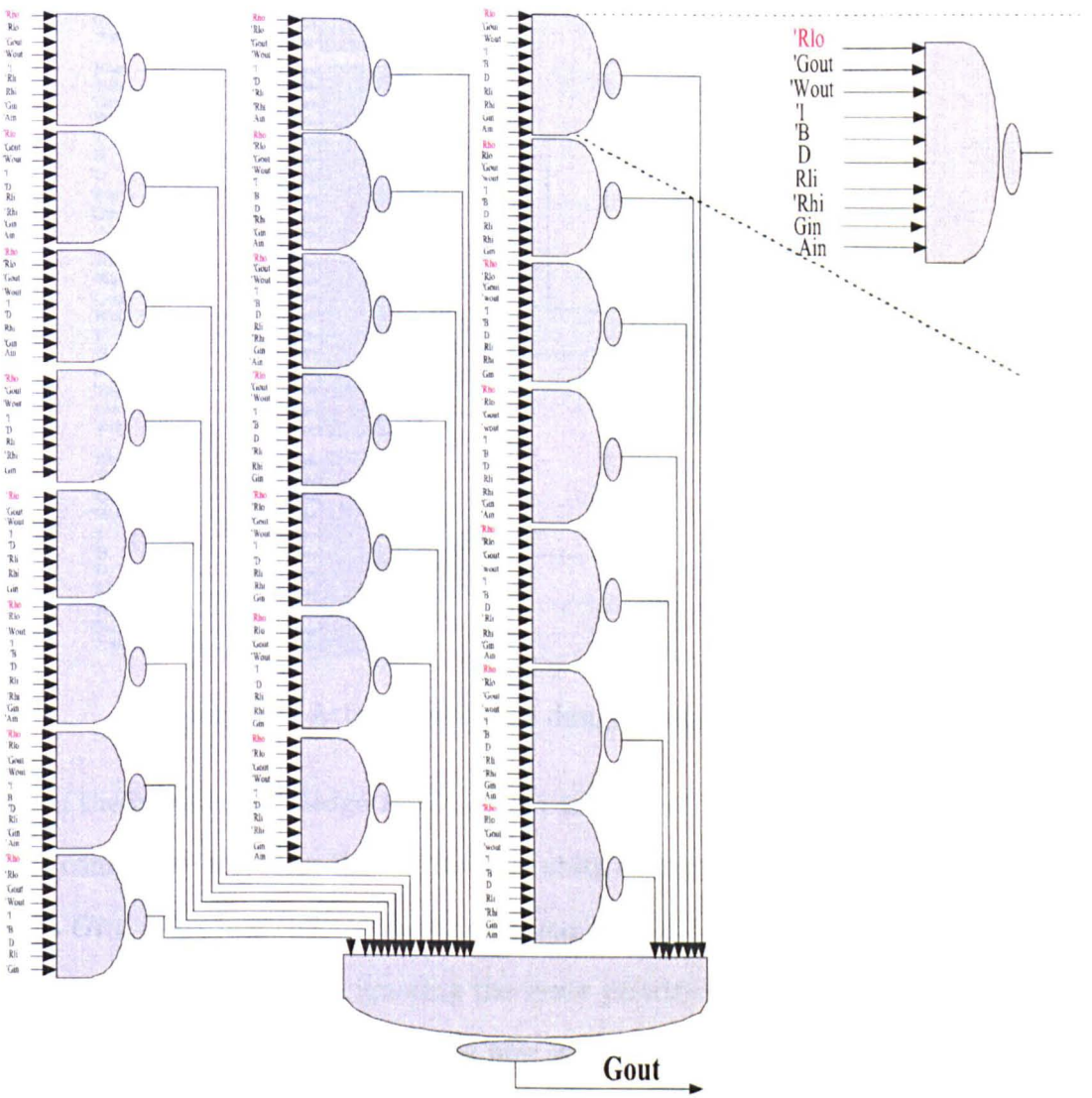


Figure 6.12: Arbiter level gate design (grant output signal (Gout)).

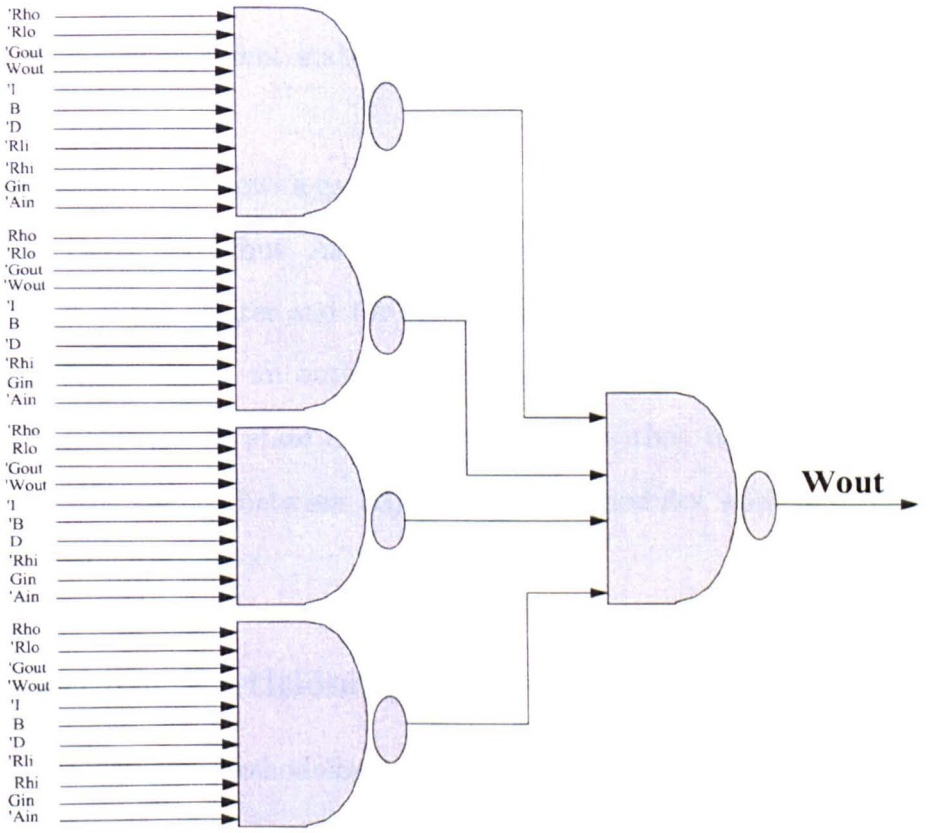


Figure 6.13: Arbiter level gate design (output signal (Wout)).

When the input acknowledge $Ackin_{i-1} = 1$ is received, this means that all processors have completed accessing the bus and the state changes to busy passive. If the input grant $Gin_i = 1$ while $Brk_i = 0$ and input request Rh_i=1, then the pass priority state is used to pass the request, ignoring the lower priority demand from this processor.

The permeative flow table can be now derived from the state transition diagram described above. The total number of inputs in this state machine is 7, and the present state requires four bits. Thus, the number of inputs is large, and of course this requires both a large flow table and a large K-map. The permeative flow table, the flow table reduction by merging rows and the simplified functions for each of the

output expressions are described in full in the appendix C1. Note that the circled states in each row represent stable states, while dashed lines represent don't care state.

Figures 6.10 to 6.13 shows a gate level design of our arbiter. There are four outputs RHO, RLO, Gout, and Wout. As shown, a simple logic gate is required to implement the behaviour of the arbiter and the arbiter depends on both the input signals and the current state. Thus, an output only fires when its appropriate inputs become available. Otherwise its state change occurs. Note that the inputs are triggered directly and the signals between adjacent arbiter modules work as a handshaking signals.

6.4.5 Arbiter Partitioning

A partitionable design methodology will become one of the design requirements that ensures low power and high performance in future processors [69, 145, 146]. It is one of the most important design issues, which is effective in block design and system verification [147]. This feature makes the design more flexible and provides a point-to-point communication between adjacent modules. Point-to-point communication in the GALS design approach provides low power and high performance [144]. It also offers a promising approach to fault tolerance problems and provides an independent communication between different system blocks.

As previously described, each arbiter connects to two other arbiters associated with adjacent processors to form an arbitration ring as shown in fig 6.4a. This arrangement could be hardwired, however by providing a routing architecture as shown in figure 6.14 reconfiguration of processors and their buses can be achieved. So, for

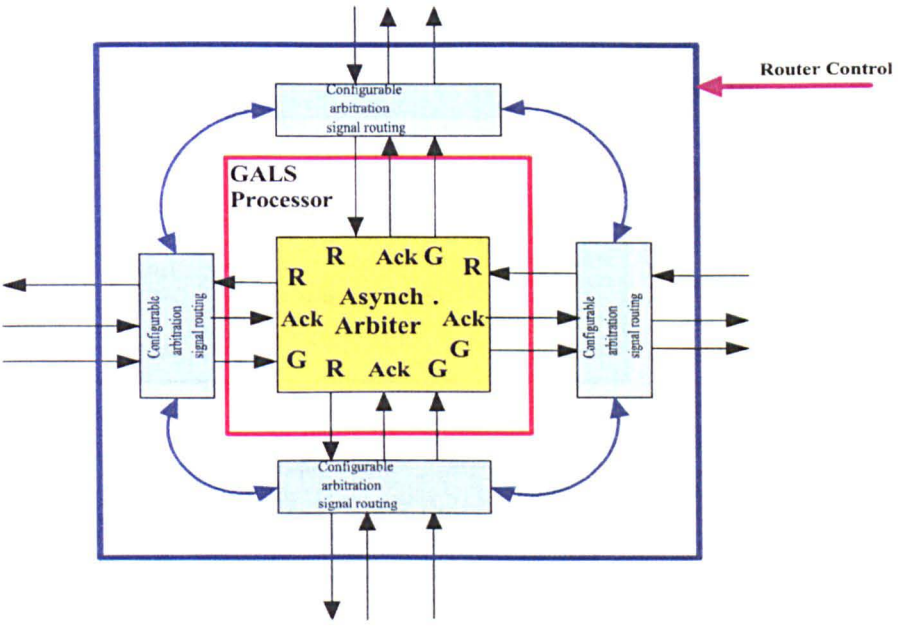


Figure 6.14: Asynchronous arbiter with programmable routing for partitionable processor arrays.

each arbiter and their associated global resources the processors can be partitioned into groups, where each group has a separate token.

6.4.6 Arbiter with N-levels of Priority

Figure 6.15 shows a block diagram for a scalable asynchronous arbiter design with n-levels of priority. As illustrated, three blocks are required to handle n requests, which comprise the processor bus access controller block, a request logic block and the state machine block. The function of the first block is to control and manipulate different levels of priority, where the priority levels can be determined by the compiler.

The second block determines whether the demand input signal has a high or low priority compared with the incoming requests. Thus if the demand line D has low

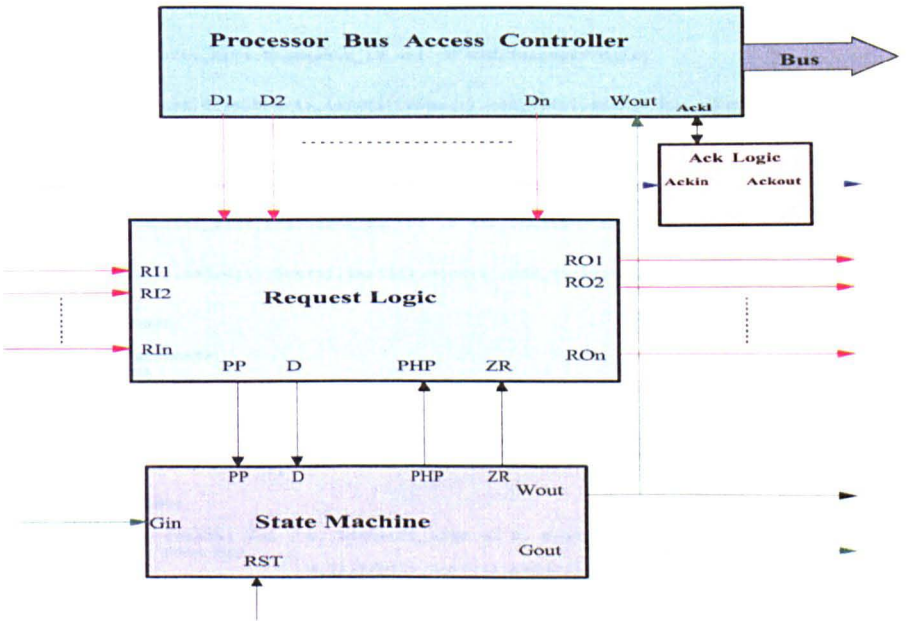


Figure 6.15: A block diagram for a scalable asynchronous arbiter design.

priority, then a high signal is asserted to the state machine through PP wire line. Otherwise, if the demand has high priority, then PP=0 is asserted.

The state machine uses the input signals from the request logic block to decide whether to pass the grant line to the next module via Gout if the current module has a lowest priority; or to activate the Wout line, which allows the processor to access the bus. So, if the current module has the high priority, then the pass-high-priority (PHP) signal is activated by the state machine to inform the request logic block that the bus access is given to the current module. Otherwise PHP=0 is asserted. The zero request line (ZR) can be used to control all output request RO lines, which block the propagation of output requests RO_i if ZR=0, or to pass the request to the next module if ZR=1.

```
   cpu0 : CPU
 generic map  (
                w,Transfer_size,Processor_id =>i ,N =>N,Tdelay=>Tdelay
              )
 port map     (
              phil(i),rst,d(i),Brk(i),init(i),wout(i),add_st(i),addr,Acklocal(i), Acq(N),WRREQ
              );
end generate input0;

 inputn: if (i>0) generate
   cpu1 : CPU
 generic map  (
                w,Transfer_size,Processor_id =>i ,N =>N,Tdelay=>Tdelay
              )
 port map     (
                phil(i),rst,d(i),Brk(i),init(i),wout(i),add_st(i),addr,Acklocal(i),Acq(i-1), WRREQ
              );
end generate inputn;
end generate microproc;

A1:for i in 0 to N generate
P0: if i = 0 generate
processor0: Arbiter
                        generic map ( w, transfer_size => 8, processor_id => i,N=>N, Tdelay => Tdelay)
                        port map    (
                                     d(i),Brk(i),init(i),reqh(i),reql(i), grant(N),Wout(i),reqh(N),
                                     reql(N),grant(i),Acq(N),Acq(i),WRREQ,Acklocal(i)
                                    );
end generate P0 ;
Pn: if (I > 0 ) generate
processorn: Arbiter
                        generic map ( w, transfer_size => 8, processor_id => i,N=>N, Tdelay =>Tdelay)
                        port map    (
                                     d(i),Brk(i),init(i),reqh(i),reql(i),grant(i-1),Wout(i), reqh(i-1)
                                     ,reql(i-1),grant(i),Acq(i-1),Acq(i),WRREQ,Acklocal(i)
                                    );
end generate Pn;
end generate A1;
```

Figure 6.16: Arbiter test bench source code.

6.5 Implementation and Simulation Results

We simulated the arbiter using VHDL, exploiting the generate statement to create networks of N processors/arbiters in the test bench. A snapshot from the arbiter test bench is shown in figure 6.16 and a full VHDL source code for the arbiter components are presented in appendix C2 and C3 respectively. The simulations used processors with different clock phases and frequencies in order to model their globally asynchronous nature. The arbiter modules were linked using arbitrary delay elements as shown in figure 6.17 to model interconnect delays.

The delay insensitive model uses unbounded delays on wires and gate elements and is a suitable method for analysation of transition-based signaling. Therefore, no matter how long the arbiter module waits for input changes when the arbiter sees

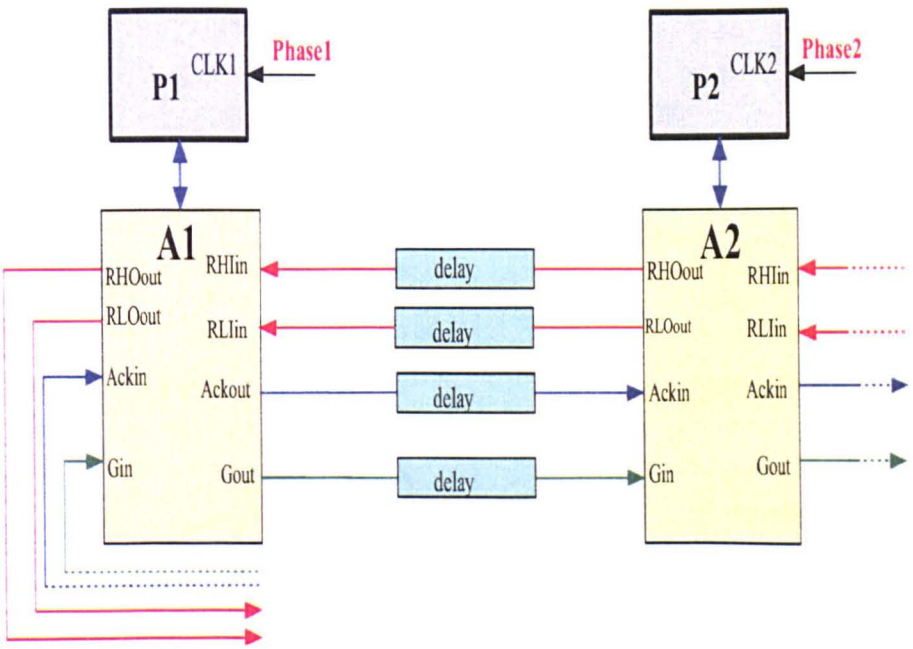


Figure 6.17: Asynchronous arbiter simulation model.

activation of the input signals, the transition is passed to the next module which will eventually know that new input values have arrived. Simulations using this approach verified correct operation of the arbiter with up to 64 processors. The processors were modelled using a high level description of the CQ and scheduling system, which will be reported in chapter 7. In effect, the sequencing of bus requests in these simulations were manually controlled by the test bench set up.

We investigated the performance of our arbiter with respect to the request-to-grant delay by replacing the processor model with a simple state machine and generated requests at delays determined by a sequence of random numbers. The state machine is shown in figure 6.18.

The state machine first generates a request (*local* state) through a demand line (D) then changes to the *wait bus* state. When a grant is received the state changes

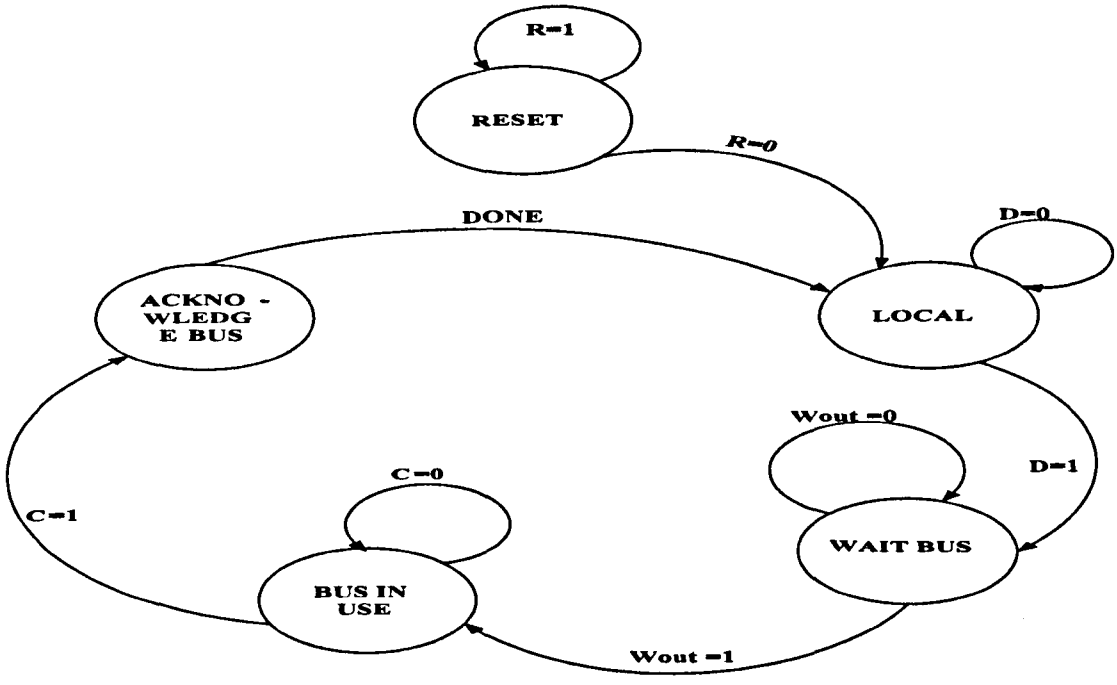


Figure 6.18: Processor state machine.

to *bus in use*. When the bus access is complete (C) the state changes to *acknowledge* informing the rest of the processors in the ring that bus is free again. The simulations used different numbers of processors i.e. 4, 8, 16, and 32. Figure 6.19 shows a linear result for the Request-to-grant delay with rate of requests (per processor per cycle).

As discussed in the previous chapter, the rate of requests to the arbiter within the context of the microthreaded CMP depends on the behaviour of the create instruction. The frequency of executing this instruction over a range of loop kernels is very low (17%) over all loop kernels considered in this analysis. Thus the bus is used infrequently, there is very little or no contention, so the delay in arbitration will primarily depend on the ring delay. Furthermore, microthreaded processors are tolerant to latency when they have created threads, so it does not matter how long it takes to create the next family of microthreads.

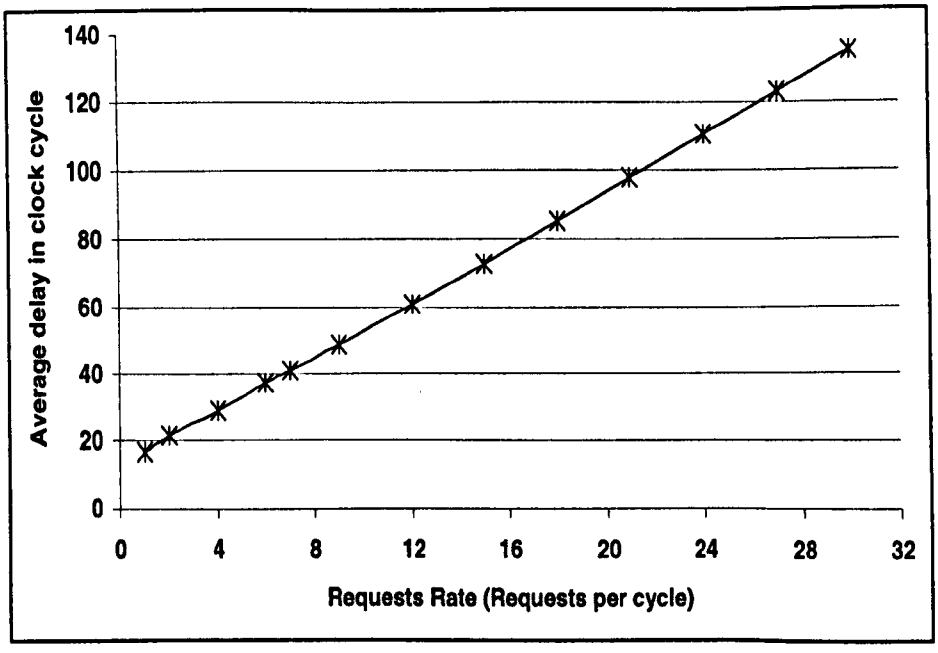


Figure 6.19: Request-to-grant delay with rate of requests (per processor per cycle).

Figures 6.20 and 6.21 shows a sample of results from simulating 8 arbiter modules. In this sample the following conditions apply: module 0 has initially reserved the token, module 7 receives a high input on the *Brk* signal line and modules 1, 2, 3, 4, 5, and 6 have high input demand request lines. As illustrated, the request signal RL_1 reaches the token before RH_7, which means that broadcast bus access is given first to processor 1 (Wout is asserted). When processor 1 releases the token, the grant signals are propagated back to give processor 7 permission to use the broadcast bus before other lower priority processors. The rest of the demand requests are granted in sequence order and based on their position in the ring configuration. More simulation results with different sizes of arbiters and different demands and brks scenarios are provided in appendix C4.

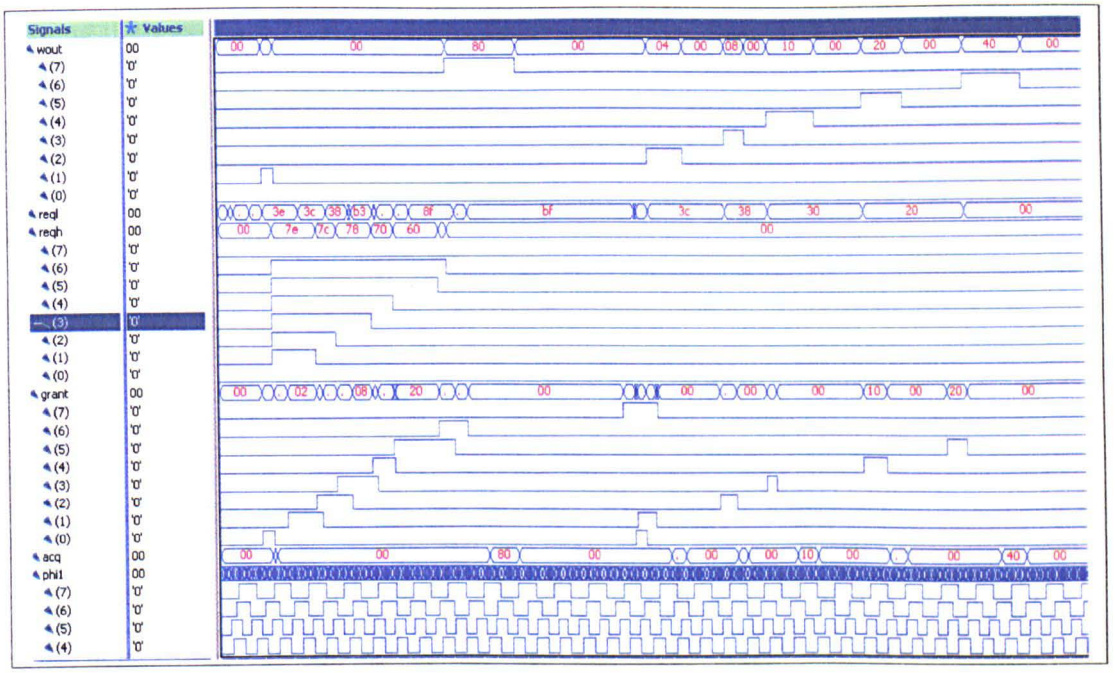


Figure 6.20: Arbiter simulation waveforms snapshot 1 (8 arbiter modules).

6.6 Summary

In this chapter we have discussed the design and the pre-layout simulation using VHDL of our asynchronous arbiter. The arbiter provides a very simple system architecture, where each module has just a few wires connecting to the next one and the last is connected to the first module in a circular fashion. Delay-insensitive methodologies with unbounded wire and gate delays were considered in the arbiter simulation procedures. The arbiter also has the advantages of GALS communication design and has the following features:

- The ring configuration to arbiter modules and the point-to-point communication between two adjacent arbiter modules provide a modular connectivity, which has full scalability in both bandwidth and area with increasing numbers of

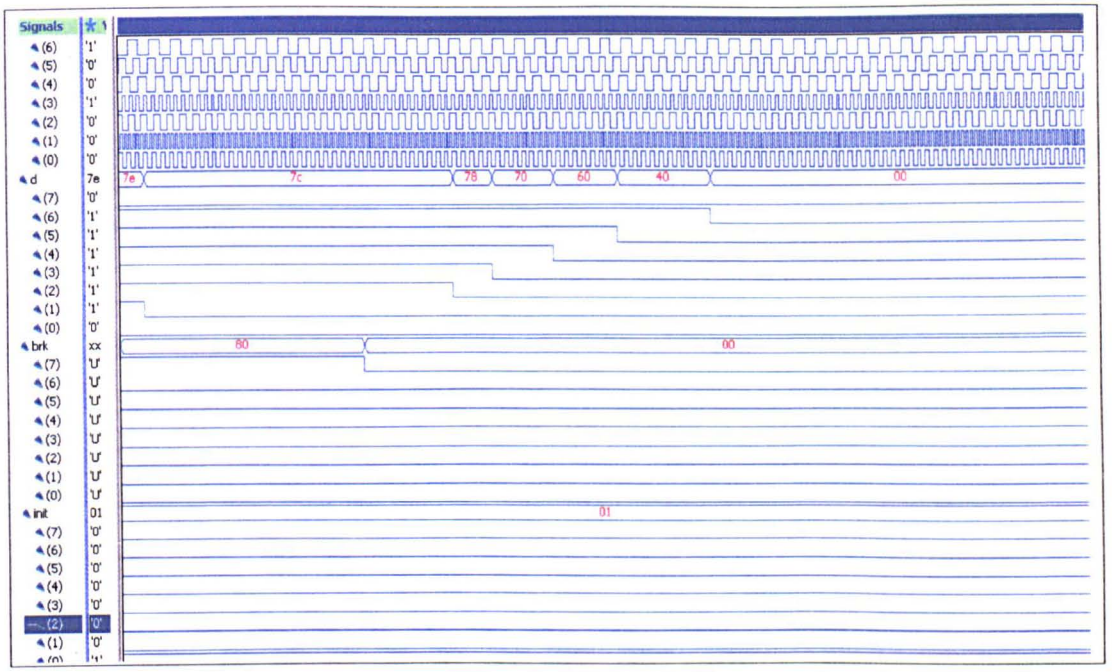


Figure 6.21: Arbiter simulation waveforms snapshot 2 (8 arbiter modules).

microthreaded processors GALS modules.

- Each arbiter module has its own control signals and implements a self-timed model. Therefore, there is no need to propagate the control signals throughout all the arbiter modules.

- There are four wires connecting every arbiter module in the chain to the next one and the last to the first in a circular fashion. The latency of the wire delay is very small. Thus the decision is made locally by each arbiter module instead of using large wire delay, which gives it a partitioning properties.

- Each arbiter has a priority policy dependent on a processor successfully executing the concurrency control instruction Brk. This mechanism provides latency hiding by decoupling the microthreaded processor from the token circulation

time. It also offers fairness in communication between processors and eliminates processor starvation.

The broadcast bus, ring network and arbiters are structured to facilitate both scalability and partitionability of the processor array.

Chapter 7

Implementation and Area estimates for Microthreaded Core and its Support Structures

7.1 Chapter Overview

In chapter 5, it was stated that the scalability of the microthreaded support structures in terms of silicon area needs to be evaluated. The chapter also showed that it is necessary to model the top-level nature of the CQ and scheduling system in order to verify their correct operation. This chapter discusses these issues and gives detailed implementations for the microthreaded microgrid support structures using VHDL.

The outline of this chapter is as follows. In the next section an overview of the microthreaded support structures is presented. Area estimates for the support structures are given in section 7.3. Implementation and simulation results for local scheduler and microthreaded pipeline are discussed in section 7.5 and 7.6 respectively. A summary of the chapter is provided in section 7.7.

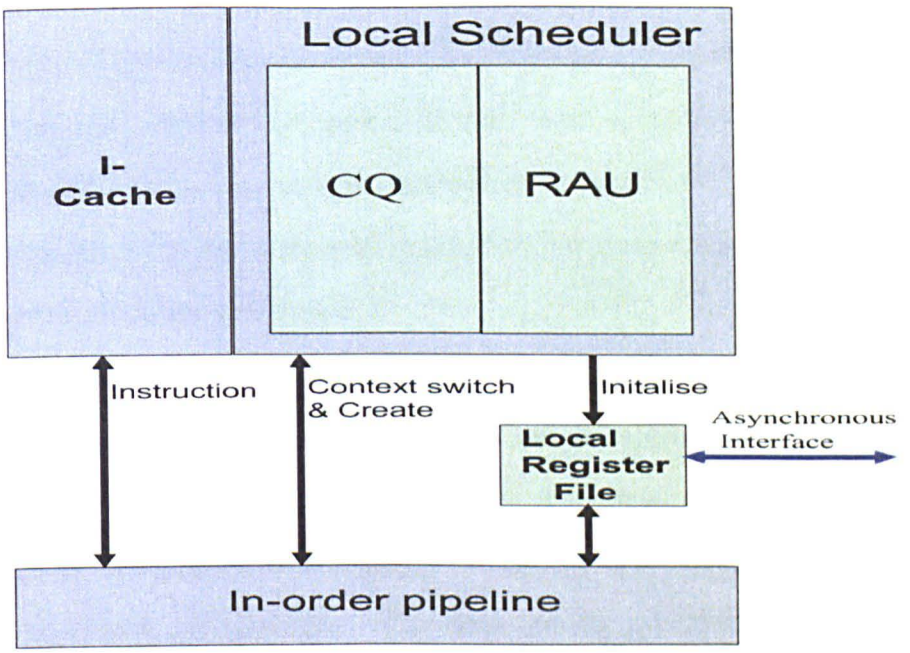


Figure 7.1: Block diagram for microthreaded support structures.

7.2 The Microthreaded Support Structures

A top level block diagram of the microthreaded support structures is shown in figure 7.1. The support structures comprise a *local scheduler (CQ and RAU)*, and a *local register file*. The support structures and the microthreaded in-order pipeline facilitate scalability in supporting high levels of ILP (e.g. thousands of processors and thousands of threads per processor). The parallelism of the code into microthreads is determined by the compiler and managed during execution by each processor's local scheduler. The local scheduler monitors local resource availability and determines when new microthreads may be started. Resource management involves allocating a set of registers for each thread created, which is performed by the RAU. The resource information also includes a free slot number in the CQ to hold the thread state. The

local scheduler determines the subset of related microthreads (thread family) that are going to execute and manages a local model of resource utilisation. During execution, microthreads may need to exchange data with other microthreads; this is done via a bank of shared registers in each processor's local register file. The local register file is fully scalable, with the implementation of its windows requiring only 5 fixed ports per processor as shown in chapter 4.

As mentioned earlier, the complexity and scaling of the processor support structures are the main significant challenges in modern processor designs. A simpler and more scalable processor requires efficient and scalable support structures with low area, and minimal communication overhead. The microthreaded microprocessor model meets these requirements. The next section provides an area estimates for microthreaded support structures.

7.3 Area Estimates for Microthreaded Support Structures

In order to demonstrate the scalability of the microthreaded support structures in terms of silicon area, this section provides area estimates and comparisons for these structures.

7.3.1 Register File

It has already been shown that a partitioned register file distributed across multiple processors is scalable, uses less area and power and has smaller delay and access times compared with global or centralised schemes [22]. Also, chapter 4 shows that a

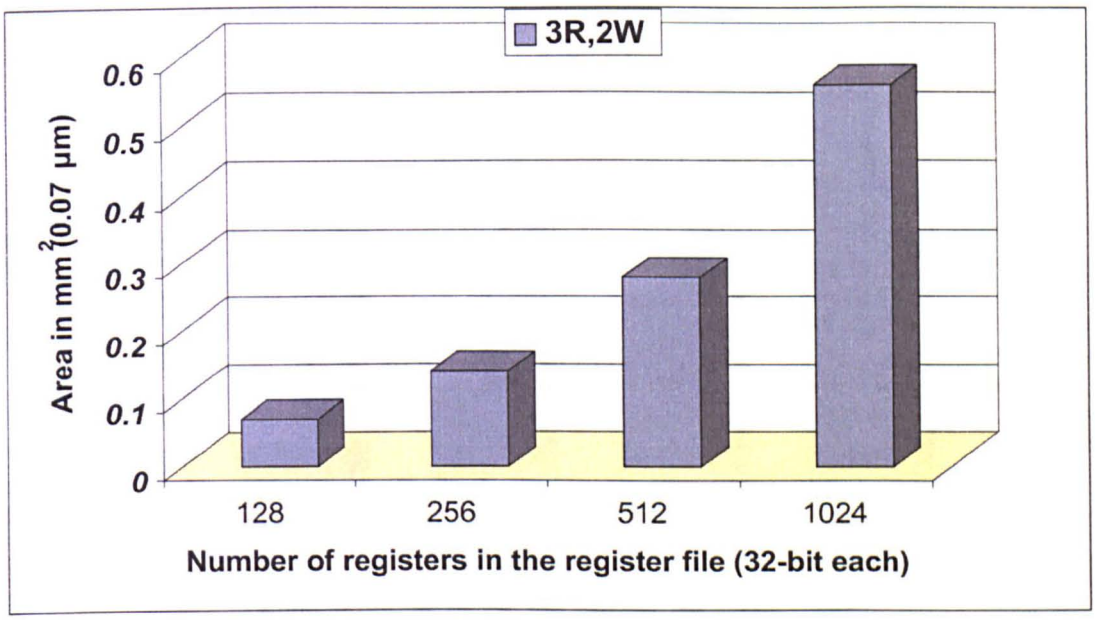


Figure 7.2: Estimated area of one processor's partition of a distributed register file comprising 5 ports per processor. The area estimate is for 0.07μm technology.

microthreaded register file is scalable and only five ports per processor are required in an implementation. Here we will demonstrate the silicon area scalability of the microthreaded register file. Using the procedure from [98, 148] we have calculated the area of our register file and compared this with the Alpha 21264. Figure 7.2 shows the estimated area for a partition of a microthreaded register file for various numbers of local registers (note that the size determines latency tolerance). It can be seen that the area of 1024 32-bit registers is less than $0.6mm^2$ in 0.07 micron technology.

The Alpha 21264 splits its integer file into two clusters that contain duplicates of the 80-entry register file. The two pipelines then access a single register file to form a cluster, and the two clusters are combined to support 4-way integer instruction execution. The architecture also has two floating-point execution pipelines organised in a single cluster with a single 72-entry register file. Figure 7.3 compares the area

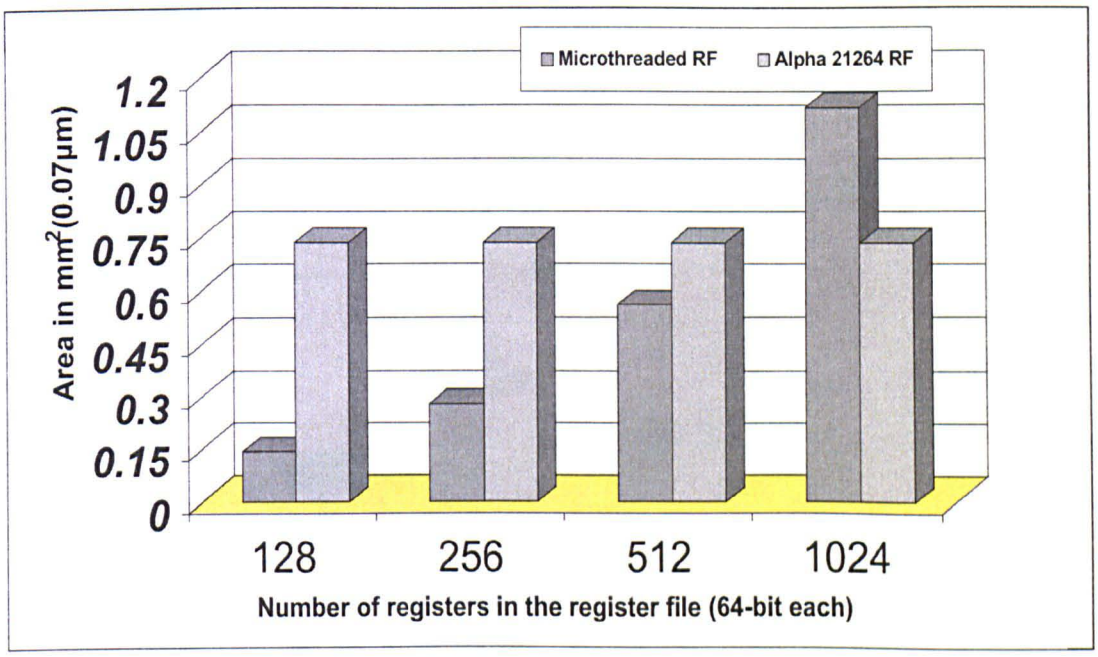


Figure 7.3: Area comparison between different sizes of a microthreaded register file partition and the alpha 21264's register file. The area estimate is for 0.07μm technology.

of a microthreaded CMP register file and the Alpha register file. The area of our register file is less than the area of alpha 21264 for all sizes up to 512 64-bit registers.

7.3.2 Register Allocation Unit

This section provides the area of the allocation scheme and compares it with the area of the register file. We have already seen that the allocation scheme is straightforward and that allocating more registers per block provides both area and propagation-delay reduction in the allocation scheme. Figure 7.4 shows a slice of RAU combinational

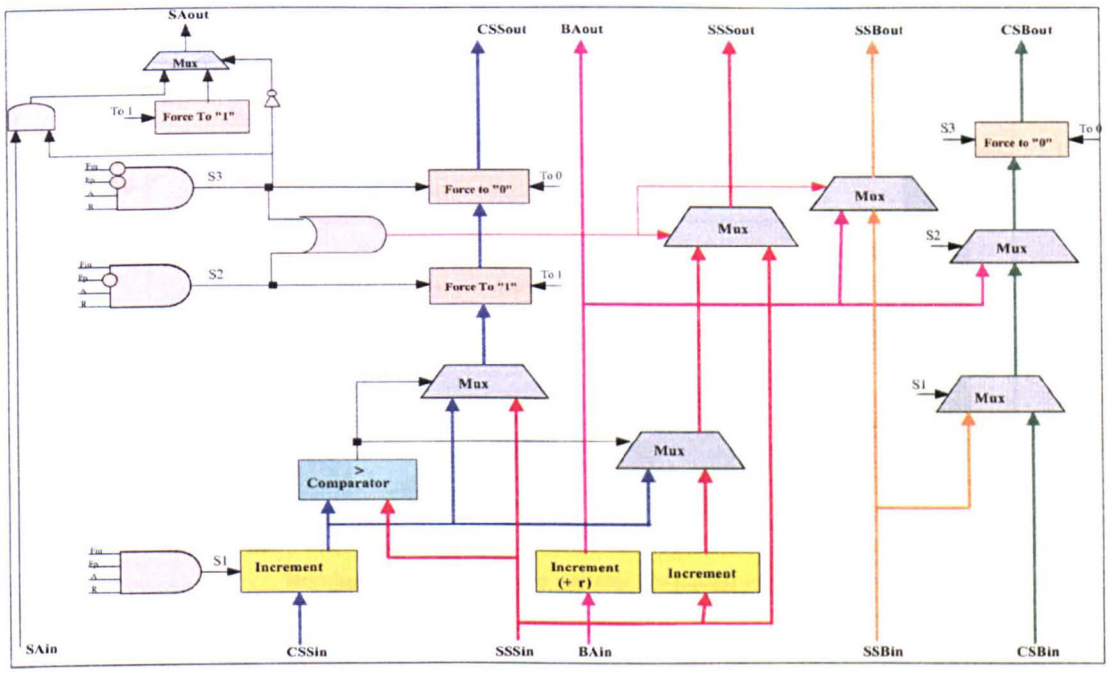


Figure 7.4: Register allocation unit's combinational logic slice design.

logic. Each slice includes a set of components: multiplexors, incrementers, comparators and logic gates. We use Standard Cell Datasheets [99] to estimate the are of these components. Figure 7.5 shows an area comparison between the register allocation scheme and the register file for 2- and 4-register allocation units. The allocation scheme uses less area than the register file in both cases. In addition, an important feature that must be considered that the allocation scheme is inversely proportional to the granularity of the allocation block, thus allocating blocks with n-registers means area and power is reduced by a factor of n. This is important because more concurrency means a greater reduction in the area and power dissipation.

A reduction in the complexity of the allocator by the use of allocation units of greater than one must be considered against any possible inefficiency in register use

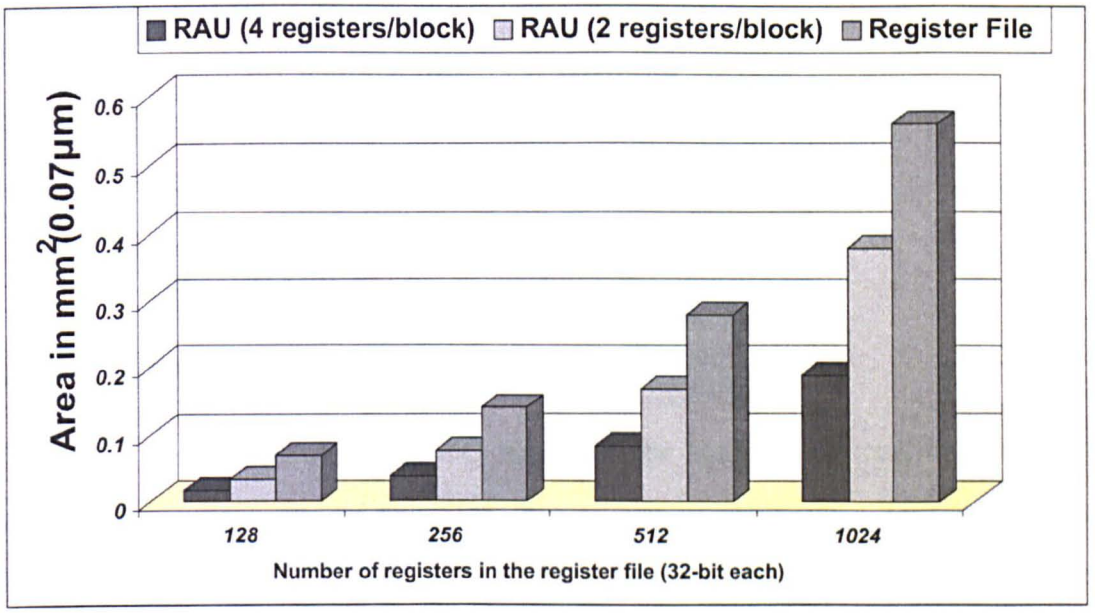


Figure 7.5: Area comparison between the register allocation unit and the register file for 2- and 4-register per allocation unit and for various sizes of register file.

caused by the blocked nature of the allocation, which may result in unused registers. Remember, using a large number of registers is important in maximising local concurrency and hence in tolerating latency. Given a known hardware scheme, any waste through non-use of allocated registers can be minimised by the compiler and this allocation scheme enables the overhead associated with dynamic allocation to be fully managed. In the limiting case, the compiler could assume microcontexts of a fixed size (16 registers), with the compiler maximising use of the microcontext by loop unrolling if necessary. The allocator then allocates registers in fixed blocks of 16, simplifying the logic and reducing the area shown in figure 7.5 by at least a further factor of 4 compared to the case of 4-register units.

Table 7.1: Thread entry format in the continuation queue for 256-entry CQ and 512 entry register file.

Field name	Number of bits
Program counter	32
Local base	9
Dependent base	10
Producer	8
Pointer	8

7.3.3 The Scheduler

Within the local scheduler, the CQ manages the state of all currently allocated threads; the components of this state are shown in table 7.1. This includes the program counter (PC), the base address of its microcontext (l-base) and the base address of a dependent microcontext if used (d-base), which includes a flag (F) to specify whether this is local or on an adjacent processor. Two additional fields are used to hold pointers to other slots in the table. The first of these is used to build queues, for example, the empty slot queue and the active-thread queue. The other is used to identify a thread's producer in the dependency chain. This is required when releasing a thread's resources, as in a dependent loop. Physical registers are shared between two different microcontexts and the producer's registers can not be released until the consumer has read them. This is implemented conservatively by releasing registers only when the consumer has been terminated. Thus the Kill instruction must backtrack one place down the dependency chain to release that threads resources. The table is initialised into a state where all slots are in the empty queue except for the main thread, if it exists on a processor, which occupies slot 0. For a 32-bit PC, a 512-location register file and a 256-entry CQ, each entry in the CQ requires 67 bits.

The structure of the CQ can be decomposed into three parts, each of which has a

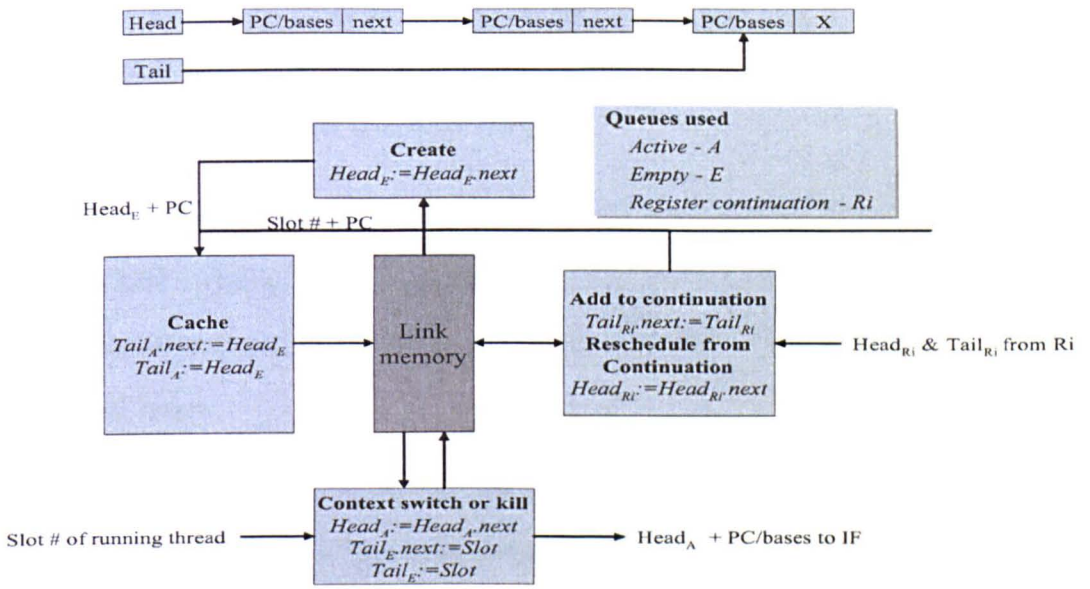


Figure 7.6: Block diagram of the interactions within the CQ, which use its link field to build: a) a queue for empty slots, b) a queue containing active slots and c) Any continuation queues for threads suspended in the register file.

different number of read and write ports :

- The first part holds the PC (32 bits) and is written on two ports, one when a thread is created and the other when a thread is rescheduled. Both may occur at a high frequency, so two ports are required to reschedule and create in the same cycle. There are also two read ports, one to access the head of the active queue to provide a PC on a context switch, and a second to obtain the PC of a suspended thread when it is rescheduled after suspension in a register. This must be sent to the I-cache to pre-fetch its code before the thread can be placed in the active queue. Again both can occur frequently and two ports are required to perform both in the same cycle;

- The second part (27 bits) holds the microcontext state (base addresses etc.)

and requires only two ports, one of which is written to when the thread is created and the other is used to access the head of the active queue to provide the base addresses on a context switch;

- The last part is used to organise the thread slots into various queues and is the link field (8 bits). This is accessed in each cycle to maintain various mutually exclusive queues, which are linked using this field. They are discussed in more detail below.

All queues are maintained using the link field to indicate the *next* slot number in the structure and two registers are used to maintain pointers to the *head* and *tail* of the corresponding queue. This is illustrated in figure 7.6. The head, tail and link fields are used to address all of the memories defined above.

Figure 7.6 also shows the various processes involved in managing the thread state, i.e. thread creation, pre-fetching the code into the I-cache, building CQs on registers and context switching. When a thread is created, a read port is used to update the head of the empty queue. The slot number from the old head of this queue is the one allocated to the new thread created and this is passed to the I-cache along with the thread's PC to initiate a prefetch. When the PC address is known to be in the I-cache, the new thread is added to the tail of the active queue, which supplies new threads to the pipeline on a context switch. When a context switch or kill occurs at the IF stage of the pipeline, a read is required to update the head of the active queue. Also, but only on a kill, a write is required to update the tail of the empty queue. This requires two ports, as the read and write are to different addresses in the CQ. Finally, a process is required to manage the CQs of threads suspended on a given register (Ri in figure 7.6). This will either write to the link field to update

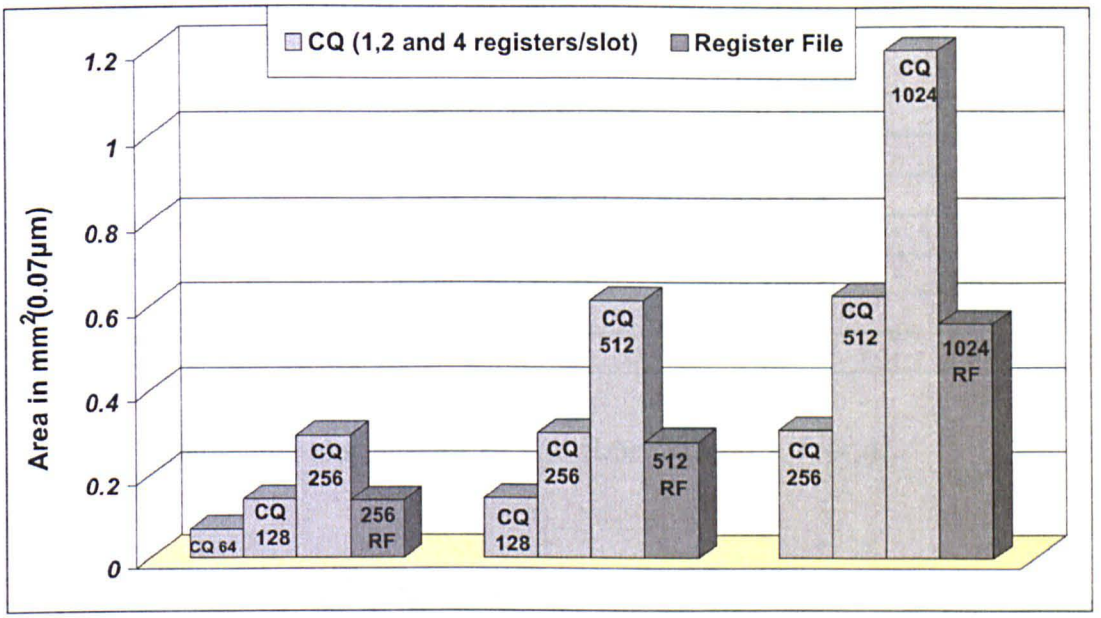


Figure 7.7: Area of the CQ compared with the register file for 1, 2 and 4-registers per slot in the CQ.

the tail of the Ri queue, when a new thread is added or read the link field to update the head of the Ri queue, when rescheduling a thread from it. This requires a single read/write port. In total therefore, this part of the CQ requires 5 ports.

The size of the CQ is related to the size of the register file through two parameters. The first is the number of registers required per microcontext (Rmc) and the second is the number of threads per microcontext (Tmc). The more registers per microcontext the smaller the CQ in comparison to the register file. The more threads per microcontext, the larger the CQ in comparison to the register file. We have already shown that over a sample of the Livermore loop kernels, the average number of registers per microcontext was 6. The optimal number of threads per context is more difficult to ascertain, without significant analysis of simulation results. For this reason, figure 7.7 shows an area comparison between the CQ and the register file for

Table 7.2: Microgrid-Core estimate area using 0.07μm technology.

Functional Block	**Size**	Area in mm^2 (0.07μm)	**%Core**
L1 I-cache	8KB, Direct map	0.178	7%
L1 D-cache	64KB, 2-Way	1.15	47%
Register file	512 (32-bit each)	0.279	12%
RAU	Allocate block of 2	0.167	7%
CQ	256-entry (67-bit each)	0.299	12%
FPU	64-bit	0.356	15%
Total Core Area (mm^2)		2.43	100%

1, 2, and 4 registers per slot in the CQ (i.e. Rmc/Tmc = 1, 2 or 4).

7.4 Estimated Core Area

In this section an estimate is given of the number of microthreaded processors that can be integrated onto a single chip using emerging technology (0.07 micron CMOS). We assume that each core in the microgrid CMP is a 32-bit RISC processor with a dedicated, 64-bit, floating-point unit (FPU). We consider two possible architectures, which correspond to the memory organisations briefly described below, i.e. with and without D-caches. To estimate the microgrid-core area, we have used CACTI to estimate the area of the L1 caches and we use [98, 149] to estimate the area of the other core components. Note that both I-cache and D-cache are single port memory structures.

In the future work we introduce two possible memory organisations for microgrid CMPs. The first uses a processor with a single L1 D-cache per processor supported by a cache-only memory architecture *(COMA)*. The second possible memory architecture eliminates the L1 D-cache completely and makes use of latency tolerance in

Table 7.3: Microgrid-Core estimate area without L1 D-cache using 0.07μm technology.

Functional Block	**Size**	Area in mm^2 (0.07μm)	**%Core**
L1 I-cache	4KB, Direct map	0.08927	8%
Register file	512 (32-bit each)	0.279	23%
RAU	Allocate block of 2	0.167	14%
CQ	256-entry (67-bit each)	0.299	25%
FPU	64-bit	0.356	30%
Total Core Area (mm^2)		1.19	100%

the processors to access a flat multi-banked memory structure. The choice of memory structure for the CMP is a complex one and is likely to be application specific. For this reason, in this section, we simply assume that half of the chip area is given over to processors and the remaining half to memory structures such as memory banks and the network to access them.

Table 7.2 gives the estimated area of a microthreaded processor core including an L1 D-cache. In this table, we assume that the processor has a direct-mapped L1 I-cache of 8KB and a two-way set-associative L1 D-cache of 64KB. It can be seen that the L1 D-cache consumes about 47% of the core area and that the register file of 512 registers consumes 12%. Based on the work presented in this thesis, we assume that the RAU allocates registers in units of 2 and that the size of the CQ is 256-entries. This gives support structures for the microthreaded model that consume 7% and 12% of the core area respectively, giving a total area for the processor core including the FPU of $2.43mm^2$.

Results for the alternative configuration without the L1 D-cache use similar parameters, with the only difference being that the L1 I-cache is reduced to 4KB. The results are shown in table 7.3. In this configuration, the support structures begin to dominate the core area, with 23% going on the register file, 25% on the CQ and 14%

on the RAU. However, the new estimated core area is now only $1.19mm^2$, which is less than half the area of the previous configuration. It should be noted that with 512 registers and 256 microthread slots, we have chosen to characterise a generous configuration that would tolerate hundreds of cycles of latency from a memory system. To put these estimates in perspective, using the model without the L1 D-cache, if we assume that half of the die area is given to memory structures, a 128-processor chip with 64 thousand registers would require $305mm^2$, which is significantly less than the area of Intel's Montecito chip.

Recently, Kumar et.al. [150] estimated the die area of chip multiprocessor with eight cores sharing a 4MB L2 cache. In their work each core is a 4-issue in-order processor (Alpha 21164) and has 64KB L1 caches (I/D). The total chip area was $127.76mm^2$. Our estimation methodology is similar to their work and we have used the same feature size. Using the same die size and the same amount of shared memory, we could support about 50 microthreaded processors each with an FPU with a combined register file size of 25 thousand registers and able to support over 10 thousand active threads. Sharing an FPU, as proposed in that paper, is quite feasible in a microthreaded processor design and this would further increase the number of processors in the same area. A co-joined dual processor single FPU processor design would require approximately $2mm^2$, allowing 64 processors with 32 thousand registers to be integrated in the same die area.

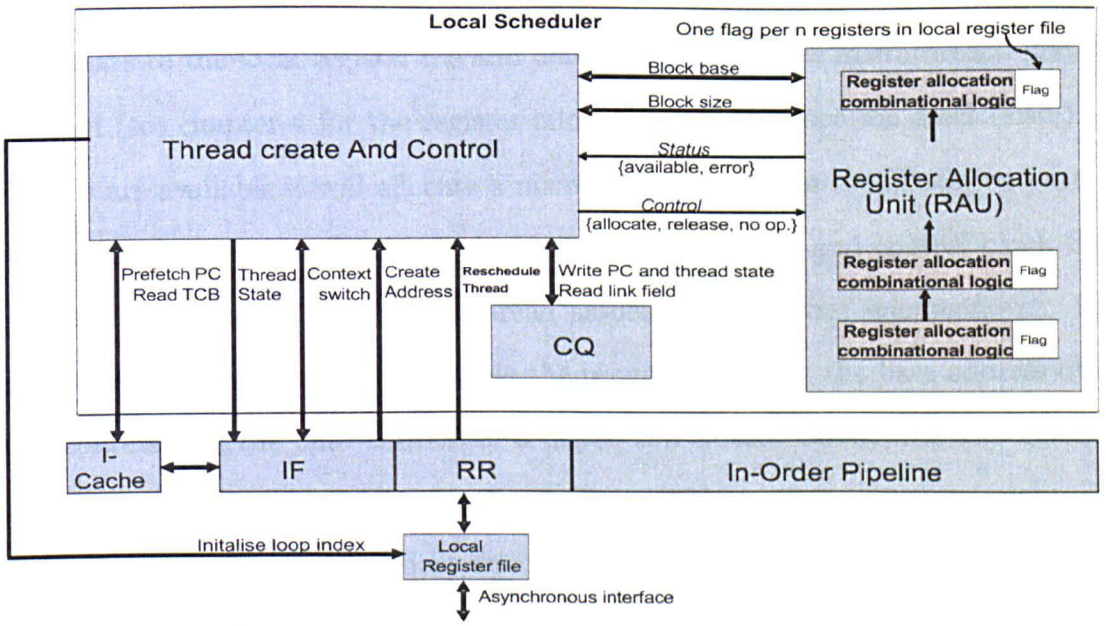


Figure 7.8: Detail of the local scheduler showing its main components and the data paths between it and other stages of the pipeline.

7.5 Implementation for a Local Scheduler and a Microthreaded Pipeline

We have already described a detailed design and implementation of the local scheduler and its components *(RAU and CQ)*. We also, explained that the area of both the allocation scheme and the scheduler queue are less than that of the register file, given reasonable assumptions about the size of each. In this section, we combine the implementation of the scheduling system together with the first two stages of the microthreaded in-order pipeline to verify their correct operation. A block diagram of a local scheduler and its connections with the I-cache and the processor pipeline is shown in figure 7.8. As shown, the local scheduler has three main components, which comprises: the *RAU*, *CQ* and *thread-create and control block*.

The RAU within each scheduler models the allocation and deallocation of microcontexts to the local register file and determines when new microthreads may be allocated (see chapter 4 for the register allocation and deallocation mechanism). If registers are available it will allocate a microcontext and pass the allocation parameters to thread-create and control block. The thread-create and control block then creates entries in the CQ for each thread associated with that microcontext. The entries, as described previously, include the program counter, the base address of its microcontext and the base address of a dependent microcontext, flag and thread's producer (see section 7.3.3 for more detail about CQ).

Thread entries are managed by two pointers, *empty head* and *active head.* The empty head pointer provides the available empty slot in the queue table. Thus, as soon as a thread's parameters become available, then the current empty slot is associated with that thread's parameters and removed from the empty head register. The next available empty slot now becomes the new slot in the empty head register for the coming thread (empty head.next = new empty head). Also, as soon as a thread's parameters and its slot number becomes available, the thread PC and its slot number are used to request the I-cache to prefetch the code before considering that thread for execution. If the thread code is available, then the I-cache acknowledges (ACK) the thread-create and control block, which results in updating of the tail of the active thread pointer.

Also, when a context switch occurs, the current head of the active queue is removed (it is now the thread executed in the next cycle) and the next thread becomes the new active thread. This requires updating the head pointer to the new active thread. Also, if in that cycle, the instruction is kill (lets assume that the threads are independent),

then two actions are required. The first is obtaining the next thread from the CQ (active head.next = new active thread), and updating the head of the active queue. The second action is to add the killed thread (immediately as it is independent) to the tail of the empty queue, by writing the slot number of the killed thread to both the next field of the current tail, and to the tail pointer (and releasing its registers). Thus the active threads are removed on the context switch and then can be added if they are rescheduled.

The thread-create and control block work as intermediaries between the RAU and CQ from one side and the I-cache, and processor pipeline on the other side. As shown in figure 7.8, this block receives a pointer to the TCB (create Address) from the processor pipeline when a create instruction executed. It also receives a pointer from the pipeline register read stage (RR) when a thread is rescheduled. A kill state (kill thread) is also required to indicate those threads that have been completed. This block, as mentioned above, also generates a request to the I-cache to prefetch the code for any thread that enters the waiting state.

If the code is available, then the I-cache acknowledges the scheduler immediately, which changes the thread's state to *Active*. Active threads must wait their turn in the CQ before being selected for execution. The thread and control block also supplies the the processor pipeline with the thread state (PC,l-base,d-base and slot number). Finally, an initialisation pointer from the scheduler to local register file is used to initialise the $L0 to the loop index.

The thread-create and control block has two main processes: fetch TCB and allocate thread, both of which work concurrently. The state machine diagram for the first process is shown in figure 7.9. As shown, the *idle* state changes to the *fetch*

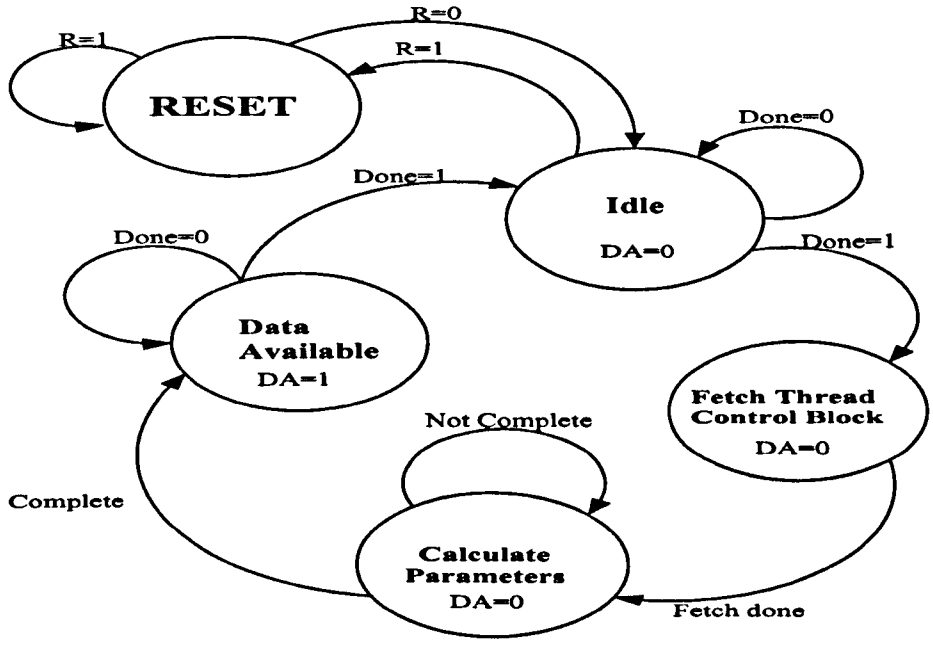


Figure 7.9: Fetch thread control block state transition diagram.

TCB state when it receives a high signal through the done line, which informs it that the previous family has been fetched. When the TCB has been fetched, the state changes to the *calculate parameters* state, in order to determine the allocation parameters. Finally, the state changes to the *data available* state, which informs the allocation process through the data available *(DA)* signal that the parameters have become available.

The second process is used to allocate a thread in each machine cycle if the required space is available. As shown in figure 7.10, the state machine changes from the *family waiting* state to the *allocate thread* state when the required parameters become available i.e. DA=1. When a thread is allocated, its parameters are stored in the CQ. Thus, from the *store parameters* state the machine returns to the allocate another thread if the given family is not completed, otherwise it changes to the *family*

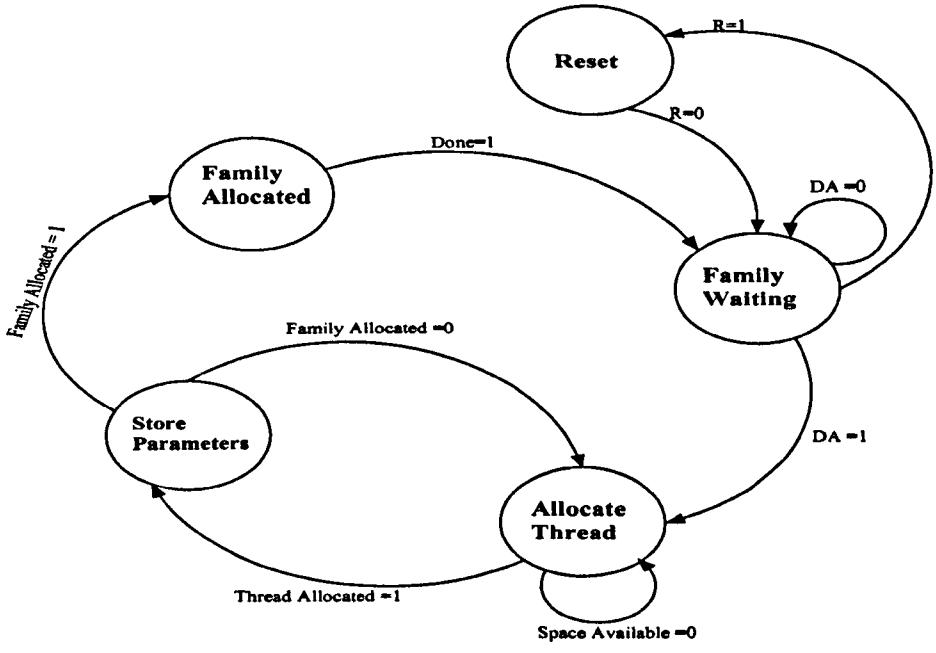


Figure 7.10: Allocate thread state transition diagram.

allocated state. In the *family allocated* state, a high signal is asserted on the fetch thread control block which indicates that the current family has been allocated.

Figure 7.11 shows the first two stages of the microthread microprocessor in-order pipeline. The instruction fetch (IF) stage fetches two instructions from the instruction memory simultaneously. The justification for fetching two instructions is to avoid a pipeline stall. The first instruction is fed to the pipeline for execution, while the second instruction is tested to see whether it is a normal instruction or a microthreaded instruction (predecode of the second instruction). The logic for predecode the second instruction (a simple hardwired decode) is entirely in the IF stage of the pipeline and there is no overhead in terms of additional pipeline cycles to perform the context switch. This is achieved by prefetching all concurrency-control instructions with the preceeding executable instruction. Thus, as mentioned earlier, a context switch

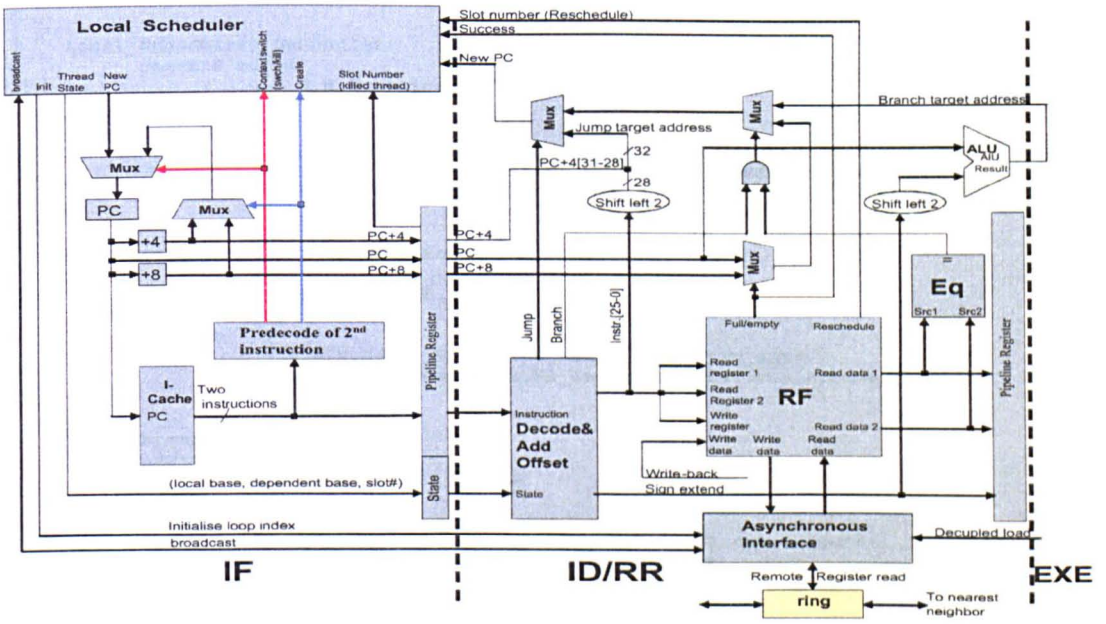


Figure 7.11: The first two stages of microthreaded in-order pipeline.

follows an executable instruction if the compiler identifies that instruction as a non-deterministic event. Another situation where the compiler will flag a context switch is following any branch or jump instructions. In this case the thread is reactivated upon the computation of the branch or jump target address, in the second stage of the pipeline after register read.

Register addressing uses a simple base + offset mechanism, where the base address is a part of a microthread's state and the offset is defined by the register specifier in the instruction execution. Notice that instructions normally complete in order but that in circumstances where the execution time is non-deterministic, such as a D-cache miss, data is written asynchronously to the register file on a port dedicated to this purpose. In this situation, instruction issue stops in a thread as soon as an instruction attempts to read a register that is empty.

```
Local_Scheduler: Scheduler
      generic map (
                    w,M,S ,Slice_id, tdelay
                  )
      port map      (
                    t_clk,Rst,WRREQ,Newpc ,Cre_address,Reschedule,Found,Family_Data,
                    Acknow ,  Conxt_switch, PC,L_Base,D_Base,Slot_pip,
                    Prefetch,Slot_cache,nxt,RDWR, RDM_prefetch ,TCB_Addrs,write_pc
                  );

Instruction_Memory : Instruction_Cache

     generic map ( w ,Slice_id, tdelay
                 )
     port map    (
                   t_clk,Rst,  RDWR, TCB_Addrs  , Family_data,
                   Inst_data,Inst_Addr,RD_Cache,RDM_prefetch,Acknow,Prefetch
                 );

Microthreaded_Pipeline:  CPU
    generic map  (
                  w,Processor_id  ,N =>N
                 )
    port map      (
                  t_clk,rst,nxt,PC,write_pc, add_st,Cre_address , Found ,
                  Conxt_switch,Inst_data,Inst_Addr,RD_Cache
                  );
```

Figure 7.12: VHDL test bench source code for local scheduler, microthreaded pipeline and I-cache.

Note that a *Swch* instruction will always update the value of the PC in the thread's state, and this update occurs after the register-read stage. This is in the case of a branch but not so obvious following a data dependency, where the state of the register will determine whether the instruction will be re-executed or not. If a register reads fails, the instruction reading the register must be re-issued, when the data is available. On the other hand, if the register read succeeds, the next instruction must be executed, which may be the next executable instruction or the one at the branch target location, thus the action at the register read stage determines the value of the thread's PC for all programmed context switches. The pre-layout simulation of the local scheduler and the first two stages of the microthreaded pipeline using VHDL is presented in the next section.

```
Controller: Control
      generic map  (
                    w,M,S ,Slice_id, tdelay
                   )
      port map     (
                    t_clk,RST,Reles_Base,Required_Size,Doallocate,Dorelease,
                    Allocate_Bas,Available_size,Error_signal,
                    Space_Found,family_Found  ,WR_create, RD_Memory, Create_Address, TCB_Addr,
                    TCB_Data,PC_Created,L_Base, D_Base,F,producer,WR_Queue,Next_Family ,don
                   );


 Allocation: Allocate
      generic map  (
                    w,M,S ,Slice_id, tdelay
                   )
      port map     (
                    t_clk,RST,Reles_Base,Required_Size,Doallocate,Dorelease,
                    Allocate_Bas,Available_size,Error_signal,Space_Found
                   );

 Continuation_Queue:  CQ
      generic map  (
                     w,M,S, Slice_id,tdelay
                   )
      port map     (
                    CLK,RST,PC_Created ,PC_Reschedule,L_Base,D_Base,
                    F,producer,WR_Queue,Contxt_switch, PC_pipeline,
                    L_Base_pip,D_Base_pip,Slot_Number_pip,
                    Prefetch_PC ,Slot_Number_cache, RD_memory_prefetch,
                    WR_PC, Ack ,don
                  );
```

Figure 7.13: VHDL code for local scheduler components.

7.6 Simulation Results Using VHDL

We have modeled the behaviour of the processor pipeline and the local scheduler in VHDL. The processor has the I-cache and the first two stages of the pipeline, which represent the top-level nature of the CQ and scheduling system. A snapshot from the VHDL test bench code for the processor pipeline and the local scheduler is shown in figures 7.12 to 7.14. The VHDL code has been run using various compile scenarios, and with different thread allocation size implementations. In effect, we have used loop kernels at this stage as we currently have no compiler to compile complete benchmarks. However, as the model only gains speedup via loops, we chosen different types of loops from scientific and other applications. Analysis of complete programs and other standard benchmarks will be undertaken when a compiler, which is currently being developed, is able to generate microthreaded code.

```
PC_reg: register_32
   port map(clk2, Rst, PC_next, PC);
            PC_incr: add32
   port map(PC, four_32,eight_32, zero1, zero2, PC_next,PC_next_8, nc11,nc12);
inst_mem: instruction_memory
   port map(PC, inst);
ID_IR_reg: Instruction_register_32
   port map(clk, Rst, inst1,inst2, ID_IR1,ID_IR2,
          Create,Context_Switch,Kill_thread,base);
NewPC_mux : mux_32
   port map( in0             => New_Pc,
             in1             => PC_next,
               swch          => Context_Switch,
              result         => WB_result
          );
ID_regs:registers
  port map(
             read_reg_1      => ID_IR1(25 downto 21),
             read_reg_2      => ID_IR1(20 downto 16),
             write_reg       => WB_rd,
             write_data      => WB_result,
             write_enable    => WB_write_enb,
             read_data_1     => ID_read_data_1,
             read_data_2     => ID_read_data_2
           );
ID_mux_rd:mux_5
  port map( in0              => ID_IR1(20 downto 16),
            in1              => ID_IR1(15 downto 11),
            ctl              => RegDst,
            result           => ID_rd
            );
   ID_sign_ext(15 downto 0) <= ID_addr;
   ID_sign_ext(31 downto 16) <= (others => '0');
```

Figure 7.14: VHDL code for microthreaded pipeline components.

Figures 7.15 to 7.18 shows samples of results showing the behaviour of the previously described state machines and processor support components. Once a family fetches, the allocation process starts allocating one thread in every machine cycle. The allocated thread parameters are stored in the CQ and wait their until it is served by the processor pipeline. The IF stage of the pipeline keeps fetching instructions from the instruction memory, until it encounters a *create*, *swch* or *kill* instruction and a process based on the behaviour of each. VHDL source code for the processor pipeline and the local scheduler, and more simulation waveforms samples are available in appendix D.

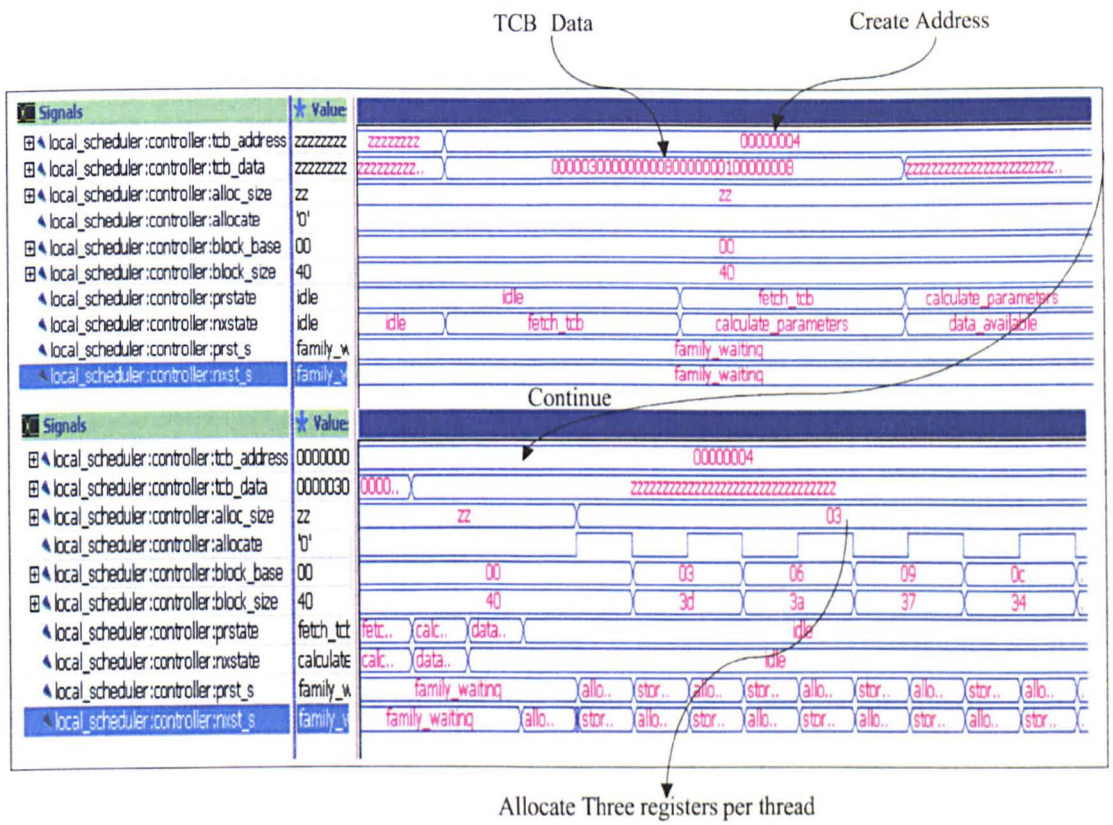


Figure 7.15: Waveforms sample result for threads creation and allocation process.

7.7 Summary

In this chapter, we have investigated the overhead of the support structures for a microthreaded microprocessor implementation, these are the CQ and RAU, as well as a larger than normal register file. All three structures are related to the local concurrency support, and hence the latency tolerance of, the processor.

We have described in detail a register allocation scheme, which dynamically allocates registers to microcontexts. It is shown that, for a given ISA, the scheme has an area proportional to the register file size. Moreover, the area required is tunable by

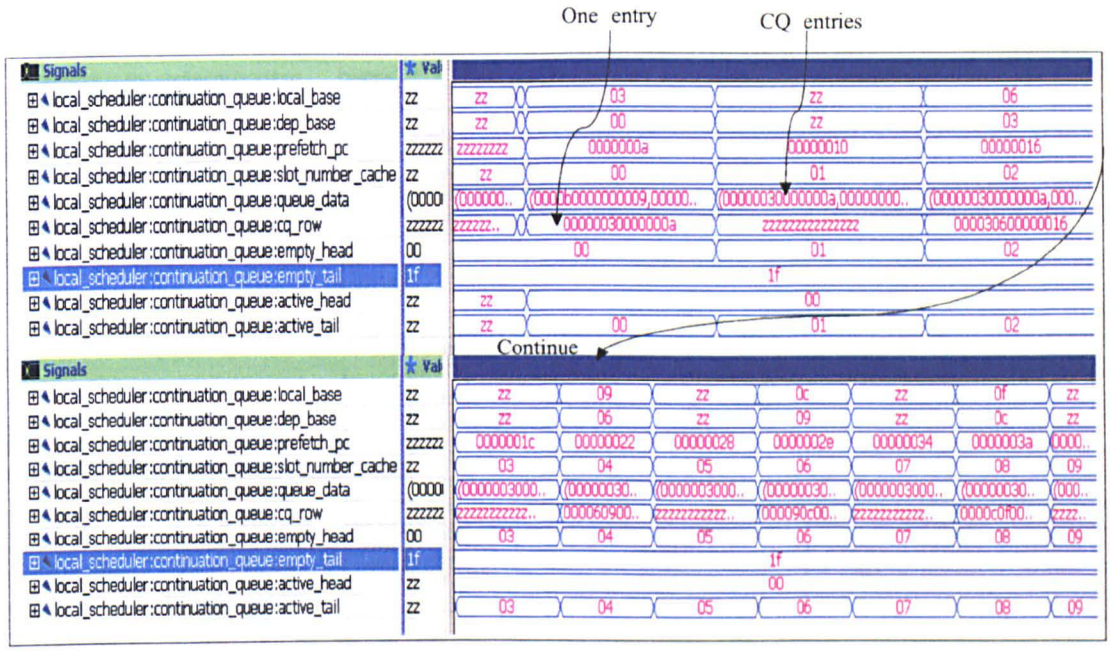


Figure 7.16: Waveforms sample result for the continuation queue.

choosing the unit of allocation, at the cost of some loss of efficiency in the use of the register file.

Our results show that the area of both the allocation scheme and the scheduler queue are less than that of the register file, given reasonable assumptions about the size of each. In effect, the size of the CQ is similar in complexity to the register file and the results in this chapter show that even considering all concurrent accesses to the CQ, the size of a 256-entry thread-state memory is smaller than the register file. The chapter also estimates the microgrid area for different configurations of memory and cache using an $0.07\mu m$ technology.

This shows the feasibility of 128-way CMPs using this emerging technology and with a generous latency tolerance capability, i.e. tolerating many hundreds of cycles of latency on memory or external I/O. We also concluded that each microthreaded

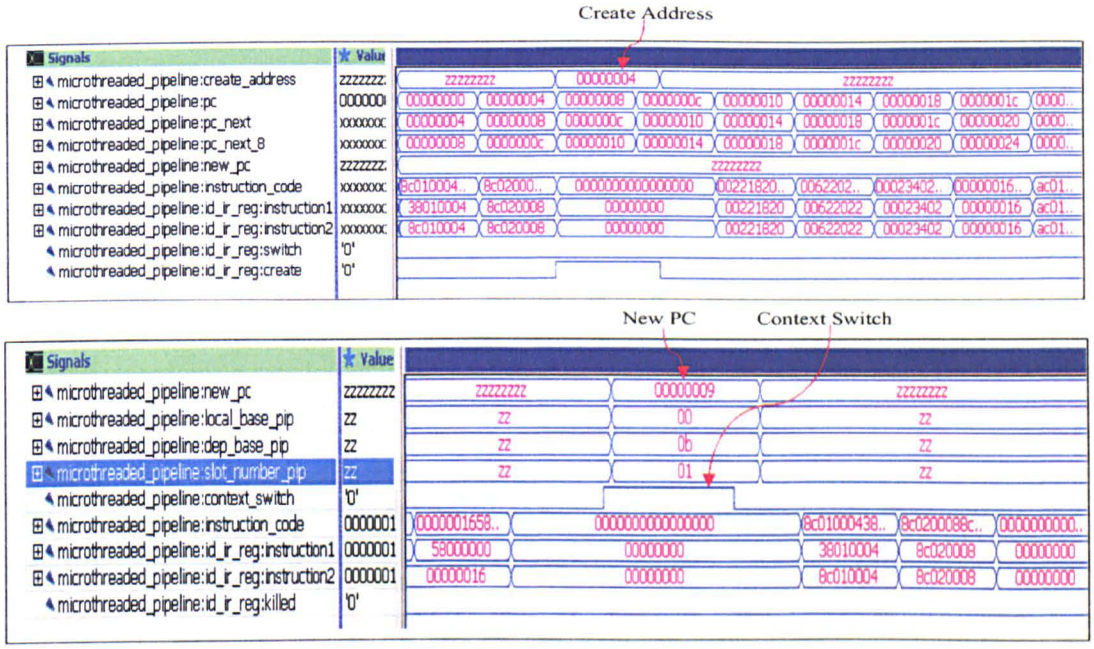


Figure 7.17: Waveforms sample result for microthreaded pipeline.

pipeline could support more than 512 synchronising registers in an area less than a 64-bit FPU, which would support in the order of hundreds of local concurrent threads.

Also, in this chapter we discussed the pre-layout simulation using VHDL of a local scheduler and the first two stages of the microthreaded in-order pipeline. The results show correct operation for theses components, and we have verified various execution scenarios and with different thread allocation size implementations.

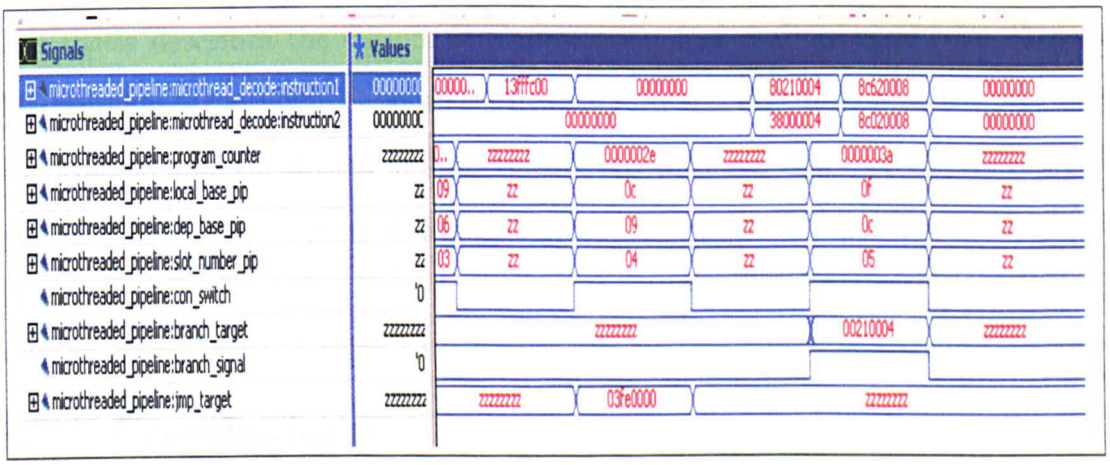


Figure 7.18: Waveforms sample result for microthreaded pipeline showing the execution for branch and jump instructions.

Chapter 8

Conclusions And Future Work

This thesis describes the design, implementation and evaluation of microthreaded CMP support structures. These structures are fully scalable, providing the possibility of a scalable implementation of a microthreaded CMP. In this chapter, we draw our conclusion for the work presented in this thesis and we discuss some aspects for future work.

8.1 Conclusions

Chip multiprocessors *(CMPs)* are becoming increasingly attractive for obtaining high performance and low power consumption, and we expect that many new microprocessor designs will be based on this approach. However, problems such as the complexity of the issue window in wide-issue processors, increasing on-chip memory in existing processors, serious clock skew, multi-ported register file scalability, centralised global communication requirements and speculative execution are obstacles and challenges facing present and future CMP designs. The microthreaded model avoids all of the above issues, and provides a suitable basis for developing systems with multiple processors on-chip. The model is based on decomposing a sequential program into small

fragments of code called microthreads, which are scheduled dynamically and which can communicate and synchronise with each other very efficiently. This process allows sequential code to be compiled for execution on scalable chip multiprocessors. Moreover, as the code is a schedule invariant, the same code will execute on any number of processors limited only by problem size.

The model exploits ILP within basic blocks and across loop bodies. In addition, this approach supports a pre-fetching mechanism that avoids any I-cache misses in the pipeline. The fully distributed register file configuration used in this approach has the additional advantage of full scalability, with the decoupling of all forms of communication from the pipeline's operation. This includes memory accesses and communication between microcontexts. Microthreading is therefore a good candidate for scalable chip multiprocessors and holds great promise for achieving scalability in future systems. However, the microthreaded model and related CMPs still have a number of problems and unresolved issues, some of which have been addressed by this thesis.

Microthreaded register file design avoids a centralised register file organisation, but its requirements in terms of the number of required read/write ports were not clear. This problem was investigated and an analysis of the register-file ports in terms of the frequency of accesses to each logical port is described in chapter 4. The results shows that the register file can be distributed between the processors and that each register file requires only 5 fixed ports, making it compact and scalable. This work has been published in the British Computer Journal [26].

The distributed implementation of a microthreaded CMP includes two forms of asynchronous communication. The first is the *broadcast bus*, used for creating threads

and distributing invariants. The second is the *shared-register ring network* used to perform communication between the register files in producer and consumer threads. Therefore, to avoid contention during bus access, and to provide fairness in communication between processors, we need some form of arbiter. Also, it is not clear how the bus interface between processors can be implemented. In this thesis we have investigated this problem and have proposed a novel ring-structured arbiter optimised for this application. The arbiter utilises the concurrency control instruction *Brk*, provided by the microthreaded microprocessor model, to set a priority policy that hides the token circulation time by decoupling the microthreaded pipeline from the ring's timing. It also provides multiple features such as modularity, and partitionable organisation *(see Chapter 6)*. This work has been published in [27, 28].

The microthread model requires dynamic register allocation and a hardware scheduler, which must support a considerable number of microthreads per processor. Allocating registers dynamically requires an efficient hardware scheme to model and allocate register usage. The design of a novel allocator and scheduler, together with detailed evaluation and simulation results are presented in chapter 4 and chapter 5. The allocator can allocate registers in fixed a block, which simplifies the logic and reduces the area significantly. In addition, the scheduler must support thread creation, context switching and thread rescheduling on every machine cycle to fully support this model, which is a significant challenge. To demonstrate the feasibility and scalability of the microthreaded support structures in term of silicon implementation, we performed a detailed implementation and area estimate of a microgrid core and its support structures using 0.07 micron technology (see chapter 7) . We show also that the support structures are of a manageable size and moreover are scalable in issue

width. This work also has been published in the Parallel Programming Journal [29]. We have concluded from this study that each pipeline could support 512 synchronising registers in an area less than a 64-bit FPU, which would support of the order of hundreds of local concurrent threads.

Also, we have shown in this thesis that the area of the support structures for a microthreaded microgrid are scalable in instruction-issue width, as they are distributed to the processors, but we have also shown that the structures are scalable in the virtual concurrency supported on a local processor, which determines the amount of latency tolerance. Because of this, performance, power and latency tolerance can all be managed, the latter in the microgrid processor design and the former two in the dynamic management of concurrency in a microgrid.

Finally we present results of the pre-layout simulation using VHDL of a local scheduler and the first two stages of the microthreaded in-order pipeline *(see Chapter 7)*. The simulations show correct operation and we have verified various execution scenarios for theses components. This work also has been submitted to [30]. In our opinion, a microthreaded CMP based on a fully distributed and scalable register file organisation and asynchronous global communication buses is a good candidate to future CMP.

8.2 Future Directions

There are several available avenues for future work on the microgrid CMP. We have divided them into four categories; the *memory system*, *multicluster architecture*, *microthreaded compiler*, and *microthreaded CMP fault tolerance*. These categories are detailed in the following subsections.

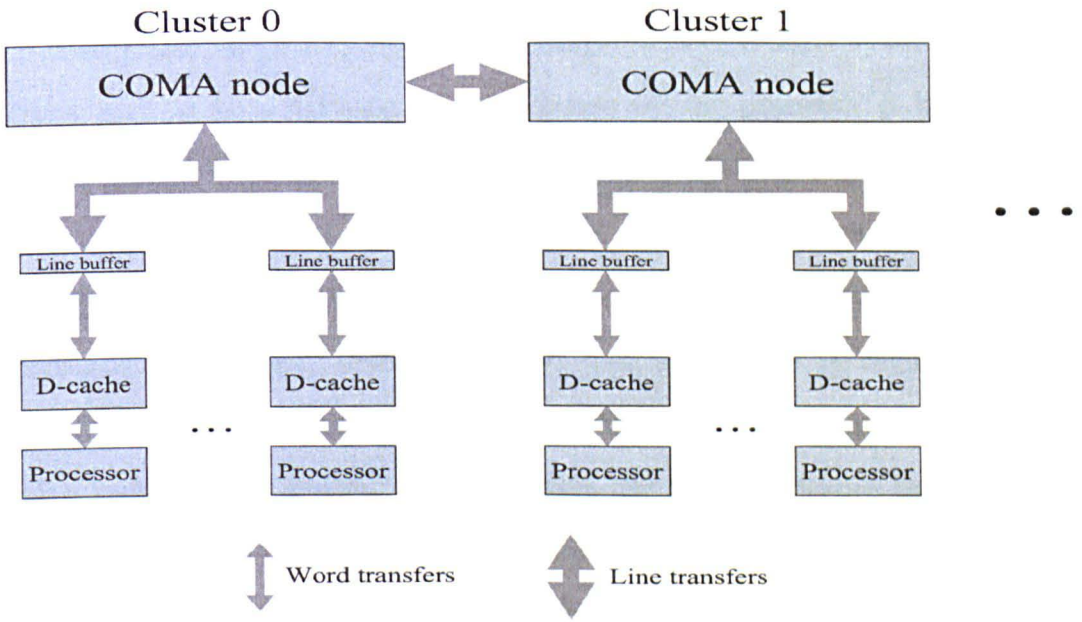


Figure 8.1: Memory architecture using COMA nodes and clusters of processors.

8.2.1 COMA versus Multibanking

The Microgrid CMP is capable of supporting a large number of processors on-chip, but such a design requires a similar number of memory banks to satisfy parallel access. The ratio of memory banks to processors is dependent on the cache hit rate and the access pattern to these banks. Two possible memory organisations are being considered. The first uses a processor with one L1 D-cache per processor supported by a cache-only memory architecture *(COMA)*. In such a memory data is automatically migrated or replicated to where it is being used by the processors. The second possible memory architecture eliminates the L1 D-cache completely and makes use of latency tolerance in the processors to access a flat multi-banked memory structure. Simulations [26] have shown that such an organisation is entirely feasible.

The advantage of the COMA structure is that it requires fewer memory banks,

as each bank can have multiple, independent cache-line buffers for each processor in a cluster (see figure 8.1) all sharing a single banked COMA node. Access to the COMA node is by a D-cache line and access by the processor is by word. This allows a number of processors, equal to the number of words in a cache line, to share a port into the COMA node without conflict, so long as there is full cache locality. Note that the deterministic distribution of threads to processors in the microthreaded model and the choice of scheduling allows data accesses to be organised in such a way as to maximise the cache hit rate and minimise accesses to the COMA nodes. Such a structure was simulated in [26] where the regular schedule produced an 80% cache hit rate with only 2-3% of memory loads causing requests to the COMA node. However, not all algorithms can be regularly mapped and some require global- rather than local-communication patterns. For example, matrix multiplication accesses data using both row and column strides through memory structures, such an algorithm would generate cache misses and bank conflicts on at least one of the strides, unless the algorithm was coded in a block structure, where the blocks matched the cache line size.

An alternate memory architecture uses a word-wide memory bank per processor with no L1 D-cache in the pipeline. All memory accesses incur a delay, dependent on location of data on chip. However, the microthreaded processors can be designed to tolerate any latency by scaling register file size and support structures to give the required local concurrency. This memory structure would still suffer from the bank conflicts in the example given above, unless some form of randomisation was employed in mapping the address space to the memory structures (see figure 8.2). The advantage of this scheme is that the complexity of the processor is reduced, by

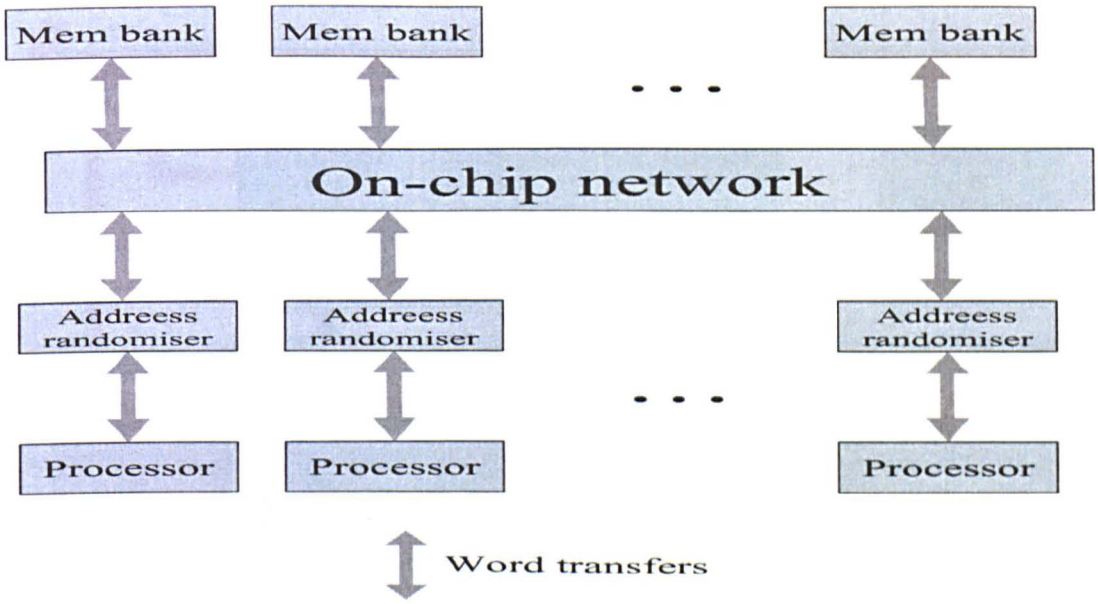


Figure 8.2: Memory architecture using a flat structure of multiple banks with address randomisation. Such an organisation would not use an L1 D-cache.

omitting the L1 D-cache. It also supports arbitrary access patterns to data, although this comes at a cost, as there is more load on the on-chip network and more energy is dissipated in moving data around the chip, as locality is ignored in randomising memory accesses. These issues have yet to be explored in depth, using our simulators, in order to find an optimal solutions.

8.2.2 Multicluster Architecture

Groups of processors in the microgrid CMP can be configured in a point-to-point ring network to form a cluster as shown in figure 8.3. The cluster has its own local COMA module, and all COMA modules are arranged in a ring organisation. To provide bandwidth and workload balancing between processors in the same cluster during remote memory access, there is word-wide access to the cache from each processor

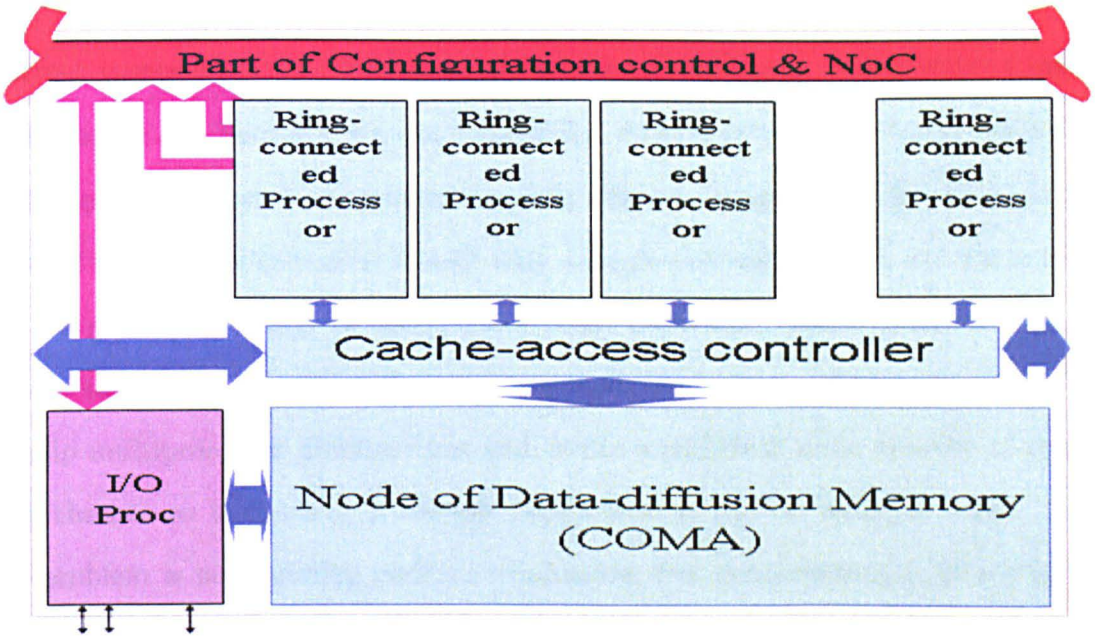


Figure 8.3: One cluster of Microgrid CMP

and cache-line wide access to memory and the COMA node. Communication between clusters is handled by a reconfigurable interconnection network on-chip *(NOC)*.

The COMA modules are connected in a point-to-point ring structure. The COMA modules use a broadcast protocol on cache misses, and a point-to-point request migration via the ring network. This aspect needs further research to determine a suitable number of processors in each cluster and to identify a mechanism that allows a group of clusters to communicate with each others with low overhead and in a scalable manner. Also, scheduling instructions in the microthreaded model is similar in complexity to that of a conventional, single-issue, in-order processor. Multiple-issue, in-order pipelines and clusters are also possible in microgrid CMPs and these aspects need further consideration, which may provide more advantages in terms of performance, reducing cycle time and providing power reduction.

8.2.3 Compiler Support

Current compilers have multiple limitations and challenges, which restrict system performance and increases system complexity. Several projects have addressed these challenges and show the requirements of an efficient compiler [55, 63, 151]. In fact, most of the existing compilers benefit only a single processor and its execution model has no global knowledge of concurrency (e.g. OOO execution). Efficient compiler design should identify parallelism automatically and must consider the advantages of chip multiprocessor architectures and create a sufficient large number of concurrent threads so that there is enough parallelism to run on multiple cores. Thus, the problem is not building multicore hardware, but programming it in a way that maximises the benefits from the continued exponential growth in CPU performance, where the architectural changes in multicore processors benefit only concurrent applications [152].

Generally, the demand for utilising ILP with multiple cores require optimisation in issues like instruction and loop scheduling, register allocation, locality optimisation, etc. [63]. Also, existing source code provides significant concurrency at the loop level and this must be exploited in any model targeting on-chip concurrency [49]. For example; explicit approaches only manage loops containing data dependencies and loop-carried dependencies are expressed as concurrently executed threads that share memory, which in fact incurs high latencies in the dependency chain.

A microthreaded microgrid compiler must consider the above issues and be responsible for code analysis (i.e. recognising dependencies between threads) and code transformations, to enable concurrency to be extracted from sequential code. It must also be responsible for thread family creation and thread grouping. Microthreaded

microgrid compiler is an active area of research, and we point here to some aspects that need further consideration. The first is that a microthreaded compiler gains speedup via loops and it is worthy to investigate the model for different applications. Also, data locality optimisation is an important issue, so it is possible to reorder groups of related threads or families of threads to increase processor locality, which can help increase parallelism and reduce memory accesses. Thus, grouping a number of independent threads together and runing these threads on one processor instead of distributing them on multiple processors may provide better performance through greater cache locality and hence fewer higher-level memory accesses. It is evident that in chip muliprocessors memory bandwidth is likely to be a limiting factor.

Moreover, it was shown that the communication between producer threads and consumer threads on remote processors required remote read actions. It could be possible for the producer thread to write data directly to the consumer thread as soon as the required data becomes available instead of using read operations. However, the problem is that the consumer thread may not yet be allocated and there is no feedback information on when this action will be happen. One possible solution to this problem is to allocate threads in contiguous registers on each processor. So, knowing both the number of threads assigned to one processor *(modulo scheduling)*, the dependency distance between the threads, and by allocating these threads in a sequence order, we can predict the allocation address of the consumer register on the next processor. The predicted address can be stored in the shared register and as soon as a new value is written to that register by a producer thread, the processor first extracts the destination address from the register, and then writes the data directly to that destination.

Another issue is that creating families of microthreads can be implemented recursively. This model has not been implemented here, as we were concerned with the basic support structures. A recursive model is an incremental improvement that would require a family table in the scheduler to hold information on concurrent families, such as global base address, rather than using a fixed creating environment (the first 32 registers). Work is being undertaken at Amsterdam University on recursive thread models and the first paper reporting this development can be found in [51]. Such a model has the potential to provide maximal concurrency in this paradigm and should be easier to compile to.

8.2.4 Toward Microgrid CMP Fault-Tolerant Communication

With the rapidly increasing complexity of parallel architectures, the probability of system failures increases as well. CMP systems have many more potential sources of failure than a single processor system. Thus a failure in one processor on chip may cause the entire system to fail. There are several groups targeting fault tolerance in both software and hardware of the CMP, but this issue is outside of the scope of this thesis. However, the microthreaded microgrid CMP provides many opportunities and advantages for developing fault tolerant system and the ring configuration of processors means that it is possible to eliminate one processor in the case of a failure. This requires a support mechanism to monitor the scheduling and execution on each processor. Another possible fault-tolerance mechanism can be applied in the arbiter described in this thesis (see chapter 6), where the arbiter can be provided with extra signals to avoid any failure in the token movement or inter-modules communication.

Bibliography

[1] Ungerer, T., Robec, B. and Silc, J. (2003) A Survey of Processors with Explicit Multithreading. ACM Computing Surveys, vol. 35, pp. 29-63.

[2] Barroso, L. A. et al. (2000) Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. Proc. of 27th Annual International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, pp. 282-293.

[3] Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. and Olukolun, K. (2000) The Stanford Hydra CMP. IEEE Micro, vol. 20, March-April, pp. 71-84.

[4] Hammond, L., Nayfah, B. A. and Olukotun, K. (1997) A Single-Chip Multiprocessor, IEEE Computer Society. vol. 30, no. 9, September, pp. 79-85.

[5] Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H. and Sinharoy, B. (2002) Power4 System Micro-architecture. IBM Journal of Research and Development, vol. 46, no. 1, pp. 5-25.

[6] Kongetira, P., Aingaran, K. and Olukotun, K. (2005) Niagara: 32-way Multithreaded Sparc Processor. IEEE Computer Society, vol. 25, no.2, March-April, pp. 21-29.

[7] McNairy, C. and Bhatia, R. (2005) Montecito: A Dual-Core, Dual-Thread Itanium Processor. IEEE Computer Society, vol. 25, no. 2, March-April, pp. 10-20.

[8] Jesshope, C. R. (2004) Scalable Instruction-level Parallelism. In Computer Systems: Architectures, Modeling and Simulation. 3rd and 4th International Workshops, SAMOS 2004, Samos, Greece, 19-21 July, pp. 383-392. LNCS 3133, Springer.

[9] Bhandarkar, D. (2003) Billion Transistor Chips in Mainstream Enterprise Platforms of the Future. Proc. of the 9th International Symposium on High-Performance Computer Architecture, Anaheim, California, 08-12 February, pp. 3. IEEE Computer Society, Washington, DC, USA.

[10] Agarwal, V., Hrishikesh, M. S., Keckler, S. W. and Burger, D. (2000) Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. Proc. of the 27th Annual International Symposium on Computer Architecture, Vancouver, British, Columbia, Canada, June, pp. 248-259. ACM Press New York, NY, USA.

[11] Onder, S. and Gupta, R. (2001) Instruction Wake-up in Wide Issue Superscalars. Proc. of the 7th International Euro-Par Conference Manchester on Parallel Processing, Manchester, UK, 28-31 August, pp. 418-427. Springer-Verlag, London, UK.

[12] Onder, S. and Gupta., R. (1998) Superscalar Execution with Dynamic Data Forwarding. Proc. of the International Conference on Parallel Architectures and compilation techniques, Paris, France, 12-18 October, pp. 130-135. IEEE Computer Society, Washington, DC, USA.

[13] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K. and Chang, K. (1996) The Case for a Single-Chip Multiprocessor. In Proc. of the Seventh International Symposium, Cambridge, MA, September, pp. 2-11. ACM Press, New York, NY, USA.

[14] Palacharla, S., Jouppi, N. P. and Smith, J. (1997) Complexity-effective Superscalar Processors. In Proc. of the 24th International Symposium on Computer Architecture, Denver, Colorado, United States, 01-04 June, pp. 206-218,. ACM Press, New York, NY, USA.

[15] Tullsen, D. M., Eggers, S. and Levy, H. M. (1995) Simultaneous Multithreading: Maximising on Chip Parallelism. Proc. of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 22-24 June, pp. 392-403. ACM Press New York, NY, USA.

[16] Burns, J. and Gaudiot, J. L. (2001), Area and System Clock Effects on SMT/CMP Processors. Proc. of the 2001 International Conference on Parallel Architectures and compilation techniques, Barcelona, Spain, 08-12 September, pp. 211-218. IEEE Computer Society, Washington, DC, USA.

[17] Jesshope, C. R. (2003) Multithreaded Microprocessors Evolution or Revolution. Proc. of the 8th Asia-Pacific Conference ACSAC'2003, Aizu, Japan, 23-26 September, pp. 21-45. LNCS 2823, Springer, Berlin, Germany.

[18] Agarwal, V., Hrishikesh, M. S., Keckler, S. W. and Burger, D. (2000) Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. Proc. of the 27th Annual International Symposium on Computer Architecture, Vancouver, British, Columbia, Canada, June, pp. 248-259.

[19] Shilov, A. (2004) Intel to Cancel NetBurst Pentum 4 Xeon Evolution. http:/www.xbitlabs.com/news/cpu/ display/20040507000306.html, (Accessed 7/1/2005).

[20] Lipasti, M. H. and Shen, J. P. (1997) Superspeculative Microarchitecture for Beyond AD 2000. IEEE Computer Society, vol. 30, no. 9, September, pp. 59-66.

[21] International Technology Roadmap for Semiconductors (2003) http://public.itrs.net, Accessed 20/4/2005.

[22] Rixner, S. et al. (2000) Register Organisation for Media Processing. International Symposium on High Performance Computer Architecture, Toulouse, France, January, pp. 375-386.

[23] Ronen, R. et al. (2001) Coming Challenges in Microarchitecture and Architecture. Proc. IEEE, vol. 89, no. 3, March, pp. 325-340.

[24] Bousias, K. and Jesshope, C. R. (2005) The Challenges of Massive On-chip Concurrency. Proc. ACSAC 2005, Springer, Singapore, (http://staff.science.uva.nl/ jesshope/Papers/ACSAC05.pdf).

[25] Jesshope, C. R. (2001) Implementing an Efficient Vector Instruction Set in a Chip Multi-processor Using Microthreaded Pipelines. Proc. ACSAC 2001, Gold Coast, Queensland, Australia, 29-30 January, pp. 80-88. IEEE Computer Society, Los Alimitos, CA, USA.

[26] Bousias, K., Hasasneh, N. M., and Jesshope, C. R. (2006) Instruction-Level Parallelism through Microthreading -A Scalable Approach to Chip Multiprocessors. British Computer Journal (BCS), vol. 49, no. 2, Mar 01, pp. 211-233.

[27] Hasasneh N. M., Bell, I. M. and Jesshope C. R. (2006), Scalable and Partitionable Asynchronous Arbiter for Microthreaded Chip Multiprocessors. LNCS as Proc Architecture of Computing Systems, vol. 3894, ARCS 2006 (Frankfurt/Main, Germany), March, pp. 252-267.

[28] Hasasneh N. M., Bell I. M., and Jesshope C. R. (2006) Asynchronous Arbiter for Micro-threaded Chip Multiprocessors, to be published, Journal of Systems Architecture (JSA).

[29] Bell, I. M., Hasasneh, N. M. and Jesshope C. R. (2006) Microgrids and Micro-contexts: Support Structures for Microthread Scheduling and Synchronisation. Intl. Journal Parallel Processing, Jan, pp. 1-9, DOI 10.1007/s10766-006-0017-y.

[30] Hasasneh, N. M., Bell, I. M. and Jesshope C. R. (2006) High Level Modelling and Design for a Microthreaded Scheduler to Support Microgrids, Submitted to 2007 ACS/IEEE International Conference on Computer Systems and Applications, AICCSA '2007 May 13-16, 2007, Amman, Jordan.

[31] Zhou, H. and Conte, T. M. (2002) Code Size Efficiency in Global Scheduling for VLIW/EPIC Style Embedded Processors. Technical Report. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC.

[32] Hwang, K. (1993) Advanced Computer Architecture. MIT and McGraw-Hill, New York St. Louis San Francisco.

[33] Klaiber, A. (2000) Transmeta Corporation The Technology Behind Crusoe Processors, http://www.transmeta.com.

[34] Sudharsanan, S., Sriram, P., Frederickson, H. and Gulati, A. (2000) Image And Video Processing Using MAJC 5200. In Proc. of the 2000 IEEE International Conference on Image Processing Vancouver, BC, Canada, 10-13 September, pp. 122-125. IEEE Computer Society, Washington, DC, USA.

[35] Cintra, M. and Torrellas, J. (2002), Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelisation for Multiprocessors, Proc. of the 8th International Symposium on High-Performance Computer Architecture, Boston, MA, USA, February 02-06, pp. 43-54. IEEE Computer Society, Washington, DC, USA.

[36] Cintra, M. Martinez, J. S. and Torrellas, J. (2000) Architecture Support for Scalable Speculative Parallelisation in Shared-Memory Multiprocessors. International Symposium on Computer Architecture, Vancouver, Canada, June, pp. 13-24. ACM Press, New York, NY, USA.

[37] Terechko, A., Thenaff, E. L., Gary, M., Eijndhoven, J. V. and Corporaal, H. (2003) Inter-Cluster Communication Models for Clustered VLIW Processors. Proc. of the 9th International Symposium on High-Performance Computer Architecture, Anaheim, California, 08-12 February, pp. 354-364. IEEE Computer Society, Washington, DC, USA.

[38] Halfhill, T. (1998) Inside IA-64. Byte Magazine, vol. 23, pp. 81-88.

[39] Schlansker, M. S. and Rau, B. R. (2000) EPIC: An Architecture for Instruction-Level Parallel Processors. Compiler and Architecture Research, HPL-1999-111. HP Laboratories Palo Alto.

[40] Alverson, R. et al. (1990) The Tera Computer System. Proc. of the 4th International Conference on Supercomputing Amsterdam, The Netherlands, 11-15 June, pp. 1-6. ACM Press, New York, NY, USA.

[41] Marr, D. T. et al. (2002) Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal, vol. 6, pp. 4-15.

[42] Emer, J. (1999) Simultaneous Multithreading: Multiple Alpha's Performance. In Presentation at the Microprocessor Forum'99, MicroDesign Resources, San Jose, California.

[43] Sohi, G. S., Breach, S. E. and Vijaykumar, T. N. (1995) Multiscalar Processors. Proc. of the 22nd Annual International Symposium on Computer Architecture, S. Margherita Ligure, Italy, 22-24 June, pp. 414-425. ACM Press, New York, NY, USA.

[44] Taylor, M., et al. (1997) The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE MICRO, Mar/Apr 2002.

[45] Waingold, E., et al. (1997) Baring it all to Software: Raw Machines. vol. 30, issue 9, IEEE Computer, pp. 86-93.

[46] Sankaralingam, K. et al. (2003) Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. Proc. of the 30th Annual International Symposium on Computer Architecture (ISCA-30).

[47] Berger, D., et. al. (2004) Scaling to the end of silicon with EDGE architectures. IEEE Computer, vol. 37, no. 7, pp. 44-55.

[48] Swanson, S., Michelson, K., Schwerin, A. and Oskin, M. (2003) WaveScalar. Proc. of the 36th International Symposium on Microarchitecture (MICRO-36 2003), pp. 291.

[49] Jesshope, C. R. (2005) Microgrids - the exploitation of massive on-chip concurrency. (Invited paper, HPC 2004Cetraro, June 2004), In Grid Computing: A New Frontier of High Performance Computing, pp. 203-223, ed. L. Grandinetti, Elsevier, Amsterdam.

[50] Bolychevsky, A., Jesshope, C. R. and Muchnick, V. (1996) Dynamic Scheduling in RISC Architectures. IEE Proc. Computer Digital Techniques, vol. 143, pp. 309-317.

[51] Jesshope C. R. (2006) Microthreading a model for distributed instruction-level concurrency, Parallel processing Letters, vol. 16(2), pp. 209-228.

[52] Sundararaman, K. and Franklin, M. (1997) Multiscalar Execution along a Single Flow of Control. Proc. of the IEEE, International Conference on Parallel Processing, Bloomington, IL, USA, 11-15 August, pp. 106-113. IEEE Computer Society, Washington, DC, USA.

[53] Breach, S. E., Vijaykumar, T. N. and Sohi, G. S. (1994) The Anatomy of the Register File in a Multiscalar Processor. Proc. of the 27th Int'l Symp. Microarchitecture, San Jose, California, United State, 30 November - 02 December, pp. 181-190. ACM Press New York, NY, USA.

[54] Gontmakher, A. and Schuster, A. (2002) Intrathreads: Techniques for Parallelising Sequential Code. 6th Workshop on Multithreaded Execution, Architecture,

and Compilation (MTEAC-6), November, Istanbul (in conjunction with Micro-35).

[55] Bernard, T. Bousias, K., Geus, B. Lankamp, M., Zhang, L. Pimentel, A. Knijnenburg, P.M.W., and Jesshope, C.R. (2006) A Microthreaded Architecture and its Compiler. Proc. 12th International Workshop on Compilers for Parallel Computers (CPC), M. Arenez, R. Doallo, B. Fraguela, and J. Tourino (eds.), pp. 326-340.

[56] Taylor, M. Lee, W., Amarasinghe, S. and Agarwal, A. (2003) Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In Proc. of the International Symposium on High Performance Computer Architecture.

[57] Mahlke, S., Lin, D., Chen, W., Hank, R. and Bringmann, R. (1992) Effective compiler support for predicted execution using the hyperblock. In proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 45-54.

[58] Codrescu, L., Wills, D. S. and Meindl, J. D. (2001) Architecture of the Atlas Chip Multiprocessor : Dynamically Parallelising Irregular Applications. IEEE Computer Society, vol. 50, pp. 67-82.

[59] Diefendorff, K. (1999) Power4 Focuses on Memory Bandwidth: IBM Confronts IA-64, Says ISA not important. Microprocessor Report, vol. 13, pp. 11-17.

[60] Balasubramonian, R., Dwarkadas, S. and Albonesi, D. (2001) Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In Proc. of the 34th International Symposium on Micro-architecture, Austin, Texas, December, pp. 237-248.

[61] Spracklen, L. and Abraham, S.G. (2005) Chip Multithreading: Opportunities and Challenges. Proc. of the 11th Intel's Symposium on High performance Computer Architecture (HPCA-11 2005), San Francisco, CA, USA, February, pp. 248-252.

[62] Ro, W., and Gaudiot, J-L. (2004) SPEAR: A Hybrid Model for Speculative Pre-Execution. Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Eldorado Hotel, Santa Fe, New Mexico, April, pp. 26-30.

[63] Zoppetti, G. M., Agrawal, G., Pollock, L., Amaral, J. N., Tang, X., and Gao, G. R. (2000) Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures. Proc. of the 14th International Conference on Supercomputing, Santa Fe, New Maxico,USA, May, pp. 306-315.

[64] Wilcox, K., and Manne, S. (1999) Alpha Processor: A history of Power issues and a look to the Future. In Cool-chips Tutorial, Held in conjunction with MICRO-32.

[65] Huh, J., Burger D., and Keckler, S.W. (2001) Exploring the Design Space of Future CMPs, In Proc. of International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, September, pp. 199-210.

[66] Preston, R. P. et al. (2002) Design of an 8-wide Superscalar RISC microprocessor with Simultaneous Multithreading. 2002 IEEE International Solid-State Circuits Conference, San Francisco, CA, February, pp. 334-335.

[67] Scott, J. (1998) Designing the Low-Power M-CORE Architecture. Proc. IEEE Power Driven Micro Architecture Workshop at ISCA98, Barcelona, Spain, June, pp. 145-150.

[68] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003) Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. Proc. of the 36th annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, USA, December, pp. 81.

[69] Yingmin, L., Brooks, D., Zhigang, H. and Skadron, K. (2005) Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. Proc. of the 11th IEEE International Symposium on high performance computer architecture (HPCA), San Francisco, CA, USA, February, pp. 71-82.

[70] Kiemb, M. and Choi, K. (2004) Memory and Architecture Exploration with Thread Shifting for Multithreaded Processors in Embedded Systems. Proc. of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, Washington DC, USA, September, pp. 230-237.

[71] Hwang, K. (1993) Advanced Computer Architecture. MIT and McGraw-Hill, New York St. Louis San Francisco.

[72] Deniel, et. al. (1990) The Directory Based Cache Coherence Protocol for the DASH Multiprocessor. Proc. of the 17th Annual International Symposium on Computer Architecture, pp. 14-159.

[73] Nitzberg, B. and Lo, v. (1991) Distributed Shared Memory: A survey of Issues and Algorithms. Computer, Aug., pp. 52-60.

[74] Amamiya, M., and Kawano, T. (1994) Design Principle of Massively Parallel Distributed-Memory Multiprocessor Architecture. Advanced Topics in Dataflow Computing and Multithreading IEEE Press, pp. 1-17.

[75] Rivers, J. A., Tyson, G. S., Davidson, E. S., and Austin, T. M. (1997) On High-Bandwidth Data Cache Design for Multi-Issue Processors. Proc. of the 30th Annual Int'l Symp. on Microarchitecture, December, pp. 46-56.

[76] Hagersten, E., Landin, A. and Haridi, S. (1991) DDM - A Cache-Only Memory Architecture. SICS Research Report R91:19.

[77] Joe, T. (1995) COMA-F: A non-hierarchical Cache Only Memory Architecture. PhD. Thesis, Department of Computer Science, (Stanford University).

[78] Landin, A. and Dahlgren, F. (1996) Bus-based COMA- Reducing Traffic in Shared Bus Multiprocessors. In Proc. 2nd International Symposium on high-performance computer architecture, IEEE computer society, pp. 95-105.

[79] Saulsbury, A., Wilkinson, T., Carter, J. and Landin, A. (1995) An Argument for Simple COMA. Proc. of the 1st IEEE Symposium on high-performance Computer Architecture (HPCA '95), pp. 276-285.

[80] Dahlgren, F. and Torrellas, J. (1999) Cache-Only Memory Architectures, IEEE computer Society, vol. 32, pp. 72-79.

[81] Warren, D. and Haridi, S. (1988) Data Diffusion Machine-a scalable shared virtual memory multiprocessor. In International conference on Fifth Generation Computer Systems, ICOT.

[82] Robinson, J. G., Baxter, D. C. and Gray, J. (1995) Advantages of COMA. Kendall Square Research.

[83] Burkhardt, H. et. al. (1992) Overview of the KSR 1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research.

[84] Kendall Square Research (1992) KSR1, Technical Summary.

[85] Hagersten, E., Haridi, S. and Landin, A. (1992) DDM-A Cache-Only Memory Architecture. Computer, September, pp. 44-54.

[86] Stenstrom, P. Joe, T. and Gupta, A., Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures, Proc. of the 19th International Symposium on Computer Architecture, 1992, pp. 80-91.

[87] Naga, H.-El (1999) MCOMA: A Multithreaded COMA architecture. Ph.D. Thesis, EE Dept., University of Southern California.

[88] Sohi, G. S. and Franklin, M. (1991) High-Bandwidth Data Memory Systems for Superscalar Processors. Proc. of the Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, April, pp. 53-62.

[89] Wilson, K. M., Olukotun, K. and Rosenblum, M. (1996) Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors. Proc. of the 23th Int'l Symp. on Computer Architecture, May, pp. 147-157.

[90] Wilson, K. M. and Olukotun, K. (1997) Designing High Bandwidth On- Chip Caches. Proc. of the 24th Int'l Symp. on Computer Architecture, June, pp. 121-132.

[91] Edmondson, J., et al. (1995) Intel Organisation of the Alpha 21164 a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. Digital Technical Journal Special 10th Anniversary Issue, vol. 7, no. 1, pp. 119-135.

[92] Yeager, K., et al. (1995) R10000 Superscalar Microprocessor. Hot Chips VII.

[93] Yoaz, A., Erez, M., Ronen, R., and Jourdan, S. (1999) Speculation techniques for improving load related instruction scheduling. Proc. Int. Symp. on Computer Architecture.

[94] Neefs, H., Vandierendonck, H., and DeBosschere, K. (2000): A technique for high-bandwidth and deterministic low-latency load/store access to multiple cache banks. Proc. Int. Symp. on High Performance Computer Architecture.

[95] Bhooshan S Thakar and Gyungho Lee (2001) Access Region Cache: A Multi-porting Solution for Future Wide-Issue Processors. Proc. of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD'01).

[96] Austin, T., and Sohi, G. (1996) High-Bandwidth Address Translation for Multiple-Issue Processors. Proc. of the 23rd Annual International Symposium on Computer Architecture, May, pp. 147-157.

[97] Mulder, J. M., Quach, N. T., and Flynn, M. J. (1991) An Area Model For On-chip Memories and its Application. IEEE Journal of Solid-State Circuits, vol. 26, no. 2, Feb., pp. 98-106.

[98] Gupta, S., Keckler, S.W. and Burger, D.C. (2000) Technology Independent Area and Delay Estimates for Microprocessor Building Blocks, Tech. Report TR2000-05, Department of Computer Sciences, The university of Texas at Austin, May, pp. 1-27.

[99] ASIC - austriamicrosystems Support Information Center, http://asic.austriamicrosystems.com/databooks/, Accessed 01/05/2005.

[100] Shivakumar, P. and Jouppi, N.P. (2001) CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Research Report 2001/2(Weston Research Laboratory), Palo Alto, California, USA, Aug. pp. 1-40.

[101] Maurer, D. (1995) Synthesis of a MIPS-I Processor Kernal Using VHDL, Master Thesis, Technische University Wien (Vienna, Austria).

[102] Symphony EDA VHDL Compiler and Simulator, http://www.symphonyeda.com/, Accessed 5/12/2003.

[103] Wulf, W. (1995) Hitting the memory wall: Implications of the obvious, ACM Computer Architecture News, March 1995.

[104] Olukotun, K., Hammond, L. and Willey, M. (1999) Improving the Performance of Speculatively Parallel Applications on the Hydra CMP, The 1999 ACM International Conference on Supercomputing, pp. 21-30.

[105] Arvind, Thomas R.E. (1980) I-structures: An Efficient Data Type for Functional Languages. Technical Report MIT/LCS/TM-210, Laboratory for Computer Science, MIT, Cambridge, MA.

[106] Hennessy, J. and Patterson, D. (2004) Computer Architecture A Quantitative Approach. Morgan Kaufmann,ISBN: 1558606041 Palo Alto, CA, Third Edition.

[107] Badaway, A-H, Aggarwal, A., Yeung, D. and Tseng, C-W (2004) The Efficacy of Software Prefetching and Locality Optimisations on Future Memory Systems. Journal of Instruction-level Parallelism, vol. 6.

[108] Lauterbach, G. and Horel, T. (1999) UltraSPARC-III: designing third generation 64-bit performance. IEEE Micro, vol. 19(3), pp. 56-66.

[109] PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual (1999). IBM Corporation.

[110] Smith, J. E. Hsu, W.-C. (1992) Prefetching in supercomputer instruction caches. Proc. of the 1992 ACM/IEEE conference on Supercomputing, Minneapolis, Minnesota, United States, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 588-597.

[111] A. Smith (1978) Sequential Program Prefetching in Memory Hierarchies. IEEE Computer, Vol. 11, pp. 7-21.

[112] Luk, C-K, and Mowry, T. C. (1998) Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors. 31st Annual ACM/IEEE International Symposium on Microarchitecture, IEEE micro, pp. 182.

[113] Spracklen, L., Chou, Y. and Abraham, S. G. (2005) Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. Proc.

of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11 2005), IEEE Computer Society, pp.225 - 236.

[114] Lee, C., Chen, K. and Mudge, T. (1997) Instruction Prefetching using Branch Prediction Information. In Proc. Intl. Conf. on Computer Design: VLSI in Computers and Processors, pp. 593-601.

[115] Reinman, G., B. Calder, B., and Austin, T. (1999) Fetch Directed Instruction Prefetching. Proc. of the 32nd annual ACM/IEEE international symposium on Microarchitecture, pp. 16-27.

[116] Mutlu, O., Kim, H., Armstrong, D. N., and Patt, Y. N. (2004) Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance. In 16th Symposium on Computer Architecture and High Performance Computing, Oct., pp. 2-9.

[117] Park, I., Powell, M. D. and Vijaykumar, T. N. (2002) Reducing Register Ports for Higher Speed and Lower Energy. Proc. of the 35th annual ACM/IEEE international symposium on Microarchitecture, Istanbul, Turkey, November 18-22, pp. 171-182. IEEE Computer Society, Los Alamitos, CA, USA.

[118] Kim, N. S. and Mudge, T. (2003) Reducing Register Ports using Delayed Write-back Queues and Operand Pre-fetch. Proc. of the 17th annual international conference on Supercomputing, San Francisco, CA, USA, 23-26 June, pp. 172-182. ACM Press, New York, NY, USA.

[119] Tseng, J. H. and Asanovic, K. (2003) Banked Multiported Register Files for High-Frequency Superscalar Microprocessors. In 30th International Symposium

on Computer Architecture, San Diego, California, 09-11 June, pp. 62-71. ACM Press, New York, NY, USA.

[120] Bunchua, S., Wills, D. S. and Wills, L. M. (2003) Reducing Operand Transport Complexity of Superscalar Processors using Distributed Register Files. Proc. of the 21st International Conference of Computer Design, San Jose, California, 13-15 October, pp. 532-535. IEEE Computer Society, Los Alamitos, CA, USA.

[121] SPARC International, Inc. (1994) The SPARC Architecture Manual Version 9, PTR Prentice Hall, Englewood Cliffs, N.J.

[122] Valluri, M. G., and Govindarajan, R. (1999) Evaluating Register Allocation and Instruction Scheduling Techniques in Out-of-Order Issue Processors. Proc. of the 1999 International conference on parallel architectures and compilation techniques (California, USA), pp. 78-83.

[123] Waldspurger, C. A., and Weihl, W. E. (1993) Register Relocation: Flexible Contexts for Multithreading. In 20th Annual International Symposium on Computer Architecture, May, pp. 120-129.

[124] Zhuang, X., Zhang, T. and Panda, S. (2004) Hardware-managed Register Allocation for Embedded Processors. Proc. of the 2004 ACM conference on languages, compilers and tools for embedded systems, pp. 192-201.

[125] Zhuang, X., and Panda, S. (2005) Differential Register Allocation. Proc. of the 2005 ACM SIGPLAN conference on programming languages and design, vol. 40, pp. 168-179.

[126] Villiger, T., Kaslin, H., Gurkaynak, F. K., Oetiker, S. and Fichtner, W. (2003) Self-timed Ring for Globally-Asynchronous Locally-Synchronous Systems. Proc. of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03), Vancouver, BC, Canada.

[127] Shapiro, D. (1984) Globally Asynchronous Locally Synchronous Circuits. PhD Thesis, Report No. STAN-CS-84-1026, Stanford University.

[128] Zhuang, S., Carlsson, W. L., Palmkvist, K. and Wanhammar, L. (2002) An Asynchronous Wrapper with Novel Handshake Circuits for GALS Systems. In Proc. of the IEEE 2002 International Conference on Communications, Circuits and Systems, Cheungdu, China, August, pp. 1521-1525.

[129] International Technology Roadmap for Semiconductors: ITRS (2003). Available at http://public.itrs.net.

[130] Muller, D. E., and Bartky, W. S. (1959) A theory of asynchronous circuits. In Proc. of an International Symposium on the Theory of Switching, Harvard University Press, April, pp. 204-243.

[131] Lewin, D., and Protheroe, D. (1992) Design of Logic Systems. Second Edition, CHARMAN and HALL, ISBN 0 412 42890 3, 0 442 31587 2(USA), Chapter 8.

[132] Unger, S. H. (1969) Asynchronous Sequential Switching Circuits. New York NY: Wiley-Interscience, John Wiley and Sons, Inc., New York.

[133] Bainbridge, W. J. and Furber, S. B. (2001) Delay Insensitive System-on-Chip Interconnect Using 1-of-4 Data Encoding. Proc. of the Seventh International Symposium on Asynchronous Circuits and Systems, pp. 118.

[134] Hauck, S. (1995) Asynchronous Design Methodologies: An Overview. Proc. of the IEEE, vol. 83, January, pp. 69-93.

[135] Takamura, A., Kuwako, M., Imai, M., Fujii, T., Ozawa, M., Fukasaku, I., Ueno, Y. and Nanya, T. (1997) TITAC-2: An Asynchronous 32-bit Microprocessor based on Scalable-Delay-Insensitive Model. Proc. of the 1997 International Conference on Computer Design (ICCD'97), Austin, TX, USA, pp. 288-294.

[136] Martin, A. J., Lines, A., Manohar, R., Nystroem, M., Penzes, P., Southworth, R., and Cummings, U. (1997) The Design of an Asynchronous MIPS R3000 Microprocessor. In Advanced Research in VLSI pp. 164-181.

[137] Guibaly, F. (1989) Design and Analysis of Arbitration Protocols. IEEE Trans. Computers vol. 38, pp. 161-171.

[138] Wilkinson, B. (1992) Comments on Design and Analysis of Arbitration Protocols. IEEE Trans. Computers, vol. 41, pp. 348-351.

[139] Valencia, M., Bellido, M., Huertas, J., Acosta, A. J. and Solano, S. (1995) Modular Asynchronous Arbiter Insensitive to Metastability. IEEE Transaction on Computers, vol. 44, pp. 1456-1461.

[140] Josephs, M. and Yantchev, J. (1996) CMOS Design of the Tree Arbiter Element. Proc. of the IEEE Trans. on VLSI Systems, vol. 4, pp. 472-476.

[141] Rigaud, J-B., Quartana, J., Fesquet, L. and Renaudin, M. (2002) High-Level Modeling and Design of Asynchronous Arbiters for On-Chip Communication Systems. Proc. of the IEEE 2002 Design, Automation and test in Europe Conference and Exhibition, pp. 1090.

[142] Liu, Y. C., and Gibson, G. A. (1984) Microcomputer Systems: The 8086/8088 Family, Prentice-Hall, Englewood Cliffs, NJ.

[143] Macii, E. and Poncino, M. (1995) The Design of Easily Scalable Bus Arbiters with Different Dynamic Priority Schemes. Proc. of 29th Asilomar Conference on Signals, Systems and Computers, vol. 1, Pacific Grove, CA, USA, pp. 211-213.

[144] Moore, S., Taylor, G., Mullins, R. and Robinson, P. (2002) Point to Point GALS Interconnect. Proc. of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC'02), pp. 69-75.

[145] El-Moursy, A., Garg, R., Albonesi, D. and Dwarkadas, S. (2005) Partitioning Multithreaded Processor with a Large Number of Threads. International Symposium on Performance Analysis of Systems and Software.

[146] Wang, L., and Selvaraj, H. (2003) A Scheduling and Partitioning Scheme for Low Power Circuit Operating at Multiple Voltage. Proc. of the Euromicro Symposium on Digital System Design (DSD'03), Belek-Antalya, Turkey, pp. 144-147.

[147] Paterson, C. et al. (2002) System-on-Chip for High Speed Communication Systems. Intel Corporation and Synopsys Inc. Report, pp. 1-25

[148] Silberman, J. et al. (1998) A 1.0 GHz single issue 64b PowerPC integer processor. ISSCC, Department of Computer Sciences, IBM Austin Research Lab., Austin, Tx, pp. 230.

[149] Lopez, D., Llosa, J., Valero, M. and Ayguade, E. (1998) Resource Widening Versus Replication: Limits and Performance-Cost Trade-Off. 12th International Conference on Supercomputing (ICS-12), Melbourne, Australia, pp. 441-448.

[150] Kumar, R., Jouppi, N.P., and Tullsen, D.M. (2004) Conjoined-Core Chip Multiprocessing. Proc. of the 37th annual International Symposium on Microarchitecture (MICRO-37 2004), Portland, Oregon, December, pp. 195-206.

[151] Gao, G. R., Theobald, K. B., Govindarajan, R., Leung, C., Hu, Z., Wu, H., Lu, J., Cuvillo J. d. , Jacquet, A., Janot, V. and Sterling, T. L. (2003) Programming Models and System Software for Future High-End Computing Systems: International Parallel and Distributed Processing Symposium (IPDPS'03), pp. 206.

[152] Sutter, H. and Larus, J. (2005) Software and the Concurrency Revolution. ACM Queue vol. 3, no. 2, September 2005.

Glossary

ALU : Arithmetic logic Unit

AM : Attractive Memory

BMT : Blocked Multi-threading

Brk : Break Instruction

Bsync: Barrier Synchronisation

CACTI: Cache Access and Cycle Time Model

CMP : Chip Multiprocessors

COMA : Cache Only Memory

CQ : Continuation Queue

Cre : Create Instruction

D-cache : Data Cache

DDM : Data Diffusion Machine

EDGE : Explicit Data Graph Execution

EPIC : Explicitly Parallel Instruction Computing

FPU : Floating Point Unit

GALS : Globally Asynchronous Locally Synchronous

GCQ : Global Continuation Queue

HEP : Heterogeneous Element Processor

HT : Hyper-threading

IA-64: Intel Architecture Processor 64-bit

IBM : International Business Machines

I-cache : Instruction Cache

ILP : Instruction Level Parallelism

IMT : Interleaved Multi-threading

IPC : Instruction Per Cycle

ISA : Instruction Set Architecture

Kill : Kill Instruction

KSR-1: Kendall Square Research

LLP : Loop Level Parallelism

L1/L2: Level1/Level2 Cache

MAJC : Microprocessor Architecture for Java Computing

MIMD : Multiple Instruction Multiple Data

μt-CMP: Microthreaded Chip Multiprocessors

NUMA : Non-Uniform Memory Architecture

OOO : Out-of-Order (superscalar processor)

PC : Program Counter

PPC : Power PC

RAU : Register Allocation Unit

RAW : Reconfigurable Architecture Workstation

RISC : Reduce Instruction Set Architecture

Rmc : Registers required per microcontext

SMT : Simultaneous Multithread/Multi-threading

SIA : Semiconductor Industry Association

SISD : Single instruction Single Data

SPMD : Single Program Multiple Data

Swch : Switch Instruction

SPARC: Scalable Processor Architecture

Tmc : Thread per Microcontext

TCB : Thread Control Block

TLB :Transition Lookaside Buffer

TLP : Thread Level Parallelism

TRIPS: The Tera-op, Reliable, Intelligently adaptive Processing System

UMA : Uniform Memory Access

VHDL : VHSIC Hardware Description Language

VHSIC: Very-High-Speed Integrated Circuit

VLIW : Very-long Instruction Word

Appendices

Appendix A

Code generation examples

This appendix provides an analysis and code generation for livermore loop kernels. Note that in all the code that follows, $G0$ is assumed to contain the value 0 and that addressing is assumed to be to word boundaries, whatever a word is. The loops considered included a number of livermore kernels, some that are independent and some that contain loop-carried dependencies. It also includes both affine and non-affine loops, vector and matrix problems, and a recursive doubling algorithm.

Array of partial Products

```
DO 1 i=2,m
   1  X(i) = X(i) * X(i-1)
```

Register allocation

Main

X(1)	$S0

Loop

I	$L0
X(i)	$L1
X(i)'	$S0

Code

```
do1:   .word 2
       .word m
       .word 1
       .word 1
       .word 2
       .word 1
       .word body
       .word 0
Main:  Lw $S0 X($G0)
       Cre loop
       Bsync
body:  Lw $L1 X($L0)
       Mul $S0 $D0 $L1
       Swch
       Sw $S0 X($L0)
       Kill
```

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals	shared
Main 1	3			1
Body m-1/l	3	2l		1
Total	3m	2l	0	l+1

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes

Main 1			1			1		
Body (m-1)	3	1			1	1	1	
Total	3m-3	m-1	1	0	m-1	m	m-1	0

2-D SOR

```
DO 1 i=2,m-1
1       X(i) = X(i-1)-2*X(i) +X(i+1)
```

Register allocation

Main

X(1)	$S0

Loop

i	$L0
X(i)	$L1
X(i)'	$S0
X(i+1)	$L2

Code

```
loop:   .word 2
        .word m-1
        .word 1
        .word 1
        .word 3
        .word 1
        .word body
        .word 0
Main:   Lw $S0 X($G0)
        Cre loop
        Bsync
        Finish
body:   Lw $L2 X+1($L0)
        Lw $L1 X($L0)
        Add $L2 $L2+$D0
        Swch
        Sub $S0 $L2 $L1
        Swch
        Sw $S0 X($L0)
        Kill
```

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	3			1
Body m-2/l	5	3l		1
Total	5m-2	3l	0	l+1

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1			1			1		
Body (m-2)	7	3			1	1	1	
Total	7m-14	3m-3	1	0	m-2	m-1	m-2	

Livermore kernel 1 – Hydro fragment

```
cdir$ ivdep
DO 1 k = 1,m
   1   X(k)= Q + Y(k) * (R * ZX(k+10) + T * ZX(k+11))
```

Register allocation

Main

Q	$G1
R	$G2
T	$G3

Loop

Temp1	$L1
Y(k)	$L2
ZX(k+10)	$L3
ZX(k+11)	$L4
Temp2	$L5

```
            .data
do1:        .word 1
            .word m
            .word 1
            .word 0
            .word 6
            .word 0
            .word body
            .word 0
main:       Lw $G1 Q
            Lw $G2 R
            Lw $G3 T
            Cre do1
            Bsync
            Finish
Body:       Lw $L4 ZX+11($L0)
            Lw $L3 ZX+10($L0)
            Lw $L2 Y($L0)
            Mul $L1 $L4 $G3
            Swch
            Mul $L5 $L3 $G2
            Swch
            Add $L1 $L1 $L5
            Mul $L1 $L1 $L2
            Swch
            Add $L1 $L1 $G1
            Sw $L1 X($L0)
            Kill
```

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	5		3	
Body m/l	9	6		
Total	9m+5	6l	3	0

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1				3				
Body m	12	8	3		0	0	0	
Total	12m	8m	3m	3	0	0	0	

Livermore kernel 2 – ICCG (incomplete Cholesky decomposition)

```
1002   II= m/2
       IPNTP= 0
 222   IPNT= IPNTP
       IPNTP= IPNTP+II
       II= II/2
       i= IPNTP+1
cdir$ ivdep
c:ibm_dir:ignore recrdeps (x)
DO 2 k= IPNT+2,IPNTP,2
       i= i+1
   2  X(i)= X(k) - V(k) * X(k-1) - V(k+1) * X(k+1)
      IF( II.GT.1) GO TO 222
```

Register allocation

Main

m/2	$G1
i	$G2
II	$L1
IPNTP	$L2
IPNT	$L3
IPNT+2	$L4
1	$L5

Do2

X(k/i)	$L1
X(k-1)	$L2
X(k+1)	$L3
V(k)	$L4
V(k+1)	$L5
i-local	$L6

```
        .data
do2:    .word 0                 #gets written by main thread
        .word 0                 #gets written by main thread
        .word 2
        .word 0
        .word 5
        .word 0
        .word body
        .word 0
main:   Mv $L1 $G1
        Mv $L2 $G0
        Mv $L5 1
loop:   Mv $L3 $L2
        Add $L4 $L3 2
        Add $L2 $L2 $L1
        Div $L1 $L1 2
        SW $L4 do2($G0)             #writes control block for loop start
        SW $L2 do1+1(G0)        #writes control block for loop limit
        Bsync
        Add $G2 $L2 1
        Cre do2
        Bgt $L1 $L5 Loop
        Finish
body:   Lw $L1 X($L0)
        Lw $L2 X-1($L0)
        Lw $L3 X+1($L0)
        Lw $L4 V($L0)
        Lw $L5 V+1($L0)
        Add $L6 $G2 $L0
        Mul $L2  $L2 $L4
        Swch
        Mul $L3 $L3 $L5
        Swch
        Sub $L1 $L1 $L2
        Swch
        Sub $L1 $L1 $L3
        Sw $L1 X($L6)
        Kill
```

Analysis

Iteration multipliers (analysis in spreadsheet)

Sum inner loops	m-logm-1
outer loop	logm-1
Setup	1

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	3	5	2	
Outer logm-1/-	13			
inner m-logm-1/1	11	7		
Total	11m+2logm-21	7l+5	2	0

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1		3	2					
Outer logm-1	10	4	2	1	0	0	0	
Inner m-logm-1	16	10	1	0	0	0	0	
Total	16m-6logm-26	10m-6logm-11	M+logm-1	logm-1	0	0	0	

Loop 3 - Inner product

```
Q=0
DO 3 k= 1,m
   2   Q= Q + Z(k) * X(k)
```

Register allocation

Main

Q	$S0

Loop

Q	$S0
Z	$L1
X	$L2

```
        .data
do3:    .word 1
        .word m
        .word 1
        .word 1
        .word 2
        .word 1
        ,word 0
        .word body
        .word last
Main:   Mv $S0 $G0
        Cre do3
        Bsync
        Finish
body:   Lw $L1 Z($L0)
        Lw $L2 X($L0)
        Mul $L1 $L1 $L2
        Swch
        Add $S0 $D0 $L1
        Kill
last:   Lw $L1 Z($L0)
        Lw $L2 X($L0)
        Mul $L1 $L1 $L2
        Swch
        Add $S0 $D0 $L1
        Swch
        Sw $S0 Q($G0)
        Kill
```

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main+last 1	8			1
Body m-1/l	4	3		1
Total	4m+4	3l		l+1

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main+last 1	5	3	2	0	1	2	1	
Body m-1	5	3	0	0	0	1	1	
Total	5m	3m	2	0	1	m+1	m	

Livermoore loop 4 – Banded Linear equation solver

```
        nt= (1001-7)/2
c
 1004 DO 404  k= 7,1001,nt
        lw= k-6
        temp= XZ(k-1)
cdir$ ivdep
    DO   4  j= 5,m,5
        temp  = temp   - XZ(lw) * Y(j)
   4    lw= lw+1
      XZ(k-1)= Y(5) * temp
 404  CONTINUE
```

Register allocation

Main

nt	$L1
k	$L2
temp	$S0/$G1
lw	$S1
Y(5)	$L3
1001	$L4

Do4

XZ(lw)	$L1
Y(j)	$L2
temp	$S0
lw	$S1

```
.data
do4:        .word 5
            .word m
            .word 5
            .word 1
            .word 2
            .word 2
            .word body
            .word last
body:       mv $L1 1001
            Sub $L1 $L1 7
            Div $L1 $L1 2
            Mv $L2 7
loop:       Sub $S1 $L2 6
            Lw $S0 XZ-1($L2)
            Cre do4
            Lw $L3 Y+5($G0)
            BSync
            Mul $G1 $G1 $L3
            Swch
            Sw $G1 XZ-1($L2)
            Add $L2 $L2 $L1
            Bgt $L2 $L4 loop
            Finish
body:       Lw $L1 XZ($D1)
            Lw $L2 Y($L0)
            Mul $L1 $L1 $L2
            Swch
            Add $S1 $D1 1
            Swch
            Sub $S0 $D0 $L1
            Kill
last:       Lw $L1 XZ($D1)
            Lw $L2 Y($L0)
            Mul $L1 $L1 $L2
            Swch
            Add $S1 $D1 1
            Swch
            Sub $G1 $D0 $L1
            Kill
```

Analysis

Iteration multipliers (analysis in spreadsheet)

Sum inner loop	.6(m+1)
Branch loop	3
Setup	1

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	4	4	1	2
Outer 3/-	9			
inner .6(m+1)/l	5	3		2
Total	3m+34	3l+4	1	2l+2

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1	2	4						
Outer 3	8	2	3	1		2		
Inner .6(m+1)	4	3				2	3	
Total	2.4m+ 28.4	1.8m+ 11.8	9	4	0	1.2m+ 6.2	1.8m+ 1.8	

Correction for last iteration is 1write to $S becomes write to $G accounted for above

Loop5 – Tri-diagonal elimination

```
1005 DO 5 i = 2,m
   5   X(i)= Z(i) * (Y(i) - X(i-1))
```

Register allocation

Main

X(1)	$S0

Do5

$L0	i
$L1	Z(i)
$L2	Y(i)
$S0	X(i)

```
            .data
do5:        .word 2
            .word m
            .word 1
            .word 1
            .word 2
            .word 1
            .word body
            .word 0
main:       Lw $S0 X+1($G0)
            Cre do5
            Bsync
            Finish
body:       Lw $L2 Y($L0)
            Lw $L1 Z($L0)
            Sub $L2 $L2 $D0
            Swch
            Mul $S0 $L2 $L1
            Swch
            Sw $S0 X($L0)
            Kill
```

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	8			1
Body m-1/l	4	3		1
Total	5m+3	3l	0	l+1

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1	0	0	1	0	0	1	0	
Body m-1	6	3	0	0	1	1	1	
Total	6m	3m	1	0	m-1	m	m-1	

Loop 6 – general linear recurrence

```
1006 DO  6  i= 2,m
        W(i)= 0.0100d0
     DO  6  k= 1,i-1
        W(i)= W(i)  + B(i,k) * W(i-k)
   6 CONTINUE
```

Register assignment

Main

m	$G1/L1
i	$G2
m*i	$G3
W(i)	$S0
i-1	$L2

Do6

W(i)	$S0
B(i,k)	$L1
W(i-k)	$L2
i,k	$L3
i-k	$L4

```
        .data
do6:    .word 1
        .word 0                     #written in main
        .word 1
        .word 1
        .word 5
        .word 1
        .word body
        .word 0
main:   Mv $L1 $G1   #move forced by 1 bus rule only 1 instruction in O(m^2)
        Mv $G2 2
loop:   Mul $G3 $G2 $L1
        Sub $L2 $G2 1
        Sw $L2 do6+1($G0)           #set loop limit to i-1
        Mv $S0 0
        Cre Do6
        Bsync
        Add $G2 $G2 1
        Ble $G2 $L1 loop
        Finish
body:   Add $L3 $L0 $G2
        Lw $L1 B($L3)
        Sub $L4 $L0 $G2             #$G2 determined
        Lw $L2 W($L4)
        Mul $L1 $L1 $L2
        Add $S0 $D0 $L1
        Swch
        Sw $S0 W($L4)
        Kill
```

Pointer chasing – locate object at position, from Europar paper

```
struct box {
        int next;
        int x1;
        int x2;
        int y1;
        int y2;
        }
struct box start;
struct box locate( int x; int y; start)
{ while (start.next != 0)
        {if (x >= start.x1)
                if (y >= start.y1)
                        if (x <= start.x2)
                                if (y<= start.y2)
                                        return start;     /*match*/
        start := start.next
        }
return -1;
}
```

Register allocation

locate

$S0	Start
$G1	x (as parameter)
$G2	y (as parameter)
$G3	start (as parameter)
$G4	locate (return)
$G7	return address

while

$L1	x/y test coordinates
$L2	local copy of start restriction on 2 global reads
$L3	local copy of next restriction on 2 global reads
$S0	next

```
            .data
while:      .word 1          #start
            .word n          #limit = number of processors
            .word 1          #step
            .word 1          #dependency distance
            .word 4          #locals
            .word 1          #globals
            .word body       #code
            .word last       #optional last iteration
```

```
locate:   Mv $S0 $G3
          cre while          #create a thread to start the while loop
          bsync
          Mv $G4 -1
          Jr  $7
body:     mv $L2 $D0         #get address of this element in list
          swch
          Lw $L3 next($L2)   #get address of next element in list
          beq $L3 $G0 kill   #Kill if  end of list
          Mv $S0 $L3         #otherwise pass on next address and search
          lw $L1 X1($L2)     #get lower X bound of object
          bge $G1 $L1 fail
          swch
          lw $L1 Y1($L2)     #get lower Y bound of object
          bge $G2 $L1 fail
          swch
          lw $L1 X2($L2)     #get upper X bound of object
          ble $G1 $L1 fail
          swch
          lw $L1 Y2($L2)     #get lower Y bound of object
          ble $G2 $L1 fail
          swch
          Mv $G4 $L2
          Jr $G7
fail:     Kill
kill:     Mv $S0 0           #propagate zero to kill remaining threads
          Kill
last:     mv $L2 $D0         #get address of this element in list
          swch
          Lw $L3 next($L2)   #get address of next element in list
          beq $L3 $G0 fail   #Kill if  end of list
          Mv $S0 $L3         #otherwise pass on next address and search
          Cre while          #and as this is last thread create new threads
          lw $L1 X1($L2)     #get lower X bound of object
          bge $G1 $L1 fail
          swch
          lw $L1 Y1($L2)     #get lower Y bound of object
          bge $G2 $L1 fail
          swch
          lw $L1 X2($L2)     #get upper X bound of object
          ble $G1 $L1 fail
          swch
          lw $L1 Y2($L2)     #get lower Y bound of object
          ble $G2 $L1 fail
          swch
          Mv $G4 $L2
```

Jr $G7
fail: Kill

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	5		5	1
Body m/l	14	3		1
Total	14m+5	3l	5	l+1

$G0 not counted

Accesses to registers assumes succeeds after m elements and fails after 2 searches on average before success

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main+succeeds 1	1		2	1		1		
Body m	6	5	3	0		1	1	
Total	6m+1	5m	3m+2	1	0	m+1	m	

Analysis

Iteration multipliers (analysis in spreadsheet)

Sum inner loop $m/2-m^{1/2}/2$
Branch loop $m^{1/2}-1$
Setup 1

Allocation of registers and instructions executed

	N_e	Locals	Globals	Shared
Main 1	2	2	3	1
Outer $m^{1/2}-1$/-	8			
inner $m/2-m^{1/2}/2$/1	7	5		1
Total	$2.5m+6.5m^{1/2}-5$	5l+2	3	l+1

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1		1	1	1				
Outer $m^{1/2}-1$	3	1	5	2		1		
Inner $m/2-m^{1/2}/2$	8	5	2		1	1	1	
Total	$4m-m^{1/2}-1$	$2.5m-1.5m^{1/2}$	$m+4m^{1/2}-4$	$m^{1/2}-1$	$.5m-.5m^{1/2}$	$.5m-.5m^{1/2}-1$	$.5m-.5m^{1/2}$	

Loop 7 equation of state fragment

```
cdir$ ivdep
 1007 DO 7 k= 1,m
      X(k)=    U(k ) + R*( Z(k ) + R*Y(k )) +
     1       T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
     2       T*( U(k+6) + Q*( U(k+5) + Q*U(k+4))))
    7 CONTINUE
```

register Allocation

Main

R	$G1
T	$G2
Q	$G3

Do7

k	$L0
U(k)	$L1
U(k+1)	$L2
U(k+2)	$L3
U(k+3)	$L4
U(k+4)	$L5
U(k+5)	$L6
U(k+6)	$L7
Y(k)	$L8
Z(k)	$L9

```
        .data
do7:    .word 1
        .word m
        .word 1
        .word 0
        .word 7
        .word 0
        .word body
        .word 0
Main:   Lw $G1 R($G0)
        Lw $G2 T($G0)
        Lw $G3 Q($G0)
        Cre do7
        Bsync
        Finish
Body:   Lw $L1 U($L0)
        Lw $L2 U+1($L0)
        Lw $L3 U+2($L0)
        Lw $L4 U+3($L0)
        Lw $L5 U+4($L0)
        Lw $L6 U+5($L0)
        Lw $L7 U+6($L0)
        Lw $L8 Y($L0)
        Lw $L9 Z($L0)
        Mul $L5 $L5 $G3
        Swch
        Add $L5 $L6 $L4
        Swch
        Mul $L5 $L5 $G3
        Add $L5 $L5 $L7
        Swch
        Mul $L5 $L5 $G2
        Mul $L2 $L2 $G1
        Swch
        Add $L2 $L2 $L3
        Swch
        Mul  $L2 $L2 $G1
        Add $L2 $L2 $L4
        Swch
        Mul $L2 $L2 $G2
        Add $L2 $L2 $L5
        Mul $L8 $L8 $G1
        Swch
        Add $L8 $L8 $L9
        Swch
        Mul $L8 $L8 $G2
```

```
Add $L2 $L2 $L8
Add $L2 $L2 $L1
Sw $L2 X($L0)
Kill
```

Analysis

Allocation of registers and instructions executed

	N_e	Locals	Globals
Main 1	5		3
Body m/l	26	10	
Total	26m+5	10l	3

$G0 not counted

Accesses to registers

	$L		$G		$S		$D	
	reads	writes	reads	writes	reads	writes	reads	writes
Main 1			3	3				
Body m	35	25	8	0				
Total	35m	25m	8m+3	3	0	0	0	

Appendix B

Allocation Scheme Source Code and Simulation Results

This appendix provides a VHDL source code and sample of simulation results for the allocation scheme described in chapter 4. In particular we describe the allocation scheme behaviour and the arbiter test bench. A sample of simulation results for different allocation scenarios is presented. Full source code for the allocation scheme is available on the DVD included with this thesis.

B.1 Allocation Scheme Architecture Behaviour

The following code describes the architecture behaviour of the allocation scheme. The allocation scheme includes three main components, which comprises slice logic, registers and flags and these components are available on the DVD attached with this thesis.

B.1.1 Allocation Slice logic Architecture Behaviour

```
Library IEEE; use IEEE.std_logic_1164.all; Use
IEEE.std_logic_unsigned.all; Use STD.TEXTIO.all; use
IEEE.std_logic_textio.all; Use IEEE.std_logic_arith.all; Entity
Slice is generic(
        w               : integer :=31;
        M               : integer :=63;
        S               : integer :=7;
        Slice_id        : integer;
        Tdelay          : time := 4 ns
      );
port   (
        BAin            :in  std_logic_vector(S downto 0);
        SSBin           :in  std_logic_vector(S downto 0);
        CSBin           :in  std_logic_vector(S downto 0);
        CSSin           :in  std_logic_vector(S downto 0);
        SASin           :in  std_logic_vector(S downto 0);
        SSSin           :in  std_logic_vector(S downto 0);
        SASI            :in  std_logic_vector(S downto 0);
        SAin            :in  std_logic;
        Errorin         :in  std_logic;
        Flagin          :in  std_logic;
        Flagprev        :in  std_logic;
        Do_allocate     :in  std_logic;
        Do_release      :in  std_logic;
        clear           :in  std_logic;
```

```
        BAout           :out std_logic_vector(S downto 0);
        SSBout          :out std_logic_vector(S downto 0);
        CSBout          :out std_logic_vector(S downto 0);
        CSSout          :out std_logic_vector(S downto 0);
        SASout          :out std_logic_vector(S downto 0);
        SSSout          :out std_logic_vector(S downto 0);
        SAout           :out std_logic;
        Errorout        :out std_logic;
        Flagout         :out std_logic
      );
End Slice;

Architecture Combination_Alloc of Slice is
constant ZeroWord       :std_logic_vector(S downto 0) := (others=>'0');
signal   Word           : std_logic_vector(S downto 0) := X"01";
type Register_Boundary  is  array (0 to 4) of std_logic_vector(7 downto 0);
signal Allocate_Boundary : Register_Boundary;

Begin Main : process (Do_allocate , Do_release,BAin, SSBin,
                      CSBin,CSSin, SASin, SSSin, SASI, SAin,
                      Errorin,Flagin,Flagprev
                     )
variable i,SSS,CSB,SAS, flag : natural := Slice_id;
variable initial: boolean    := False;
variable currentbase,SSB     :natural ;
variable CSS                 : natural :=0;
```

```
variable Temp                : natural :=1;
variable Temp_reg,Temp_reg2  : natural :=0;
variable t :std_logic_vector(S downto 0);

Begin
if      ( Flagin  ='0' and Flagprev= '0'  and Do_allocate='0'
          and Do_release='0' and slice_id=0 ) then
          CSSout    <= SSSin ;
          SSSout    <= SSSin ;
          SSBout    <= SSBin;
          CSBout    <= CSBin;
          SAout     <= '1';
else if ( Flagin  ='0' and Flagprev= '0'  and
          Do_allocate ='0'and Do_release='0' ) then
          CSSout    <= SSSin ;
          SSSout    <= unsigned(SSSin) + unsigned(word);
          SSBout    <= SSBin;
          CSBout    <= CSBin;
          SAout     <= '1';
else if ( Flagin  ='0' and flagprev='1' and
          Do_allocate ='0' and Do_release='0') then
          CSSout    <= word;
          CSBout    <= conv_std_logic_vector(Slice_id,8);
          SSSout    <= SSSin ;
          SSBout    <= SSBin;
          SAout     <= '1';
```

```
else if (  Do_allocate ='0'  and  Do_release='0' and
           flagin='1' and flagprev='1') then
           CSSout    <= X"00";
           CSBout    <= X"00";
           SSSout    <= SSSin;
           SSBout    <= conv_std_logic_vector( slice_id,8);
           SAout     <= SAin;
else if (  Do_allocate ='0'  and  Do_release='0' and
           flagin='1'and flagprev='0' ) then
           CSSout    <= X"00";
           CSBout    <= X"00";
           SSSout    <= unsigned(SSSin) + unsigned(word);
           SSBout    <=  conv_std_logic_vector(slice_id,8);
           SAout     <= SAin;
end if; end if; end if; end if;
           flagout   <= flagin;
           BAout     <= conv_std_logic_vector(Slice_id+1,8);
           SASout    <= X"00";
end if; if      (( Do_allocate ='1') and (SASI >ZeroWord )
           and Do_release='0' ) then
           t         :=  SASI;
           SASout    <= t;
           flagout   <= '1';
else if (( Do_allocate ='1') and(flagin ='0') and
         ( Do_release='0')  and (SASI = ZeroWord )
           and  ( SASin > ZeroWord )  ) then
```

```
            t        :=  unsigned(SASin) - unsigned(word);
            SASout    <= t;
            flagout   <= '1';
else if (( Do_allocate ='1') and(Do_release='0')
            and SASin= ZeroWord  ) then
            SASout    <= X"00";
            flagout   <= flagin;
end if; end if;
            CSSout    <= X"00";
end if;

if      (( Do_allocate ='0')and  (Do_release='1')
            and (SASI > ZeroWord ) and (SASin= ZeroWord ) ) then
            t  :=  SASI;
            SASout    <= t;
            flagout   <= '0';
else if (( Do_allocate ='0') and(Do_release='1')
            and  (flagin ='1')  and  ( SASI > ZeroWord )
            and (SASin= ZeroWord ) ) then
            t := SASin;
            SASout    <= t;
            flagout   <= '0';
else if (( Do_allocate ='0') and  (flagin ='1') and (Do_release='1')
            and (SASI = ZeroWord ) and  ( SASin > ZeroWord ) ) then
            t  := unsigned(SASin) - unsigned(word);
            SASout    <= t;
```

```
            flagout   <= '0';
else if (( Do_allocate ='0')and(Do_release='0')and(flagin='0'))then
            SASout    <= "00000000";
            flagout   <= flagin;
end if; end if; end if; CSSout  <= "00000000"; end if; if      (
Errorin = '1') then
           Errorout <= '1';
else if ( Flagin  ='1' and SASin > ZeroWord) then
           Errorout <= '1';
else
           Errorout <= '0';
end if; end if; end process main; end Combination_Alloc;
```

B.1.2 Register Architecture Behaviour

```
Library IEEE; Use IEEE.std_logic_1164.all; Use
IEEE.std_logic_unsigned.all; Use STD.TEXTIO.all; use
IEEE.std_logic_textio.all; Use IEEE.std_logic_arith.all; Entity
Register_Size is generic (
          w              : integer :=31;
          M              : integer :=63;
          S              : integer :=7;
          Slice_id       : integer;
          Tdelay         : time := 4 ns

        );
port     (
```

```
        clk             : in    std_logic;
        rst             : in    std_logic;
        initalize       : in    std_logic;
        Req_Size        : in    std_logic_vector(S downto 0);
        Alloc           : in    std_logic;
        Releas          : in    std_logic;
        Selected_Base   : in    std_logic_vector(S downto 0);
        Released_Base   : in    std_logic_vector(S downto 0);
        SA              : in    std_logic;
        Slice_ASI       : out   std_logic_vector(S downto 0)
       );
end Register_Size; Architecture Registers of Register_size is

type Reg_typ is array(M downto 0) of std_logic_vector(S downto 0);
signal  reg: std_logic_vector(S downto 0):=X"00";
Begin
RRR:process( clk,rst, Req_Size, Selected_Base, Releas,
              Alloc,initalize,Released_Base
          )
variable j : natural := Slice_id; Begin if      ( Alloc = '1' ) then
If      ( conv_integer(Selected_Base(7 downto 0))=j) then
          Reg <= Req_Size;
          Slice_ASI <=  Reg;
else
          Slice_ASI <=  "00000000";
end if; else if ( Releas = '1' and Alloc ='0') then
```

```
if  ( conv_integer(Released_Base(7 downto 0))> j) then
          Slice_ASI <=  Reg;
else
          Slice_ASI <=  "00000000";
end if;
else
          Slice_ASI <=  "00000000";
end if; end if; end process; end Registers;
```

B.1.3 Flag Architecture Behaviour

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL; LIBRARY std; USE
std.textio.ALL; Use IEEE.std_logic_arith.all; Entity Flag is generic
(
         w              : integer :=31;
         M              : integer :=63;    --  Max slice size
         S              : integer :=7;
         Slice_id       : integer;
         Tdelay         : time := 4 ns
        );
Port     (
          CLK           :in std_logic;
          RST           :in std_logic;
          Flgin         :in  std_logic;
          Dallocate     :in  std_logic;
          Drelease      :in std_logic;
          Flgout        :out std_logic;
```

```
          Flgprev       :out std_logic
         );
End Flag; Architecture flags of flag is type ram_typ is array(1
downto 0) of std_logic; signal fg: std_logic:='0'; Begin fff:
process(clk,rst, Flgin,  Dallocate ) variable j : natural :=
Slice_id; variable k : natural := Slice_id; Begin if      ( rst =
'1') then
          fg <= '0';
else if ( j = 0  and Dallocate ='0' ) then
          Flgout     <=  '0';
          Flgprev    <= '0';
else if ( Dallocate ='0' and j >0) then
          Flgout     <=  fg;
          Flgprev    <=  fg;
end if; end if; end if; if      ( Dallocate ='1') then
          fg <= flgin;
else if ( Drelease ='1') then
          fg         <= flgin;
end if; end if; end process; end  flags;
```

B.2 Allocation Scheme Test Bench

The following code describes the allocation scheme test bench.

```
Library IEEE; use IEEE.std_logic_1164.all; use
IEEE.std_logic_unsigned.all; use STD.TEXTIO.all; use
IEEE.std_logic_textio.all; use IEEE.std_logic_arith.all;
```

```
Architecture Allocation-Behav of Allocate is

type Slice_Base_array is array (0 to M ) of std_logic_vector(S downto 0);
type Slice_Value_array is array (0 to M ) of std_logic_vector(S downto 0);
constant ZeroWord       : std_logic_vector(S downto 0) := (others =>'0');
signal   BAi            : Slice_Base_array ;
signal   SSBi           : Slice_Value_array;
signal   CSBi           : Slice_Value_array;
signal   CSSi           : Slice_Value_array;
signal   SASi           : Slice_Value_array:=(others  => X"00");
signal   SSSi           : Slice_Value_array;
signal   SASIi          : Slice_Value_array:=(others  =>  X"00");
signal   SAi            : std_logic_vector(M downto 0);
signal   Errori         : std_logic_vector(M downto 0);
signal   Fgi            : std_logic_vector(M downto 0);
signal   Fgo            : std_logic_vector(M downto 0);
signal   Fprev          : std_logic_vector(M downto 0);
signal   init           : std_logic_vector(M downto 0);
signal   t_clk          : std_logic :='0';
signal   sel            :Slice_Base_array ;
Component Flag is
generic(
         w              : integer :=31;
         M              : integer :=63;
         S              : integer :=7;
         Slice_id       : natural;
```

```
        tdelay          : time := 4 ns
      );
Port   (
        CLK             :in std_logic;
        RST             :in std_logic;
        Flgin           :in  std_logic;
        Dallocate       :in  std_logic;
        Drelease        :in std_logic;
        Flgout          :out std_logic;
        Flgprev         :out std_logic
      );
End component;
component Register_Size is generic (
        w               : integer :=31;
        M               : integer :=63;
        S               : integer :=7;
        Slice_id        : natural;
        tdelay          : time := 4 ns
       );
port    (
        clk             : in    std_logic;
        rst             : in    std_logic;
        initalize       : in    std_logic;
        Req_Size        : in    std_logic_vector(S downto 0);
        Alloc           : in    std_logic;
        Releas          : in    std_logic;
```

```
        Selected_Base  : in    std_logic_vector(S downto 0);
        Released_Base  : in    std_logic_vector(S downto 0);
        SA             : in    std_logic;
        Slice_ASI      : out   std_logic_vector(S downto 0)
       );
end component;
Component Slice is
generic (
        w              : integer :=31;
        M              : integer :=63;
        S              : integer :=7;
        Slice_id       : integer:=63;
        tdelay         : time := 4 ns
      );
port  (
        BAin           :in  std_logic_vector(S downto 0);
        SSBin          :in  std_logic_vector(S downto 0);
        CSBin          :in  std_logic_vector(S downto 0);
        CSSin          :in  std_logic_vector(S downto 0);
        SASin          :in  std_logic_vector(S downto 0);
        SSSin          :in  std_logic_vector(S downto 0);
        SASI           :in  std_logic_vector(S downto 0);
        SAin           :in  std_logic;
        Errorin        :in  std_logic;
        Flagin         :in  std_logic;
        Flagprev       :in  std_logic;
```

```
        Do_allocate     :in  std_logic;
        Do_release      :in  std_logic;
        clear           :in  std_logic;
        BAout           :out std_logic_vector(S downto 0);
        SSBout          :out std_logic_vector(S downto 0);
        CSBout          :out std_logic_vector(S downto 0);
        CSSout          :out std_logic_vector(S downto 0);
        SASout          :out std_logic_vector(S downto 0);
        SSSout          :out std_logic_vector(S downto 0);
        SAout           :out std_logic;
        Errorout        :out std_logic;
        Flagout         :out std_logic
   );
End Component;
for all: flag
use entity work.flag(flags);
for all: Register_Size
use entity work.Register_Size(Registers);
for all :Slice
use entity work.Slice(Combination_Alloc);
 Begin
        BAi(0)          <=X"00";
        SSBi(0)         <=X"00";
        CSBi(0)         <=X"00";
        CSSi(0)         <=X"00";
        SSSi(0)         <= X"01";
```

```
        SAi(0)          <= '0';
U1: for i in 0 to (M)  generate
flag0: if i =0 generate
F0 : flag
generic map (
             M ,S ,Slice_id =>i, tdelay => tdelay
            )
Port  map   (
             clk ,rst,  Fgi(i)  , Doallocate, Dorelease,
             Fgo(i),Fprev(i)
            );
end generate flag0;
flagm: if i >0 generate
Fm : flag
generic map (
             M ,S ,Slice_id =>i, tdelay => tdelay
            )
Port  map   (
             clk ,rst,Fgi(i),Doallocate,Dorelease, Fgo(i),Fprev(i)
            ) ;
end generate flagm;
end generate U1;
U2: for i in 0 to M  generate
Reg0: if i =0 generate
R0 : Register_Size
generic map (
```

```
            M ,S ,Slice_id =>i, tdelay => tdelay
           )
Port  map   (
            clk, rst, init(i), Required_Alloc_Size, Doallocate,
            Dorelease,Allocate_Base, Release_Base , SAi(i) ,SASIi(i)
           );
end generate Reg0;
Regn: if i >0 generate
Rn : Register_Size
generic map (
            M ,S ,Slice_id =>i, tdelay => tdelay
           )
Port  map   (
            clk, rst, init(i), Required_Alloc_Size, Doallocate,
            Dorelease,Allocate_Base, Release_Base , SAi(i) ,SASIi(i)
           );
end generate Regn;
end generate U2;
U3: for i in 0 to (M)  generate
slice0: if i =0 generate
S10 : Slice
generic map (
            M,S ,Slice_id =>i, tdelay => tdelay
           )
port map    (
            BAi(i), SSBi(i) ,CSBi(i), CSSi(i), SASi(i),
```

```
            SSSi(i), SASIi(i), SAi(i), Errori(i),Fgo(i),
            Fprev(i),  Doallocate , Dorelease,init(i),
            BAi(i+1),SSBi(i+1),CSBi(i+1) ,CSSi(i+1),SASi(i+1),
            SSSi(i+1), SAi(i+1), Errori(i+1), Fgi(i)
           );
end generate slice0;
slicen: if ((i > 0) and (i < M )) generate
Sln : Slice
generic map (
            M,S ,Slice_id =>i, tdelay => tdelay
           )
port map    (
            BAi(i), SSBi(i) ,CSBi(i), CSSi(i), SASi(i),
            SSSi(i), SASIi(i), SAi(i),Errori(i), Fgo(i),
            Fgo(i-1),Doallocate , Dorelease,init(i),
            BAi(i+1),SSBi(i+1),CSBi(i+1) ,CSSi(i+1),SASi(i+1),
            SSSi(i+1), SAi(i+1), Errori(i+1), Fgi(i)
           );
end generate slicen;
slicefinal: if(i =M) generate
Slfin : Slice
generic map (
            M,S ,Slice_id =>i, tdelay => tdelay
           )
port map    (
            BAi(i), SSBi(i) ,CSBi(i), CSSi(i), SASi(i),
```

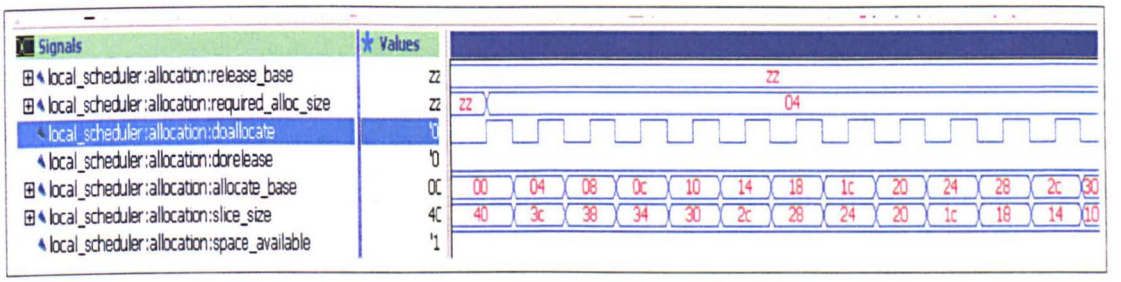


Figure B.1: Simulation waveforms for allocation four registers per thread (Register file size is 64-registers).

```
            SSSi(i), SASIi(i), SAi(i), Errori(i),Fgo(i),
            Fgo(i-1), Doallocate , Dorelease,init(i),
            BAi(i),Allocate_Base,CSBi(i) ,CSSi(i),SASi(i),
            Slice_Size , Space_Available,Input_Error, Fgi(i)
           );
end generate slicefinal; end generate U3; end Allocation_Behav;
```

B.3 Simulation Results

In this section, a different simulation result for the allocation scheme with different allocation scenarios are presented (see figures B.1 to B.5).

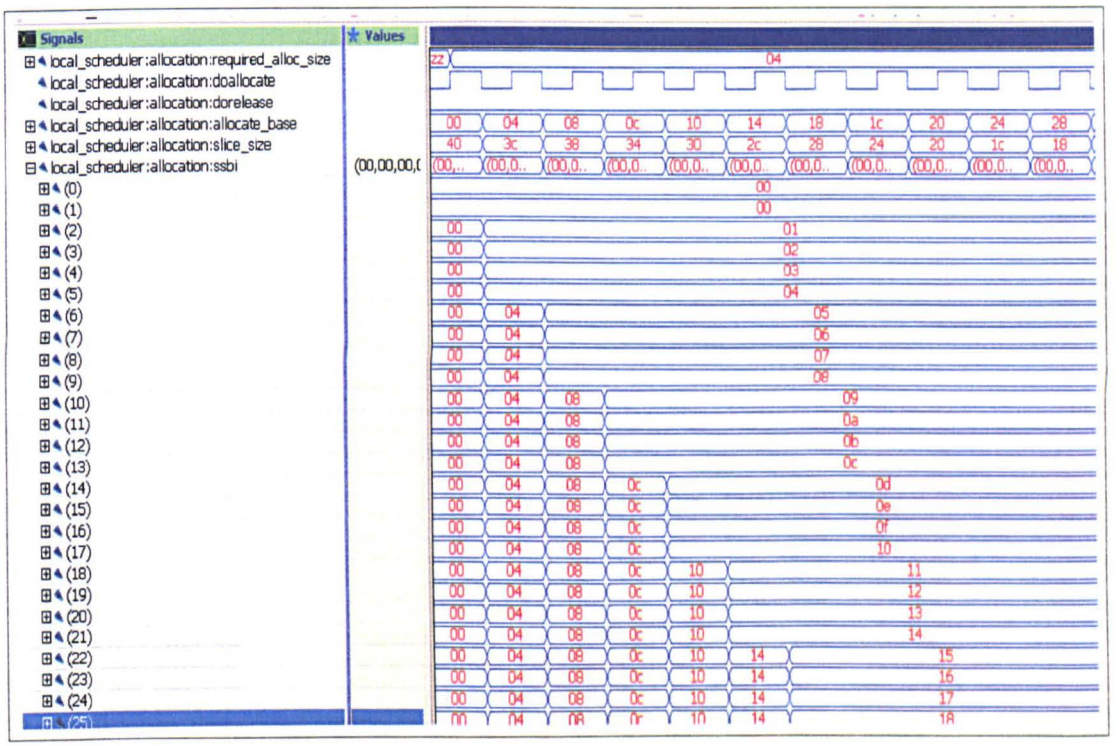


Figure B.2: Simulation waveforms showing slice parameters values, four registers per thread (waveforms sample one).

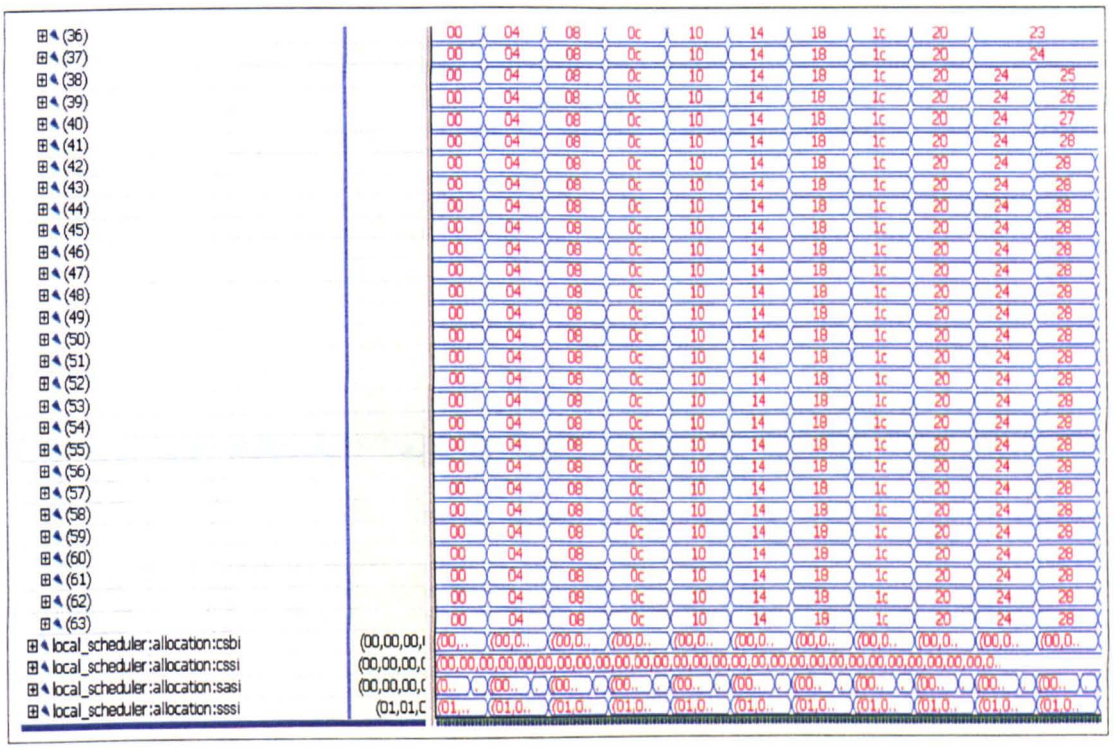
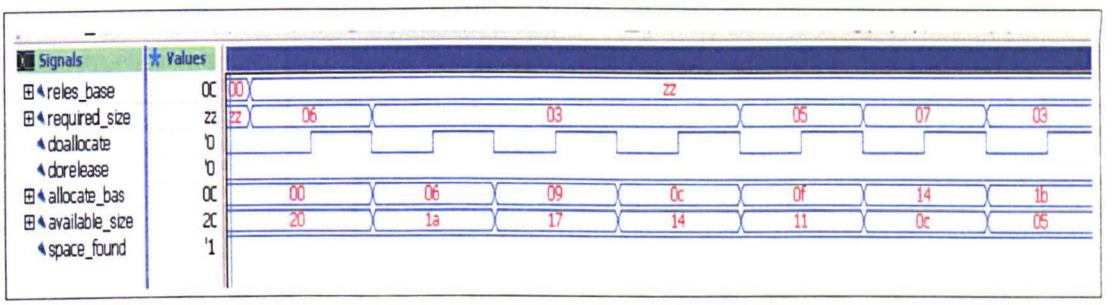
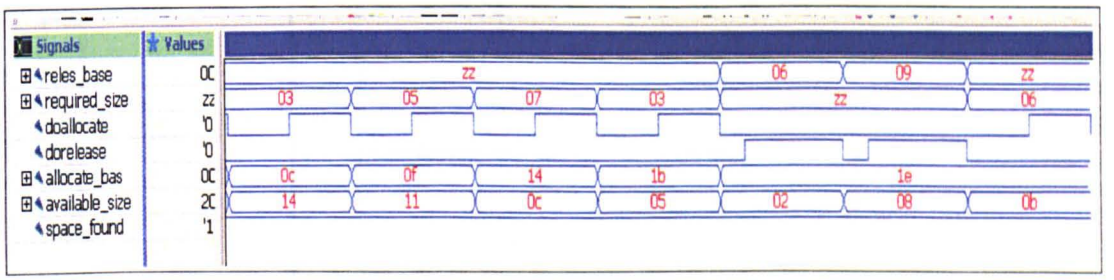


Figure B.3: Simulation waveforms showing slice parameters values, four registers per thread (waveforms sample two).



Waveforms sample one



Waveforms sample two

Figure B.4: Simulation waveforms for allocation and de-allocating different slice sizes per thread (Register file size is 32-registers).

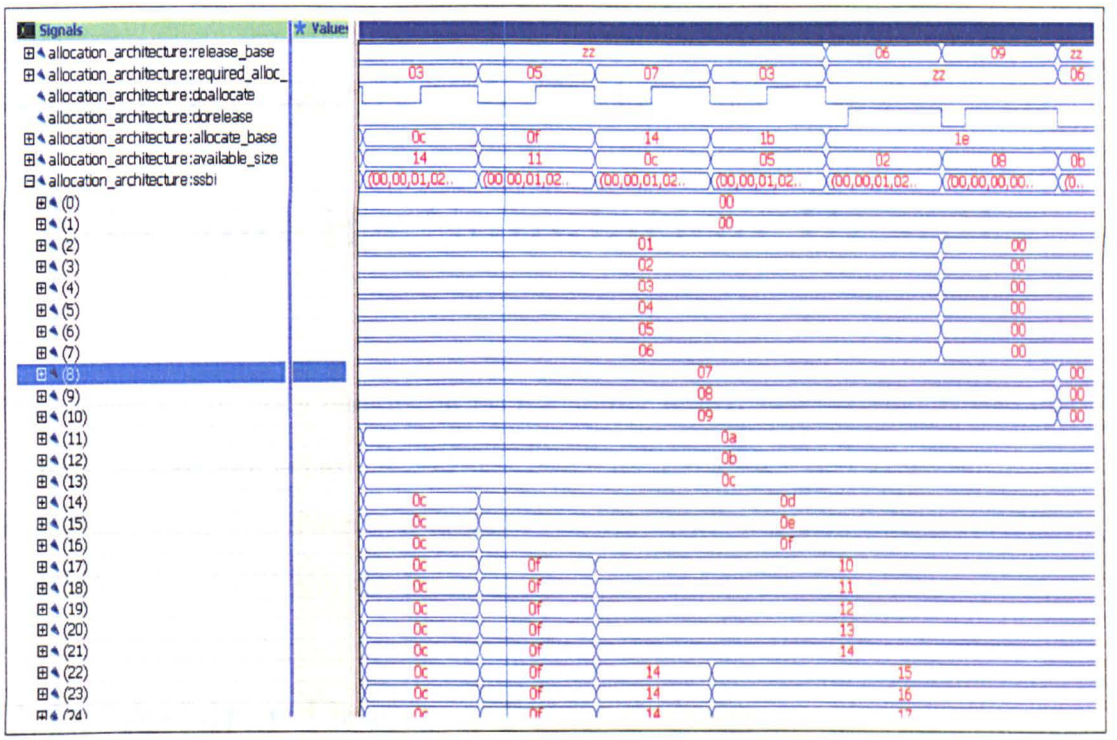


Figure B.5: Simulation waveforms showing slice parameters values, different register sizes per thread.

Appendix C

Asynchronous Arbiter Source Code and its Simulation Results

This appendix provides a description for the arbiter design methodology. It also presents a VHDL architecture behaviour and test bench for the arbiter and provides a sample of simulation results with different scenarios. The full VHDL source code for our arbiter's on the DVD included with this thesis.

C.1 Arbiter Design Methodology

This section continues section 6.4.4 from chapter 6. It describes the arbiter permeative flow table, reduced table with merging rows and a function minimisation. There are eight states, however an asynchronous version of this machine can be minimized. Two states *reset* and *grant priority* (S1 and S2 in the tables below) can be merged, where the elimination of redundant stable states allows us to draw a simplified and minimized state machine. Tables C.1 to C.4 show the arbiter permeative table. It also shows the state reduction and the state minimisation. The minimisation functions can be described as follows:

Table C.1: Arbiter permeative table and state minimisation (snapshot one).

Present State	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0000000	0000001	0000010	0000011	0000100	0000101	0000110	0000111	0001000	0001001	0001010	0001011	0001100	0001101	0001110	0001111
S1	S2	S7														
S2	(S2)		S3		S3						S3		S3			
S3			(S3)													
S4																
S5	S3		S3		S3				(S5)							
S6																
S7		(S7)							S8		S8		S8			
S8									(S8)							
Output1	0000	0000														
Output2	0000		1000		0100						1000		0100			
Output3				1000												
Output4																
Output5	0000		1000		0100				0010							
Output6																
Output7		0000							0010		0010		0010			
Output8									0010							

Present State	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	0010000	0010001	0010010	0010011	0010100	0010101	0010110	0010111	0011000	0011001	0011010	0011011	0011100	0011101	0011110	0011111
S1																
S2	(S2)		S3		S3						S3		S3			
S3																
S4																
S5																
S6																
S7	S8		S8		S8		S8		S8		S8		S8			
S8																
Output1																
Output2			1000		0100						1000		0100			
Output3																
Output4																
Output5																
Output6																
Output7	0010		0010		0010		0010		0010		0010		0010			
Output8																

Inputs are a function of Ain Gin Rhi Rli D B I
Outputs are a function of Rho Rlo Gout Wout

Table C.2: Arbiter permeative table and state minimisation (snapshot two).

Present State	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	0100000	0100001	0100010	0100011	0100100	0100101	0100110	0100111	0101000	0101001	0101010	0101011	0101100	0101101	0101110	0101111
S1																
S2									S8		S8		S8			
S3			S6		S6						S6		S6			
S4																
S5	S3		S3		S3											
S6			(S6)													
S7									S8		S8		S8			
S8	S2		S2		S2											
Output1																
Output2									0010		0010		0010			
Output3			0001		0001						0001		0001			
Output4																
Output5	0000		1000		0100											
Output6			0001													
Output7									0010		0010		0010			
Output8	0000		0000		0000											

Present State	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	0110000	0110001	0110010	0110011	0110100	0110101	0110110	0110111	0111000	0111001	0111010	0111011	0111100	0111101	0111110	0111111
S1																
S2	S8		S8		S8				S8		S8		S8			
S3	S4		S6		S4				S4		S6		S4			
S4					(S4)											
S5																
S6																
S7	S8		S8		S8				S8		S8		S8			
S8																
Output1																
Output2	0010		0010		0010				0010		0010		0010			
Output3	1000		0001		1000				1000		0001		1000			
Output4					1000											
Output5																
Output6																
Output7	0010		0010		0010				0010		0010		0010			
Output8																

Inputs are the function of Ain Gin Rhi Rli D B I
Outputs are a function of Rho Rlo Gout Wout

Table C.3: Arbiter permeative table and state minimisation (snapshot three).

Present State	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	1000000	1000001	1000010	1000011	1000100	1000101	1000110	1000111	1001000	1001001	1001010	1001011	1001100	1001101	1001110	1001111
S1																
S2			S3		S3						S3		S3			
S3																
S4	S5		S5		S5				S5		S5		S5			
S5	S3		S3		S3											
S6	S7		S7		S7				S7		S7		S7			
S7									S8		S8		S8			
S8	S2		S2		S2											
Output1																
Output2			1000		0100						1000		0100			
Output3																
Output4	0010		0010		0010				0010		0010		0010			
Output5	0000		1000		0100											
Output6	0000		0000		0000				0000		0000		0000			
Output7									0010		0010		0010			
Output8	0000		0000		0000											

Present State	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	1010000	1010001	1010010	1010011	1010100	1010101	1010110	1010111	1011000	1011001	1011010	1011011	1011100	1011101	1011110	1011111
S1																
S2			S3		S3						S3		S3			
S3																
S4																
S5																
S6	S7		S7		S7				S7		S7		S7			
S7	S8		S8		S8				S8		S8		S8			
S8																
Output1																
Output2			1000		0100						1000		0100			
Output3																
Output4																
Output5																
Output6	0000		0000		0000				0000		0000		0000			
Output7	0010		0010		0010				0010		0010		0010			
Output8																

Inputs are the function of Ain Gin Rhi Rli D B I
Outputs are a function of Rho Rlo Gout Wout

Table C.4: Arbiter permeative table and state minimisation (snapshot four).

Present State	96 1100000	97 1100001	98 1100010	99 1100011	100 1100100	101 1100101	102 1100110	103 1100111	104 1101000	105 1101001	106 1101010	107 1101011	108 1101100	109 1101101	110 1101110	111 1101111
S1																
S2									S8		S8		S8			
S3																
S4	S5		S5		S5				S5		S5		S5			
S5	S3		S3		S3											
S6	S7		S7		S7				S7		S7		S7			
S7									S8		S8		S8			
S8	S2		S2		S2											
Output1																
Output2									0010		0010		0010			
Output3																
Output4	0010		0010		0010				0010		0010		0010			
Output5	0000		1000		0100											
Output6	0000		0000		0000				0000		0000		0000			
Output7									0010		0010		0010			
Output8	0000		0000		0000											

Present State	112 1110000	113 1110001	114 1110010	115 1110011	116 1110100	117 1110101	118 1110110	119 1110111	120 1111000	121 1111001	122 1111010	123 1111011	124 1111100	125 1111101	126 1111110	127 1111111
S1																
S2	S8		S8		S8				S8		S8		S8			
S3	S4				S4				S4				S4			
S4																
S5																
S6	S7		S7		S7				S7		S7		S7			
S7	S8		S8		S8				S8		S8		S8			
S8																
Output1																
Output2	0010		0010		0010				0010		0010		0010			
Output3	1000				1000				1000				1000			
Output4																
Output5																
Output6	0000		0000		0000				0000		0000		0000			
Output7	0010		0010		0010				0010		0010		0010			
Output8																

Inputs are the function of Ain Gin Rhi Rli D B I
Outputs are a function of Rho Rlo Gout Wout

RHO = Ŕho Ŕlo Ẃout Í B Ď Ŕli Ŕhi Ǵin +

Ŕho Ŕlo Ǵout Ẃout Í Ɓ Ď Rhi Gin +

Ŕho Rlo Ǵout Ẃout Í Ɓ D Rhi Gin +

Rho Ŕlo Ǵout Ẃout Í B Ď Ŕli Ǵin Áin +

Ŕho Rlo Ǵout Ẃout Í B Ď Rli Ŕhi Ǵin +

Rho Rlo Ǵout Ẃout Í B Ď Rli Rhi Ǵin +

Ŕho Ŕlo Gout Ẃout Í B Ď Ŕli Ŕhi Gin +

Rho Ŕlo Ǵout Ẃout Í Ɓ D Ŕli Rhi Gin Áin +

Rho Ŕlo Ǵout Ẃout Í B Ď Ŕli Rhi Ǵin Ain.

RLO = Ŕho Ŕlo Ẃout Í Ɓ D Ŕli Ŕhi Ǵin +

Ŕho Ŕlo Gout Ẃout Í Ɓ D Ŕli Ŕhi Gin +

Rho Rlo Ǵout Ẃout Í Ɓ D Rli Rhi Ǵin +

Ŕho Rlo Ǵout Ẃout Í Ɓ D Rli Ŕhi Ǵin +

Rho Ŕlo Ǵout Ẃout Í Ɓ D Ŕli Rhi Ǵin.

Gout = Ŕho Ŕlo Ǵout Ẃout Í Ŕli Rhi Ǵin Áin +

Ŕlo Ǵout Ẃout Í Ď Rli Ŕhi Ǵin Ain +

Ŕho Ŕlo Ǵout Ẃout Í Ď Rhi Ǵin Ain +

Ŕho Ǵout Ẃout Í Ď Rli Ŕhi Gin +

Ŕlo Ǵout Ẃout Í Ď Ŕli Rhi Gin +

Ŕho Ŕlo Ẃout Í Ɓ Ď Rli Ŕhi Ǵin Áin +

Ŕho Ŕlo Ǵout Ẃout Í B Ď Rli Ǵin Áin +

Ŕho Ŕlo Ǵout Ẃout Í Ɓ D Rli Gin +

Ŕho Ŕlo Ǵout Ẃout Í Ď Ŕli Ŕhi Áin +

Rho Ŕlo Ǵout Ẃout Í Ɓ D Ŕhi Ǵin Ain +

Ŕho Ǵout Ẃout Í Ɓ D Rli Ŕhi Gin Áin +

Ŕlo Ǵout Ẃout Í Ɓ D Ŕli Rhi Gin +

Ŕho Ŕlo Ǵout Ẃout Í Ď Rli Rhi Gin +

Rho Rlo Ǵout Ẃout Í Ď Rli Rhi Gin +

Rho Ŕlo Ǵout Ẃout Í Ď Rli Ŕhi Gin Ain +

Ŕlo Ǵout Ẃout Í Ḃ D Rli Ŕhi Gin Ain +

Rho Rlo Ǵout Ẃout Í Ḃ D Rli Rhi Gin +

Ŕho Ŕlo Ǵout Ẃout Í Ḃ D Rli Rhi Gin +

ŔhoŔlo Ǵout Ẃout Í Ḃ Ď Rli Rhi Ǵin Áin +

ŔhoŔlo Ǵout Ẃout Í Ḃ D Ŕli Rhi Ǵin Ain +

Rho Ŕlo Ǵout Ẃout Í Ḃ D Ŕli Ŕhi Gin Ain +

Ŕho Rlo Ǵout Ẃout Í Ḃ D Rli Ŕhi Gin Ain.

Wout = Ŕho Ŕlo Ǵout Wout Í B Ď Ŕli Ŕhi Gin Áin +

Rho Ŕlo Ǵout Ẃout Í B Ď Ŕhi Gin Áin +

Ŕho Rlo Ǵout Ẃout Í Ḃ D Ŕhi Gin Áin +

Rho Ŕlo Ǵout Ẃout Í B Ď Ŕli Rhi Gin Áin.

C.2 Arbiter Architecture Behaviour

The following code describes the architecture behaviour of the arbiter. The arbiter includes other components such as the processor and clock generator and these component available on the DVD attached to this thesis.

```
Use std.textio.all; Library IEEE; Use IEEE.std_logic_1164.all; Use
IEEE.std_logic_unsigned.all; Use IEEE.std_logic_arith.all; Use
ieee.vital_primitives.all; Entity nand_Block is generic( Delay :
Time
      );
port   (
```

```
        Rh,Rl,G,I,D,B,Ack,nRh,nRl,nGin,nI,nD,nB,nAck:in std_logic;
        TWout,TRH,TRL,TGout      : inout std_logic:='0'
      );
end nand_Block; Architecture  behv of nand_Block is signal
s01,s02,s03,s04,s11,s12,s13,s14,s15,s16,s17,s18,s19,
      s21,s22,s23,s24,s25,s31,s32,s33,s34,s35,s36,s37,s38,
      s39,s310,s311,s312,s313,s314,s315,s316,s317,s318,
      s319,s320,s321,s322       :  std_logic:='0';
signal PWout,PRH,PRL,PGout      : std_logic:='0';
signal nPWout,nPRH,nPRL,nPGout  : std_logic:='1';

Begin
s01  <= not(
           nPRH  and nPRL and nPGout and PWout and nRh and nRl and
           nI and B and nD and G
          );
s02  <= not(
           PRH and nPRL and nPGout and nPWout and nRh and nI and B
           and nD and G and nAck
          );
s03  <= not(
           nPRH and PRL and nPGout and nPWout and nRh  and nI and
           nB and G and D
          );
s04 <=  not(
           PRH and nPRL and nPGout and nPWout and Rh and nRl and
```

```
            nD and nI and G and B
           );
TWout <=not(
            s01 and s02 and s03 and s04
           );
s11  <=  not(
             nPRH and nPRL and nPWout and  nRh and nRl and nGin and
             nI and nD and B
            );
s12  <=  not(
             nPRH and nPRL and nPGout and nPWout and Rh and G and
             nI and nD and nB
            );
s13  <=  not(
             nPRH and PRL and nPGout and nPWout and Rh and nGin and
             nI and D and nB
            );
s14  <=  not(
             nPRH and nPRL and nPGout and nPWout and nRl and nGin and
             nI and nD and B and nAck
            );
s15  <=  not(
             nPRH and PRL and nPGout and nPWout and  nRh and Rl and
             nGin and nI and nD and B
            );
s16  <=  not(
```

```
            PRH and PRL and nPGout and nPWout and  Rh and Rl and
            nGin and nI and nD and B
           );
s17  <=  not(
            nPRH and nPRL and PGout and nPWout and nRh and
            nRl and G and nI and nD and B
           );
s18  <=  not(
            PRH and nPRL and nPGout and nPWout and Rh and
            nRl and G and nI and D and nB and nAck
           );
s19  <=  not(
            PRH and nPRL and nPGout and nPWout and Rh and
            nRl and nGin and nI and nD and B and Ack
           );
TRH  <=  not(
            s11 and s12 and s13 and s14 and s15 and s16 and s17
            and s18 and s19
           );
s21  <=  not(
            nPRH and nPRL and nPWout and nRh  and nRl and
            nGin and nI and D and nB
           );

s22  <=  not(
            nPRH and nPRL and PGout and nPWout and nRh and
```

```
            nRl and G and nI and D and nB
           );
s23  <=  not(
            PRH and PRL and nPGout and nPWout and Rh and
            Rl and nGin and nI and D and nB
           );
s24  <=  not(
            nPRH and PRL and nPGout and nPWout and nRh and
            Rl and nGin and nI and D and nB
           );
s25  <=  not(
            PRH and nPRL and nPGout and nPWout and Rh and
            nRl and nGin and nI and D and nB);
TRL  <=  not(
            s21 and s22 and s23 and s24 and s25
           );
s31  <=  not(
            nPRH and nPRL and nPGout and nPWout and
            Rh and nRl and nGin and nI and nD and nB
           );
s32   <= not(
            nPRL and nPGout and nPWout and nRh and Rl
            and nGin and nI and nD and Ack
           );
s33   <= not( nPRH and nPRL and nPGout and nPWout and
             Rh and Rl and nGin and nI and nD and Ack
```

```
            );
s34    <= not( nPRH and nPGout and nPWout and nRh and Rl
             and G and nI and nD and Ack
            );
s35    <= not(
             nPRL and nPGout and nPWout and Rh and nRl
             and G and nI and nD and nAck
            );
s36    <= not(
             nPRH and nPRL and nPWout and nRh and Rl
             and nGin and nI and nD and nB and nAck
            );
s37    <= not(
             nPRH and nPRL and nPGout and nPWout and
             Rl and nGin and nI and nD and B and nAck
            );
s38    <= not(
             nPRH and nPRL and nPGout and nPWout and
             Rh and Rl and nGin and nI and D and nB
            );
s39    <= not(
             PRH and nPRL and nPGout and nPWout and nRh
             and nRl and G and nI and nD and Ack
            );
s310   <= not(
             PRH and nPRL and nPGout and nPWout and nRh
```

```
            and  nGin and nI and D and nB and Ack
           );
s311  <= not(
            nPRH and nPGout and nPWout and nRh and
            Rl and G and nI and D and nB and nAck
           );
s312  <= not(
            nPRL and nPGout and nPWout and Rh and
            nRl and G and nI and D and nB
           );
s313  <= not(
            nPRH and nPRL and nPGout and nPWout and
            Rh and Rl and G and nI and nD
           );
s314  <= not(
            PRH and PRL and nPGout and nPWout and
            Rh and Rl and G and nI and nD
           );
s315  <= not(
            PRH and nPRL and nPGout and nPWout and
            nRh and Rl and G and nI and nD and Ack
           );
s316  <= not(
            PRL and nPGout and nPWout and nRh and
            Rl and G and nI and D and Ack
           );
```

```
s317  <= not(
            PRH and PRL and nPGout and nPWout and
            Rh and Rl and G and nI and D and nB
           );
s318  <= not(
            nPRH and nPRL and nPGout and nPWout and
            Rh and Rl and G and nI and D and nB
           );
s319  <= not(
            nPRH and nPRL and nPGout and nPWout and Rh
            and Rl and nGin and nI and nD and nB and nAck
           );
s320  <= not(
            nPRH and nPRL and nPGout and nPWout and Rh and
            nRl and nGin and nI and D and nB and Ack
           );
s321  <= not(
            PRH and nPRL and nPGout and nPWout and nRh and
            nRl and G and nI and D and nB and Ack
           );
s322  <= not(
            nPRH and PRL and nPGout and nPWout and nRh and
            Rl and G and nI and D and nB and Ack
           );
TGout <= not(
            s31  and s32 and s33 and  s34 and s35 and s36
```

```
            and s37 and s38 and s39 and s310 and s311 and
            s312 and s313 and s314 and s315 and s316 and
            s317 and s318 and s319 and s320 and
            s321 and s322
           );
Present_State : process ( TWout,TRH,TRL,TGout) Begin
     PWout  <=  TWout;
     PRH    <=  TRH;
     PRL    <=  TRL;
     PGout  <=  TGout;
     nPWout <=  not (TWout);
     nPRH <=    not (TRH);
     nPRL <=    not (TRL);
     nPGout <=  not (TGout);
end process; end behv;
```

C.3 The Asynchronous Arbiter Test Bench

The following code describes the arbiter test bench.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY work;
USE work.std_components.ALL;
Entity test is
end test;
Architecture a of test  is
```

```
constant N :integer:=15;
constant M :integer:=15;
type period_vector is array ( natural range <>) OF time;
constant pr: period_vector(0 to N)   :=( 4 ns,2 ns,6 ns,3 ns,8 ns,7 ns,
                                         1 ns, 4 ns,5 ns,6 ns,1 ns,
                                         9 ns,2 ns,7 ns,9 ns,3 ns
                                       );
constant cpw : period_vector(0 to N) :=( 2 ns, 1 ns,3 ns, 5 ns,4 ns,
                                         3 ns,5 ns, 6 ns,8 ns, 9 ns,
                                         2 ns, 6 ns,2 ns,5 ns,6 ns,7 ns
                                       );
signal t_clk            : std_logic :='0';
signal Wout             : std_logic_vector(N downto 0);
signal reql             : std_logic_vector(N downto 0);
signal reqh             : std_logic_vector(N downto 0);
signal grant            : std_logic_vector(N downto 0);
signal Acq              : std_logic_vector(N downto 0);
signal addr             : std_logic_vector (15 downto 0);
constant cps            : time := 30 ns;
constant processor_id   : natural:=15;
constant transfer_size  : integer:=2;
constant w              : integer:=15;
constant Tdelay         : time:=4 ns;
constant ZeroWord       : std_logic_vector(N downto 0);
signal WRREQ            : std_logic:='0';
signal Ack              : std_logic;
```

```
signal rst              : std_logic;
signal phi1,reset       : std_logic_vector(N downto 0);
signal d                : std_logic_vector(N downto 0);
signal Brk              : std_logic_vector(N downto 0);
signal init             : std_logic_vector(N downto 0);
signal cmp              : std_logic_vector(N downto 0);
signal add_st           : std_logic_vector(N downto 0) ;
signal Acklocal         : std_logic_vector(N downto 0);
signal chip_select      :std_logic;
signal RDWR             :std_logic;
signal Mem_Add          :std_logic_vector (15 downto 0);
signal Mem_DAT          :std_logic_vector (15 downto 0);
signal Mem_clk          :std_logic;
Begin
       add_st           <= (others  => 'Z');
       reql             <= (others  => 'Z') ;
       reqh             <= (others  => 'Z') ;
       grant            <= (others  => 'Z');
       Acq              <= (others  => 'Z');
       rst              <='0';
       Ack              <='Z';
       addr             <= (others =>'Z');
       Mem_DAT          <= "ZZZZZZZZZZZZZZZZ";
microproc: for i in 0 to N  generate
proc0: if i =0 generate cpu0 :CPU
generic map(
```

```
            w,Transfer_size,Processor_id=>i ,N=>N,Tdelay=>Tdelay
           )
port map   (
            phi1(i),rst,d(i),Brk(i),init(i),wout(i),add_st(i),
            addr,Acklocal(i), Acq(N),WRREQ
           );
end generate proc0;
procn: if (i>0) generate cpu1 : CPU
generic map(
            w,Transfer_size,Processor_id=>i ,N=>N,Tdelay=>Tdelay
           )
port map   (
            phi1(i),rst,d(i),Brk(i),init(i),wout(i),add_st(i),
            addr,Acklocal(i),Acq(i-1), WRREQ
           );
end generate procn;
end generate microproc;
cg :  for i in 0 to N generate
C0: if i = 0 generate clk0: clock_gen
generic map(
            period =>pr(i), Tpw =>cpw(i), Tps =>cps
           )
port map   (
            phi1(i), reset(i)
           );
end generate C0; Cn: if (I >0  )generate
```

```
clkn: clock_gen
generic map(
            period => pr(i), Tpw => cpw(i),
            Tps => cps
           )
port map    ( phi1(i), reset(i)
           );
end generate Cn;
end generate cg;
A1:for i in 0 to N generate
P0: if i = 0 generate processor0:Arbiter
generic map(
            w, transfer_size =>8, processor_id =>i,
            N=>N, Tdelay => Tdelay
           )
port map    (
            d(i),Brk(i),init(i),reqh(i),reql(i),
            grant(N), Wout(i),reqh(N),reql(N),
            grant(i),Acq(N),Acq(i),WRREQ,Acklocal(i)
           );
end generate P0 ;
Pn: if (I > 0 ) generate
processorn: Arbiter
generic map(
            w, transfer_size => 8,processor_id =>i,
            N=>N,Tdelay=>Tdelay
```

```
          )
port map  (
           d(i),Brk(i),init(i),reqh(i),reql(i),
           grant(i-1),Wout(i),reqh(i-1),reql(i-1),
           grant(i),Acq(i-1),Acq(i),WRREQ,Acklocal(i)
          );
end generate Pn;
end generate A1;
process(Add_st)
Begin
if (Add_st = ZeroWord) then
WRREQ <=  '0'   ;
else
WRREQ <=  '1' after tdelay ;
end if;
end process;
end a;
```

C.4 Simulation Results

In this section, simulation results for different sizes of arbiter and different demands and brks scenarios are given. Figures C.1 to C.3 show a sample of results from simulating 8 arbiter modules. In this sample the following conditions apply: module 0 has initially reserved the token, module 4 receives a high input on the *Brk* signal line and modules 1, 2, 5, 6, 7 all have high input demand request lines. As illustrated, the request signal RL_1 reaches the token before RH_4, which means that broadcast

bus access is given first to processor 1 (Wout is asserted). When processor 1 releases the token, the grant signals are propagated back to give processor 4 permission to use the broadcast bus before other low priority processors. The rest of the demand requests are granted in sequence order based on position in the ring configuration.

Figures C.4 to C.7 show a sample of results from simulating 16 arbiter modules. In this example the following conditions apply: module 0 has initially reserved the token, module 15 receives a high input on the *Brk* signal line and the other modules have high input demand request lines. As illustrated, the request signal RL_1 and RL_2 reaches the token before RH_15, which means that broadcast bus access is given first to processor 1 then to processor 2 (Wout is asserted). When processor 2 releases the token, the grant signals are propagated back to give processor 15 permission to use the broadcast bus before other low priority processors. The rest of the demand requests are granted in sequence order based on position in the ring configuration.

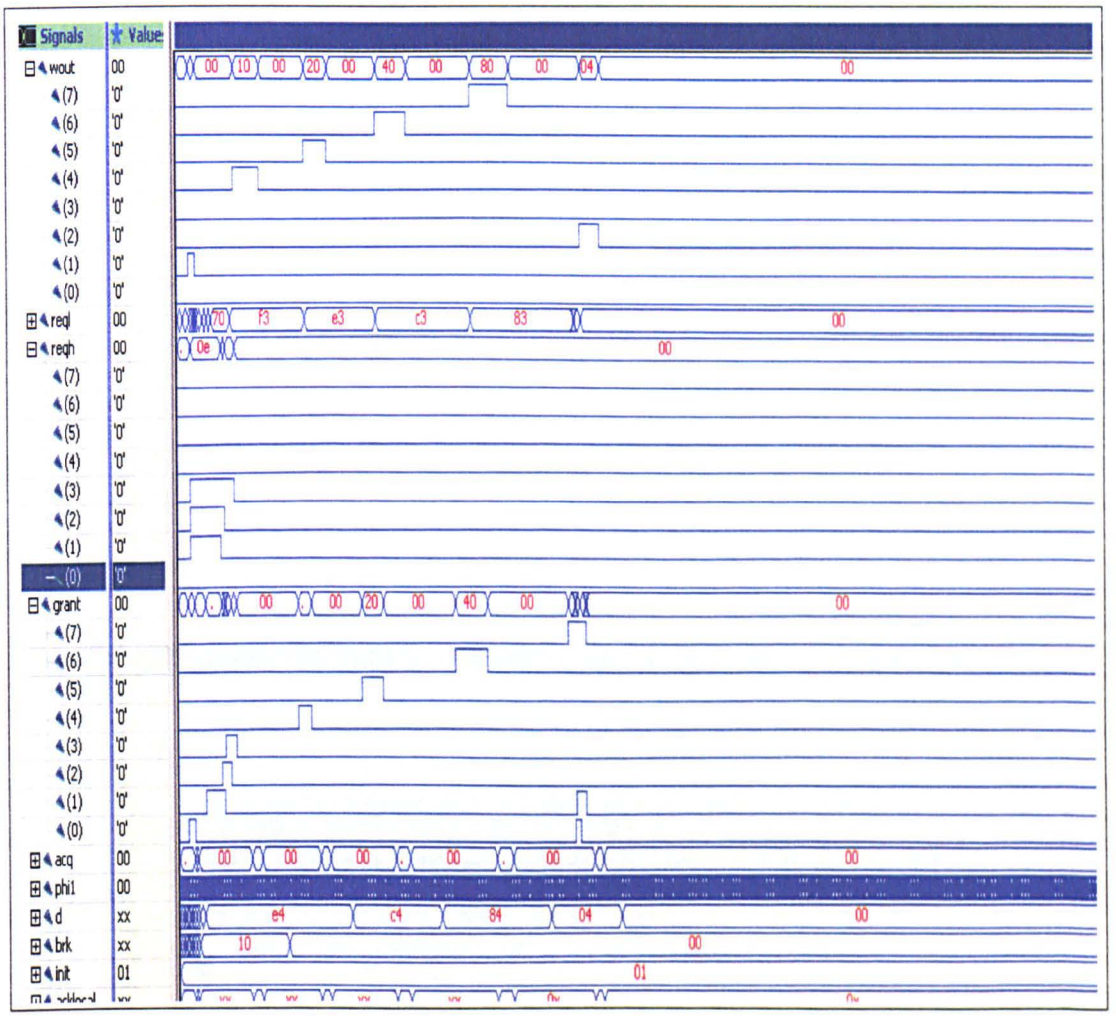


Figure C.1: Simulation waveforms showing arbiter signals, 8 arbiter modules (waveforms sample one).

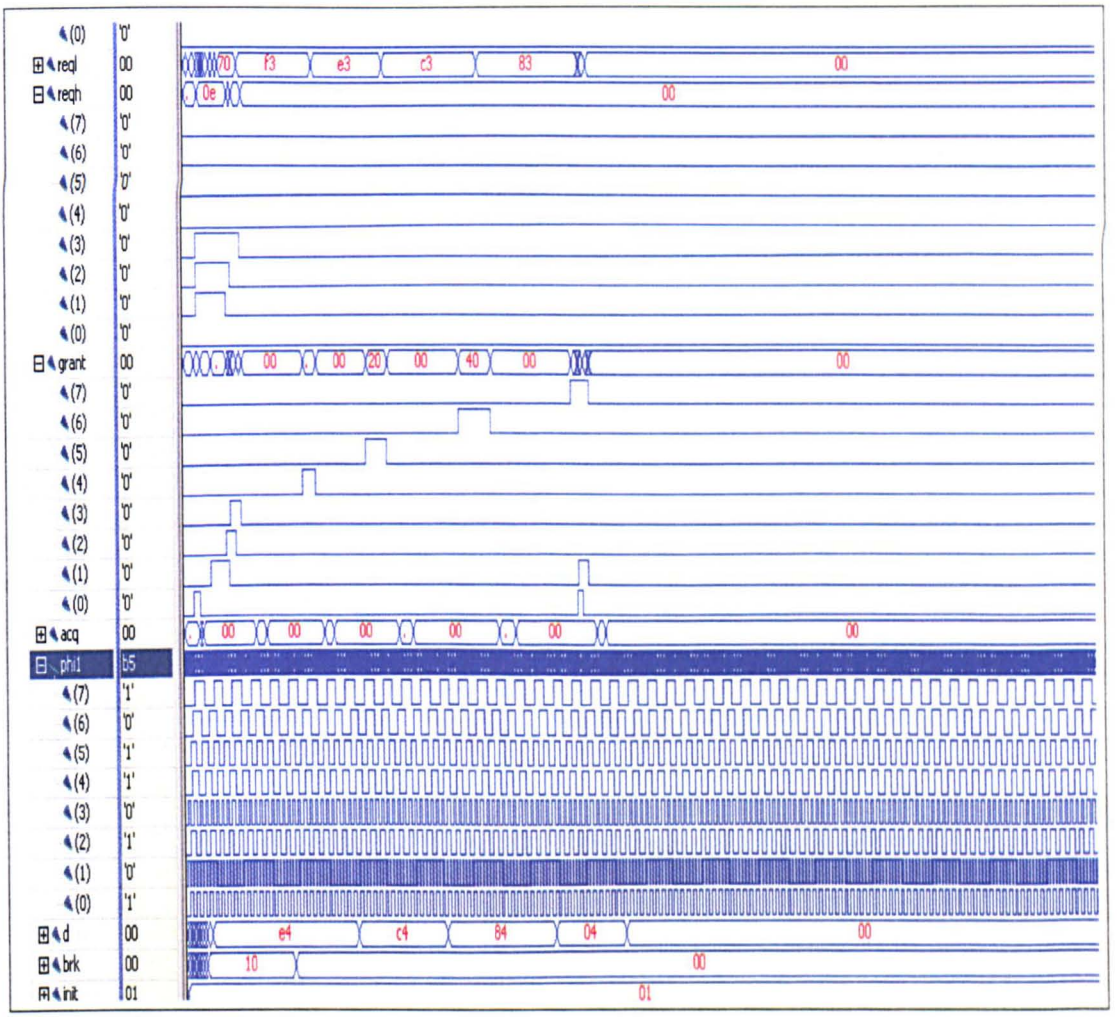


Figure C.2: Simulation waveforms showing arbiter signals, 8 arbiter modules (waveforms sample two).

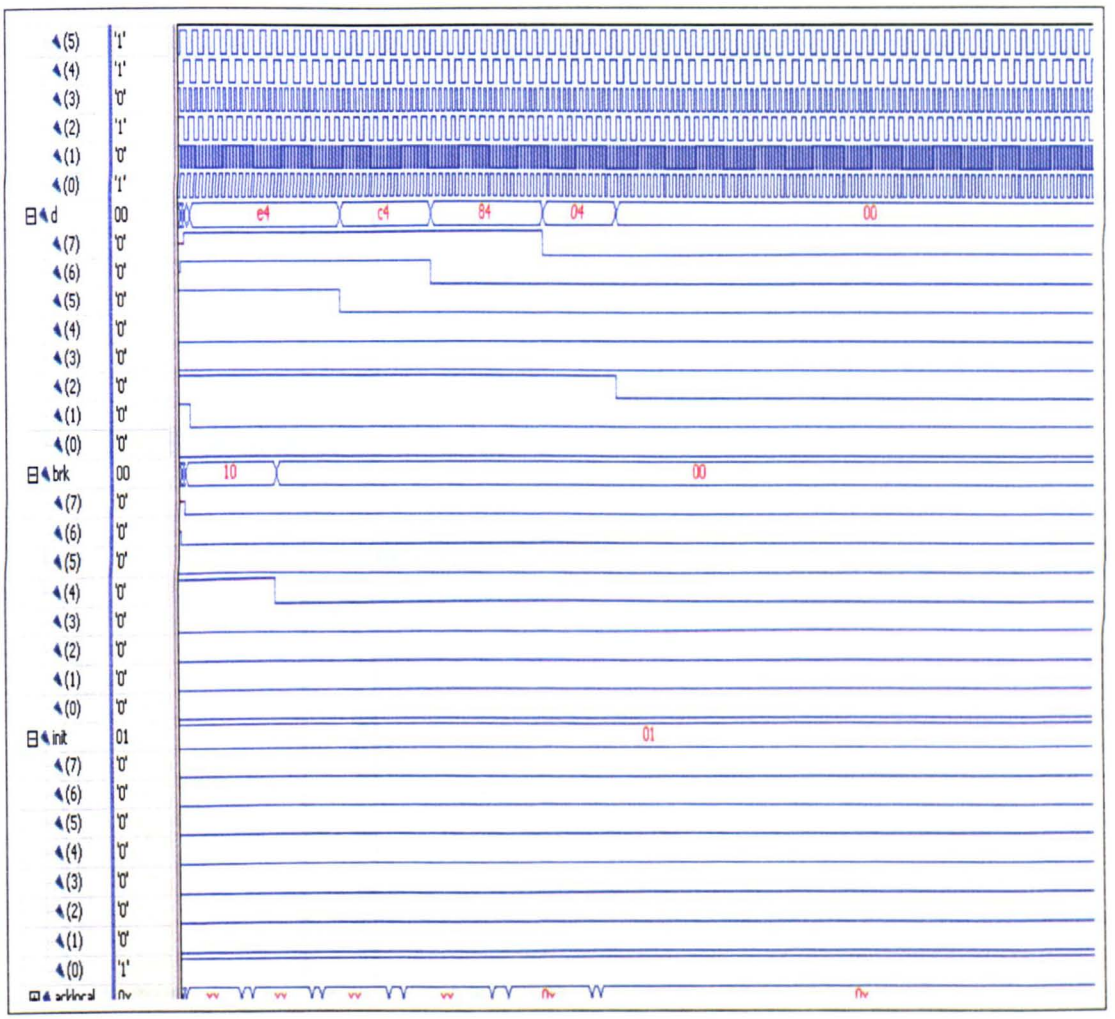


Figure C.3: Simulation waveforms showing arbiter signals, 8 arbiter modules (waveforms sample three).

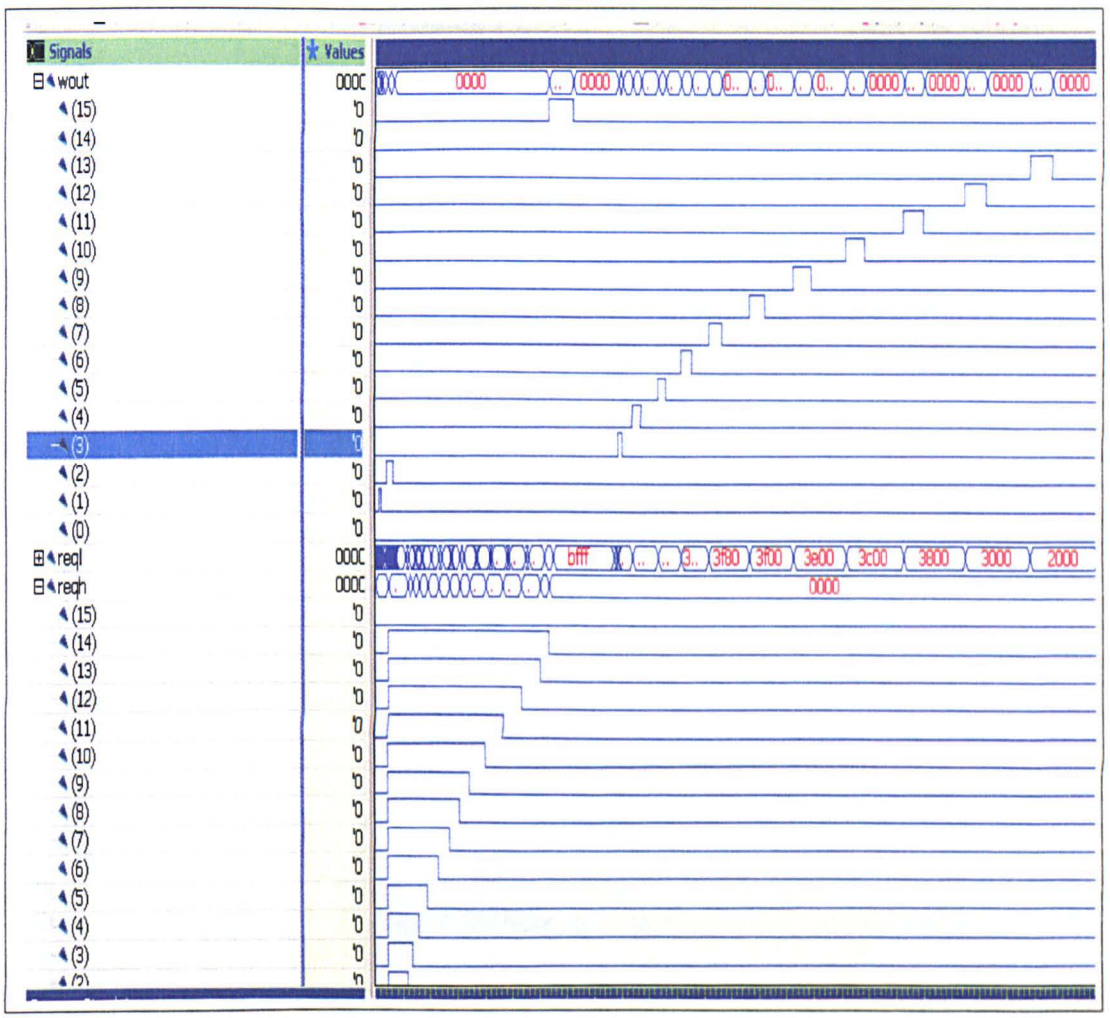


Figure C.4: Simulation waveforms showing arbiter signals, 16 arbiter modules (waveforms sample one).

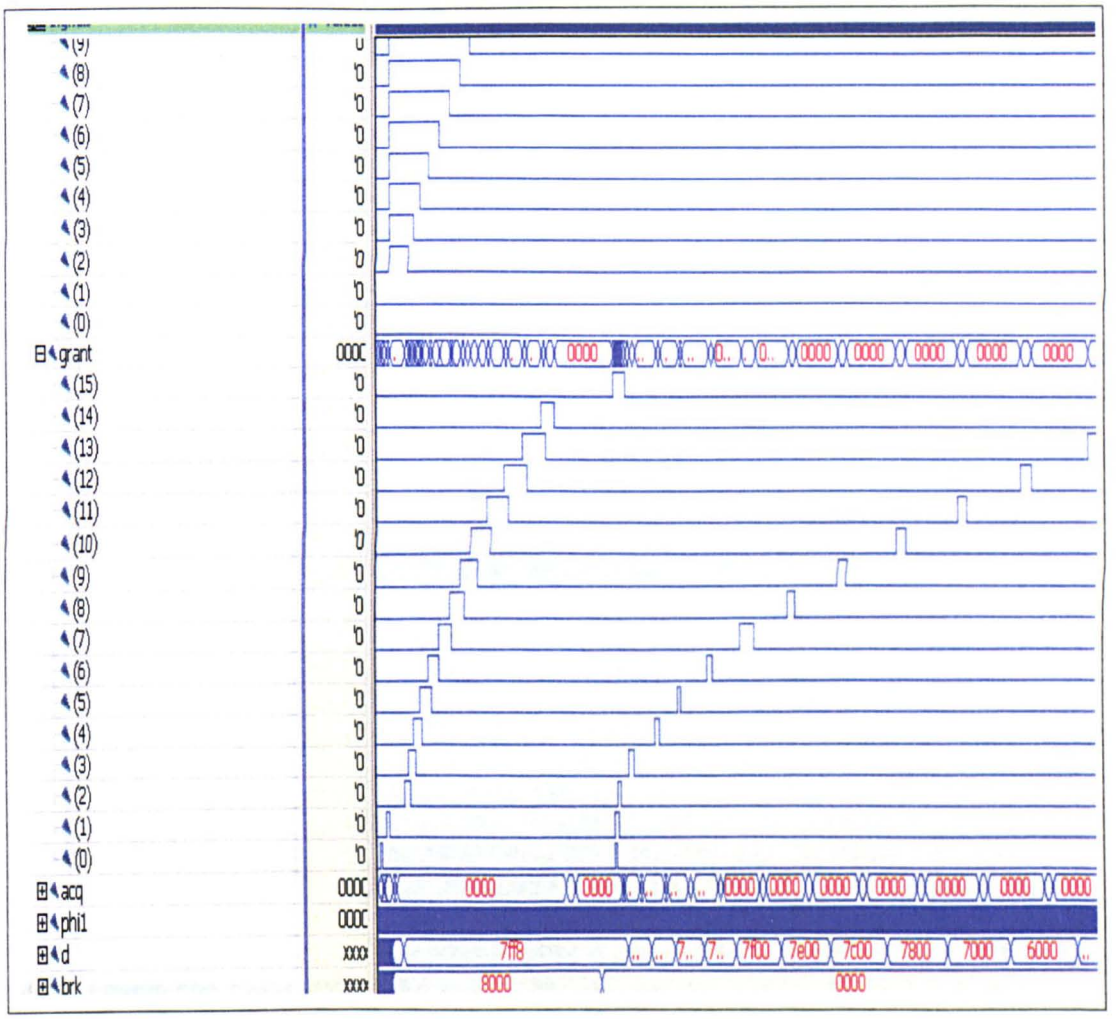


Figure C.5: Simulation waveforms showing arbiter signals, 16 arbiter modules (waveforms sample two).

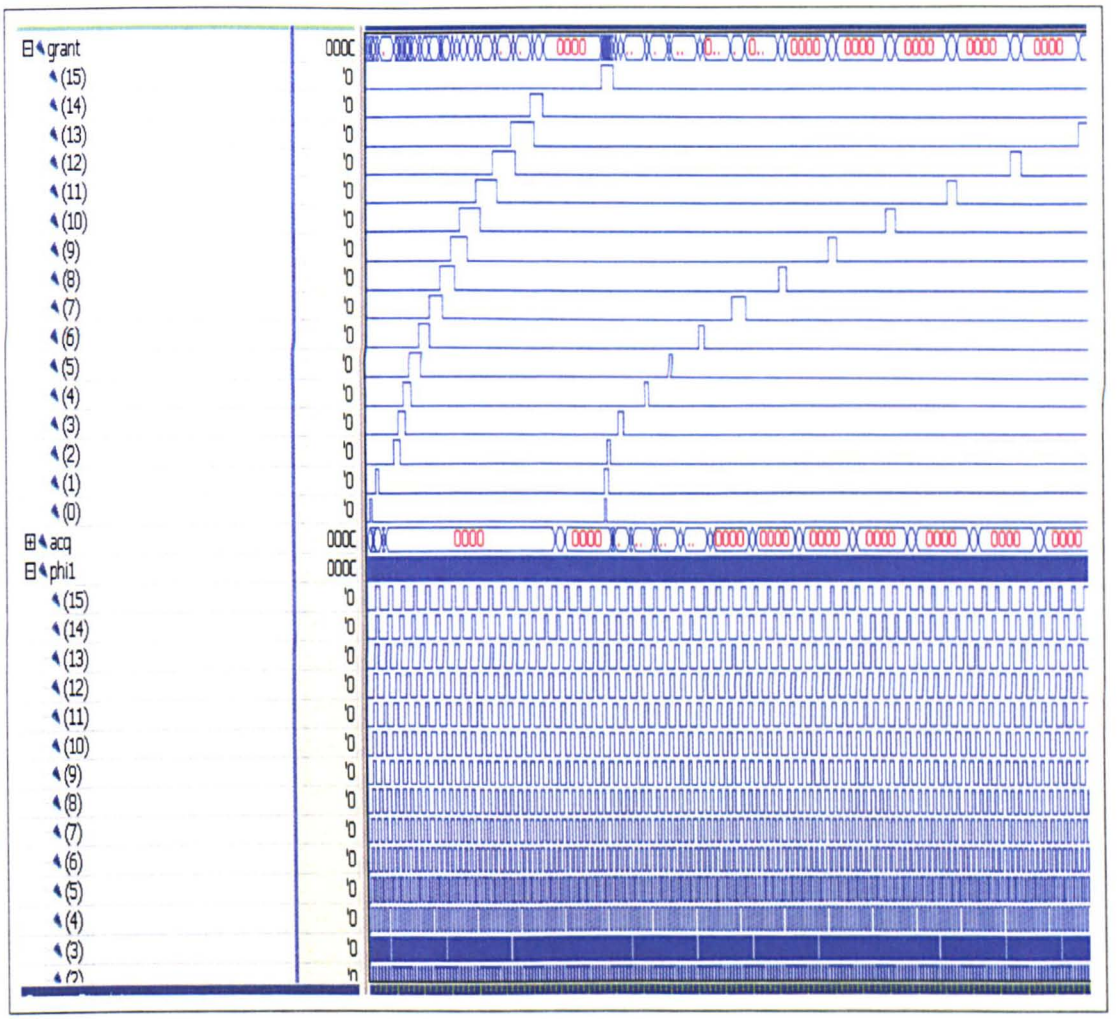


Figure C.6: Simulation waveforms showing arbiter signals, 16 arbiter modules (waveforms sample three).

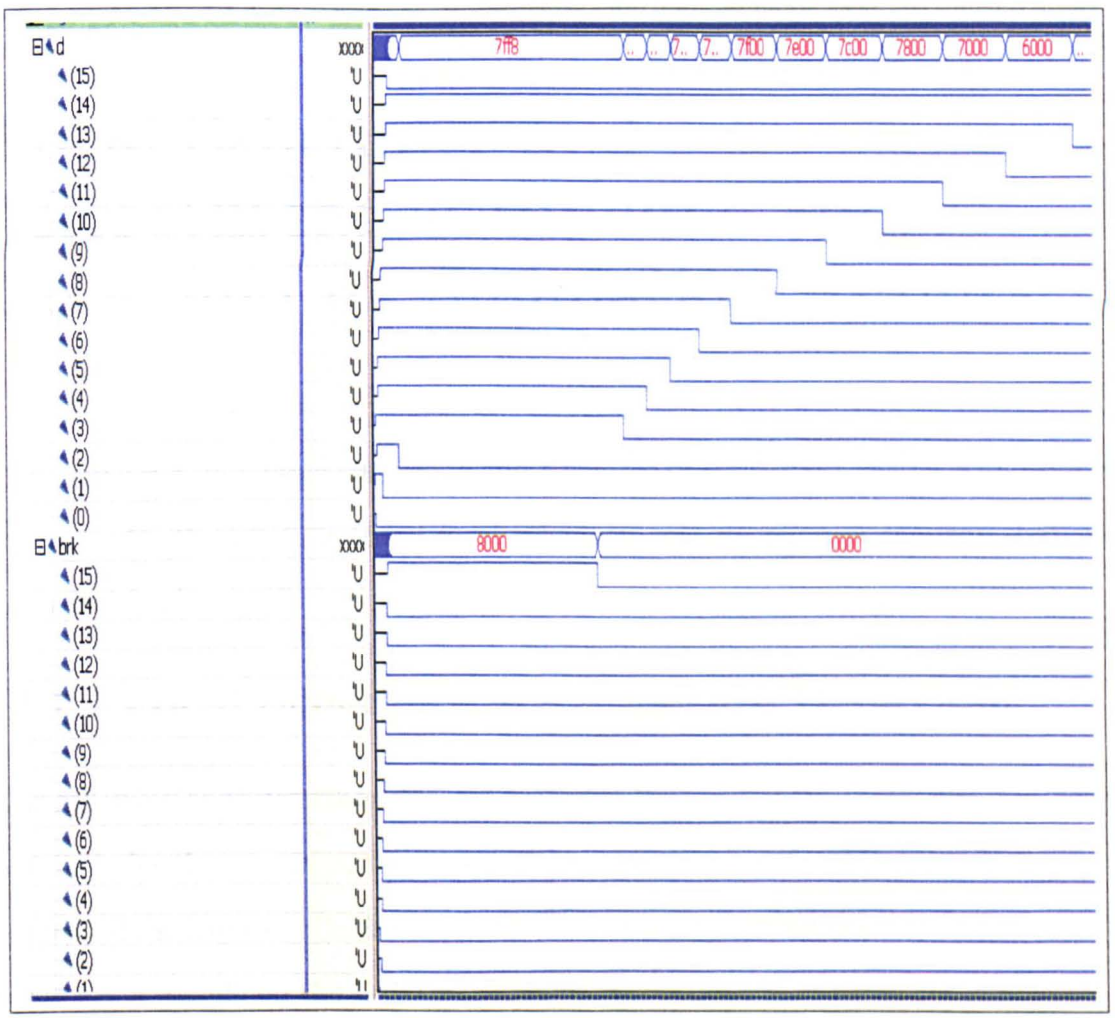


Figure C.7: Simulation waveforms showing arbiter signals, 16 arbiter modules (waveforms sample four).

Appendix D

Local Scheduler and Microthreaded Pipeline Source Code and its Simulation Results

This appendix provides source code and a sample of simulation results with different scenarios for the microthreaded local scheduler and its in-order pipeline that were described in chapter 7.

D.1 Local Scheduler Architecture Behaviour

The following code describes the architecture behaviour of the scheduler. The scheduler includes three main components: allocation scheme (allocate), thread-create and control block (controller), and the CQ. The architecture behaviour of these components are available on the DVD attached to this thesis.

```
Architecture Behaviour of Scheduler is
signal t_clk                :std_logic :='0';
signal Reles_Base           :std_logic_vector(S downto 0);
signal Required_Size        :std_logic_vector(S downto 0);
signal Doallocate           :std_logic;
signal Dorelease            :std_logic;
signal Allocate_Bas         :std_logic_vector(S downto 0);
```

```
signal Available_size      :std_logic_vector(S downto 0);
signal Error_signal        :std_logic;
signal Space_Found         :std_logic;
signal Pointer             :std_logic_vector(7 downto 0);
signal L_Base              :std_logic_vector(7 downto 0);
signal D_Base              :std_logic_vector(7 downto 0);
signal PC_Created          :std_logic_vector(31 downto 0);
signal PC_Reschedule       :std_logic_vector(31 downto 0);
signal F                   :std_logic;
signal producer            :std_logic_vector(7 downto 0);
signal WR_Queue            :std_logic;
Component Control is
generic (
         w                 : integer :=31;
         M                 : integer :=63;
         S                 : integer :=7;
         Slice_id          : integer;
         tdelay            : time := 4 ns
        );
port     (
         CLK               :in    std_logic;
         RST               :in    std_logic;
         Rel_Base          :out   std_logic_vector(S downto 0);
         Alloc_Size        :inout std_logic_vector(S downto 0);
         allocate          :inout std_logic:='0';
         release           :out   std_logic;
         Block_Base        :in    std_logic_vector(S downto 0);
         Block_Size        :in    std_logic_vector(S downto 0);
         Error_in          :in    std_logic;
         Available_Space   :in    std_logic;
         wrreq             :in    std_logic;
         Read_Mem          :out   std_logic;
         Create_Address    :in    std_logic_vector(31 downto 0);
         TCB_Address       :out   std_logic_vector(31 downto 0);
         TCB_data          :in    std_logic_vector(127 downto 0);
         PC_Creat          :out   std_logic_vector(31 downto 0);
         Loc_Base          :out   std_logic_vector(7 downto 0);
         Dp_Base           :out   std_logic_vector(7 downto 0);
         flg               :out   std_logic;
         prod              :out   std_logic_vector(7 downto 0);
```

```
        Write_CQ          :inout std_logic;
        Next_famil        :out   std_logic;
        store_done        :in    std_logic
       );
End component;
Component Allocate is
generic (
        w                 :integer :=31;
        M                 :integer :=63;
        S                 :integer :=7;
        Slice_id          :integer;
        tdelay            :time := 4 ns
       );
port    (
        CLK               :in    std_logic;
        RST               :in    std_logic;
        Release_Base      :in    std_logic_vector(S downto 0);
        Required_Alloc_Size:in   std_logic_vector(S downto 0);
        Doallocate        :in    std_logic;
        Dorelease         :in    std_logic;
        Allocate_Base     :inout std_logic_vector(S downto 0);
        Slice_Size        :out   std_logic_vector(S downto 0);
        Input_Error       :out   std_logic;
        Space_Available   :out   std_logic
       );
End component;
Component CQ is
generic (
        w           : integer :=31;
        M           : integer :=63;
        S           : integer :=7;
        Slice_id    : integer;
        tdelay      : time := 4 ns
       );
port    (
        CLK               :in  std_logic;
        RST               :in  std_logic;
        PC_Created        :in  std_logic_vector(31 downto 0);
        PC_Reschedule     :in  std_logic_vector(31 downto 0);
        Dep_Base          :in  std_logic_vector(7 downto 0);
```

```
        Flag                :in  std_logic;
        produc              :in  std_logic_vector(7 downto 0);
        WR_CQ               :in  std_logic;
        Context_switch      :in  std_logic;
        PC_pipeline         :out std_logic_vector(31 downto 0);
        Local_Base_pipeline :out std_logic_vector(7 downto 0);
        Dep_Base_pipeline   :out std_logic_vector(7 downto 0);
        Slot_Number_pipeline:out std_logic_vector(7 downto 0);
        Prefetch_PC         :out std_logic_vector(31 downto 0);
        Pointer             :in  std_logic_vector(7 downto 0);
        Slot_Number_cache   :out std_logic_vector(7 downto 0);
        Read_cache          :out std_logic;
        Write_Pc            :out std_logic;
        Acknow              :in  std_logic;
        done                :out std_logic
       );
End Component;
for Controller: Control
use entity work.Control( Behav);
for  Allocation:   Allocate
use entity work.Allocate(Allocation_Behav);
 for Continuation_Queue:  CQ
   use entity work.CQ(CQBehav);
Begin
Reles_Base        <= (others  => 'Z');
Allocate_Bas      <= (others  => 'Z');
Available_size    <= (others  => 'Z');
Prefetch_PC       <= (others  => 'Z');
Slot_Number_pip   <= (others  => 'Z');
Slot_Number_cache <= (others  => 'Z');
PC_pipeline       <= (others  => 'Z');
L_Base_pip        <= (others  => 'Z');
D_Base_pip        <= (others  => 'Z');
producer          <= (others  => 'Z');
F                 <='Z';
Controller: Control
generic map (
             w,M,S ,Slice_id, tdelay
           )
port map    (
```

```
              t_clk,RST,Reles_Base,Required_Size,Doallocate,Dorelease,
              Allocate_Bas,Available_size,Error_signal,Space_Found,
              WR_create, RD_Memory, Create_Address, TCB_Addr,TCB_Data,
              PC_Created,L_Base, D_Base,F,producer,WR_Queue,Next_Family
            );
 Allocation: Allocate
generic map (
              w,M,S ,Slice_id, tdelay
            )
port map    (
              t_clk,RST,Reles_Base,Required_Size,Doallocate,Dorelease,
              Allocate_Bas,Available_size,Error_signal,Space_Found
            );
Continuation_Queue:  CQ
generic map (
               w,M,S, Slice_id,tdelay
            )
port map    (
               CLK,RST,PC_Created ,PC_Reschedule,L_Base,D_Base,
               F,producer,WR_Queue,Contxt_switch, PC_pipeline,
               L_Base_pip,D_Base_pip,Slot_Number_pip,
               Prefetch_PC ,Slot_Number_cache, RD_memory_prefetch,
               WR_PC, Ack
            );
end   Behaviour;
```

D.2 Microthreaded Pipeline Architecture Behaviour

The following code describes the architecture behaviour of the microthreaded pipeline. The architecture includes other components such as the multiplexer, predecode, instruction register, adder, decoding, and register file. The architecture behaviour of these components are available on the DVD attached to this thesis.

```
Architecture Processor of CPU is
constant ZeroWord       : std_logic_vector(31 downto 0);
type state is ( Reset, Local, Wait_Bus,Bus_in_Use);
type state_pip is (Rset, Schedule, Fetch, Decode);
```

```
type ram_typ is array(0 to 7) of STD_LOGIC_VECTOR(31 downto 0);
subtype word_64 is std_logic_vector(63 downto 0);
subtype word_32 is std_logic_vector(31 downto 0);
subtype word_5  is std_logic_vector(4 downto 0);
signal prstate, nxstate : state_pip :=Rset ;
signal pstate, nstate   : state :=Reset ;
signal addr             : ram_typ;
signal clk_count        : integer := 0;
signal reset_count      : bit:= '0';
signal t_clk            : std_logic :='0';
signal zero_32          : word_32 := (others=>'0');
signal zero1            : std_logic := '0';
signal zero2            : std_logic := '0';
signal four_32          : word_32 := x"00000004";
signal eight_32         : word_32 := x"00000008";
signal Rst              : std_logic := '1';
signal clk2             : std_logic := '1';
signal clk_bar          : std_logic := '0';
signal counter          : integer := 0;
signal nc11,nc12        : std_logic;
signal Create           : std_logic;
signal Context_Switch   : std_logic;
signal Kill_thread      : std_logic;
signal PC_next          : word_32;
signal PC_next_8        : word_32;
signal New_Pc           : word_32;
signal PC               : word_32;
signal instruction_code : word_64;
alias inst1             :std_logic_vector(31 downto 0)
                         is instruction_code (31 downto 0);
alias inst2             :std_logic_vector(63 downto 32)
                         is instruction_code (63 downto 32);
signal ID_IR1           : word_32;
signal ID_IR2           : word_32;
signal ID_read_data_1   : word_32;
signal ID_read_data_2   : word_32;
signal ID_sign_ext      : word_32;
signal RegDst           : std_logic := '0';
signal ID_rd            : word_5;
alias  ID_addr          : std_logic_vector(15 downto 0)
```

```
                          is ID_IR1(15 downto 0);
signal WB_IR             : word_32;
signal WB_read           : word_32;
signal WB_pass           : word_32;
signal WB_rd             : word_5;
signal MemtoReg          : std_logic := '1';
signal WB_result         : word_32;
signal WB_write_enb      : std_logic := '1';
component register_32 is
port(
     clk                 : in  std_logic;
     Rst                 : in  std_logic;
     input               : in  std_logic_vector(31 downto 0);
     output              : out std_logic_vector(31 downto 0)
    );
end component register_32;
component add32 is
port(
     a                   : in  std_logic_vector(31 downto 0);
     b                   : in  std_logic_vector(31 downto 0);
     c                   : in std_logic_vector(31 downto 0);
     cin1                : in  std_logic;
     cin2                : in  std_logic;
     sum1                : out std_logic_vector(31 downto 0);
     sum2                : out std_logic_vector(31 downto 0);
     cout1               : out std_logic;
     cout2               : out std_logic
    );
end component add32;
component instruction_memory is
port(
     addr  : in  std_logic_vector (31 downto 0);
     inst  : out std_logic_vector (31 downto 0));
end component instruction_memory;
component Instruction_register_32 is
port(
     clk                 : in  std_logic;
     Rst                 : in  std_logic;
     instruction1        : in  std_logic_vector (31 downto 0);
     instruction2        : in  std_logic_vector (31 downto 0);
```

```
    output1            : out std_logic_vector (31 downto 0);
    output2            : out std_logic_vector (31 downto 0);
    Create             : out std_logic;
    Switch             : out std_logic;
    Killed             : out std_logic;
    Base_Address       : out std_logic_vector (7 downto 0)
   );
end component Instruction_register_32;
component mux_32 is
port(
    in0                : in  std_logic_vector (31 downto 0);
    in1                : in  std_logic_vector (31 downto 0);
    swch               : in  std_logic;
    result             : out std_logic_vector (31 downto 0));
end component mux_32;
component registers is
port(
    read_reg_1         : in  std_logic_vector (4 downto 0);
    read_reg_2         : in  std_logic_vector (4 downto 0);
    write_reg          : in  std_logic_vector (4 downto 0);
    write_data         : in  std_logic_vector (31 downto 0);
    write_enable       : in  std_logic;
    write_clk          : in  std_logic;
    read_data_1        : out std_logic_vector (31 downto 0);
    read_data_2        : out std_logic_vector (31 downto 0));
end component registers;
component mux_5 is
port(
    in0                : in  std_logic_vector (4 downto 0);
    in1                : in  std_logic_vector (4 downto 0);
    ctl                : in  std_logic;
    result             : out std_logic_vector (4 downto 0));
end component mux_5;
for PC_reg: register_32
use entity work.register_32(behavior);
for PC_incr: add32
use entity work.add32(behavior);
for inst_mem: instruction_memory
use  entity work.instruction_memory(behavior);
for PC_reg: Instruction_register_32
```

```
use entity work.Instruction_register_32(behavior);
for NewPC_mux : mux_32
use entity work.mux_32(behavior);
for ID_IR_reg: register_32
use entity work.register_32(behavior);
for ID_regs:registers
use entity work.registers(behavior);
for ID_mux_rd:mux_5
use entity work.mux_5(behavior);
Begin
New_Pc  <=Program_Counter ;
PC_reg: register_32
port map(
         clk2, Rst, PC_next, PC
        );
PC_incr: add32
port map(
         PC, four_32,eight_32, zero1, zero2,
         PC_next,PC_next_8, nc11,nc12
        );
inst_mem: instruction_memory
port map(
         PC, instruction_code
        );
ID_IR_reg: Instruction_register_32
port map(
         clk, Rst, inst1,inst2, ID_IR1,ID_IR2,
         Create,Context_Switch,Kill_thread,base
        );
NewPC_mux : mux_32
port map(
         in0          => New_Pc,
         in1          => PC_next,
         swch         => Context_Switch,
         result       => WB_result
        );
ID_regs:registers
port map(
         read_reg_1      => ID_IR1(25 downto 21),
         read_reg_2      => ID_IR1(20 downto 16),
```

```
        write_reg      => WB_rd,
        write_data     => WB_result,
        write_enable   => WB_write_enb,
        write_clk      => clk_bar,
        read_data_1    => ID_read_data_1,
        read_data_2    => ID_read_data_2
       );
ID_mux_rd:mux_5
port map(
        in0    => ID_IR1(20 downto 16),
        in1    => ID_IR1(15 downto 11),
        ctl    => RegDst,
        result => ID_rd
       );
ID_sign_ext(15 downto 0) <= ID_addr;
ID_sign_ext(31 downto 16) <= (others => '0');
process(Create,Context_Switch)
Begin
if ( Create = '0') then
     Create_Address <=(others  => 'Z');
     WR_TCB <= '0';
     Con_switch <= Context_Switch;
     Kill <= Kill_thread;
else
     Create_Address <= ID_sign_ext;
     WR_TCB <= '1';
     Con_switch <= Context_Switch;
     Kill <= Kill_thread;
end if; if( Context_Switch ='1') then
    Con_switch <= Context_Switch;
    Kill <= Kill_thread;
else
    Con_switch <='0';
    Kill <= Kill_thread;
end if; end process;
end processor;
```

D.3 Local Scheduler And Microthreaded Pipeline Test *Bench*

The following code describes the test bench of the local scheduler and microthreaded pipeline.

```
architecture Dynamic_Allocation of testbench is
constant cps : time  := 30 ns;
constant N                    :integer:=7;
constant S                    :integer:=7;
constant M                    :integer:=63;
constant processor_id         :natural:=7;
constant Slice_id             :integer:=63;
constant tdelay               :time   :=4 ns;
signal   t_clk                :std_logic :='0';
signal   Rst                  :std_logic:='0';
constant w                    :integer:=31;

signal RDWR                   :std_logic;
signal WRREQ                  :std_logic:='0';
signal add_st                 :std_logic;
signal nxt                    :std_logic;
signal init                   :std_logic_vector(7 downto 0);
signal release                :std_logic;
signal nextpc                 :std_logic_vector(31 downto 0);
signal Cre_address            :std_logic_vector(31 downto 0);
signal Tag_cache              :std_logic_vector(31 downto 0);
signal Acknow                 :std_logic;
signal Newpc                  :std_logic_vector(31 downto 0);
signal PC                     :std_logic_vector(31 downto 0);
signal L_Base                 :std_logic_vector(7 downto 0);
signal D_Base                 :std_logic_vector(7 downto 0);
signal Slot_pip               :std_logic_vector(7 downto 0);
signal Slot_cache             :std_logic_vector(7 downto 0);
signal flg                    :Boolean;
signal Prefetch               :std_logic_vector(31 downto 0);
signal Family_Data            :std_logic_vector(127 downto 0);
signal Reschedule             :std_logic_vector(31 downto 0);
signal TCB_Addrs              :std_logic_vector(31 downto 0);
```

```
signal allocated_Successfuly :std_logic;
signal Conxt_switch          :std_logic;
signal Kill_thread           :std_logic;
signal Kill_slot             :std_logic_vector(7 downto 0);
signal write_pc              :std_logic;
signal Inst_data             :std_logic_vector(127 downto0);
signal Inst_Addr             :std_logic_vector(31 downto 0);
signal RD_Cache              :std_logic;
signal RDM_prefetch          :std_logic;
signal Successes            :std_logic;
signal Reschedule_PC        :std_logic_vector(31 downto 0);
signal WR_New_PC            :std_logic; Begin

RST <= '0';
Local_Scheduler: Scheduler
generic map (
             w,M,S ,Slice_id, tdelay
            )
port map    (
             t_clk,Rst,WRREQ,Newpc ,Cre_address,Reschedule_PC,
             Family_Data, Acknow ,  Conxt_switch, Kill_thread,
             kill_slot,successes,PC,L_Base,D_Base,Slot_pip,
             WR_New_PC,Prefetch,Slot_cache,nxt,RDWR,
             RDM_prefetch ,TCB_Addrs
           );

Instruction_Memory : Instruction_Cache

generic map ( w ,Slice_id, tdelay
            )
port map    (
             t_clk,Rst,  RDWR, TCB_Addrs,Family_data,
             Inst_data,Inst_Addr,RD_Cache,RDM_prefetch,
             Acknow,Prefetch
            );

Microthreaded_Pipeline:  CPU
generic map  (
              w,Processor_id  ,N =>N
             )
```

```
port map    (
             t_clk,rst,nxt,PC,L_Base,D_Base,Slot_pip,
             WR_New_PC,init,Cre_address ,WRREQ,
             Conxt_switch,Kill_thread,kill_slot,
             Inst_data,Inst_Addr,RD_Cache,Successes,
             Reschedule_PC
            );
end  Dynamic_Allocation;
```

D.4 Simulation Results

This section provides VHDL simulation results of the local scheduler and the first two stages of the microthreaded in-order pipeline. Different execution scenarios are presented (see figures D.1 to D.5).

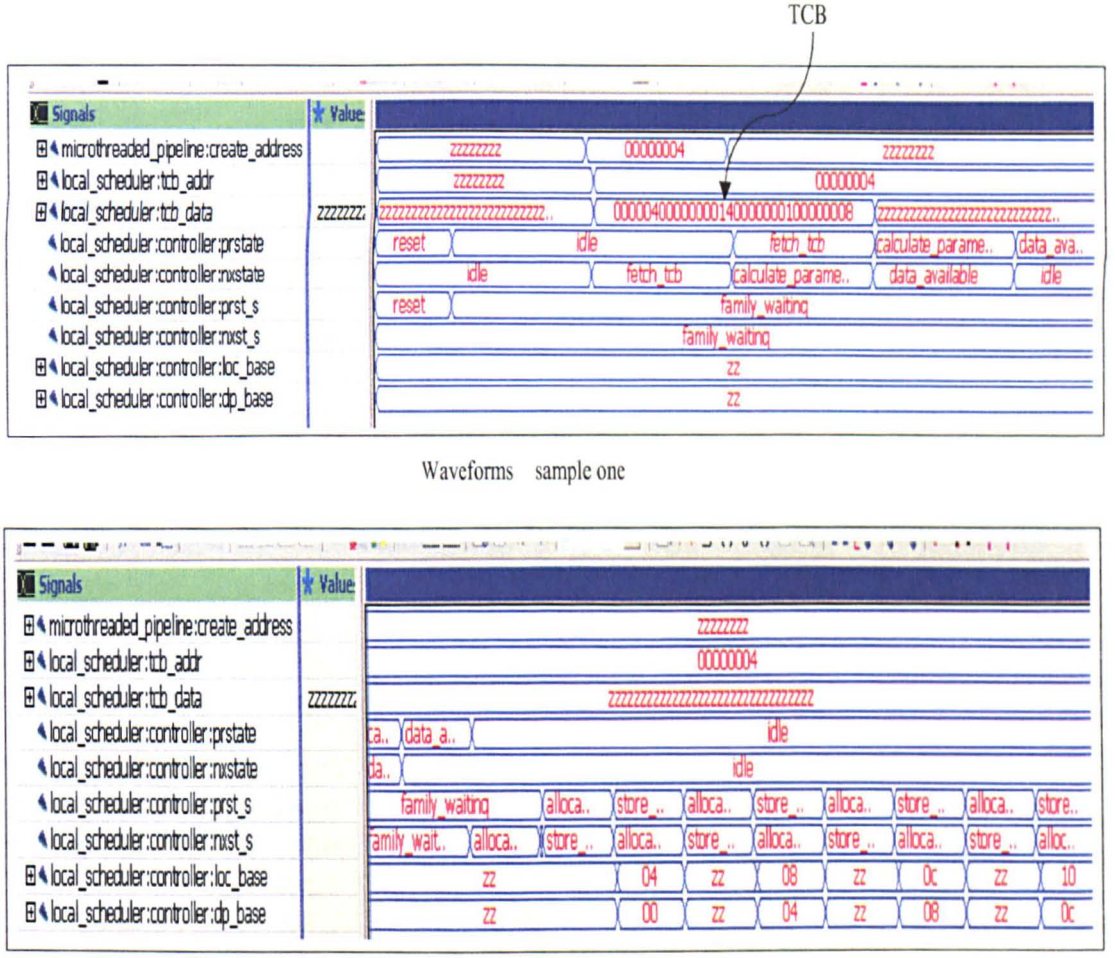
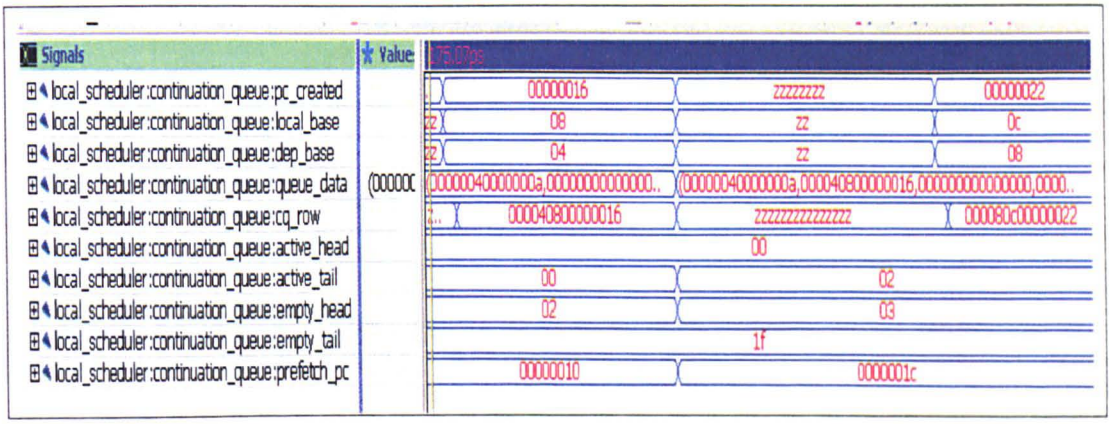
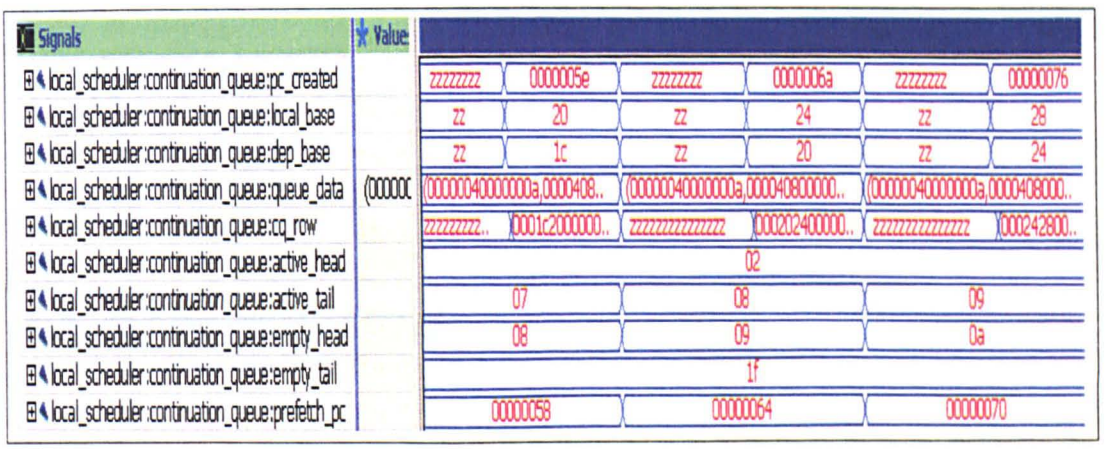


Figure D.1: Simulation waveforms showing family creation and threads allocation.

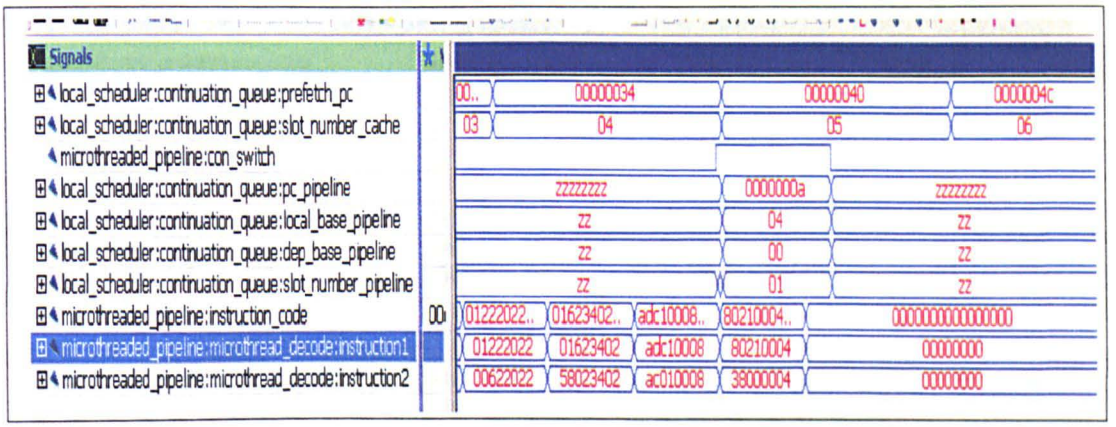


waveforms sample one

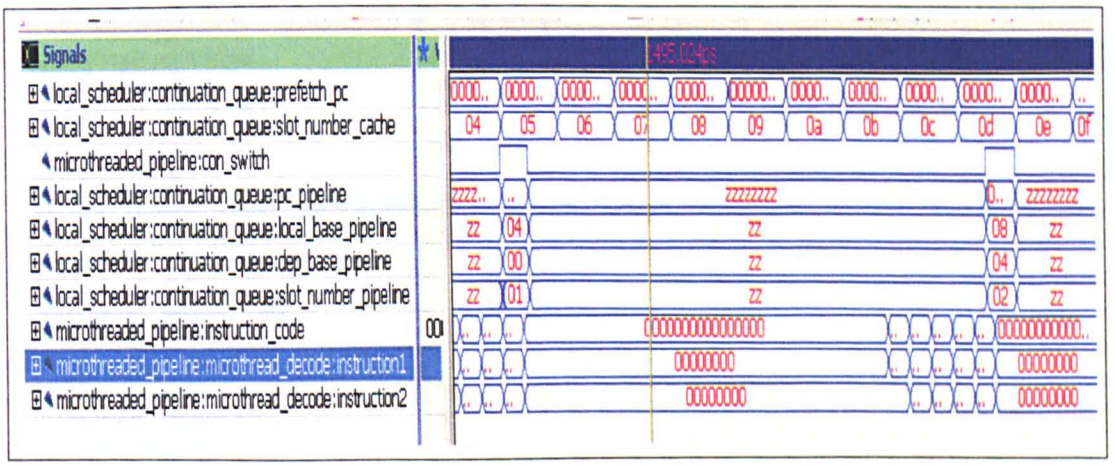


waveforms sample two

Figure D.2: Simulation waveforms showing thread state in the continuation queue.



Waveforms sample one



Waveforms sample two

Figure D.3: Simulation waveforms showing instruction fetch state and microthreaded pipeline with a context switching.

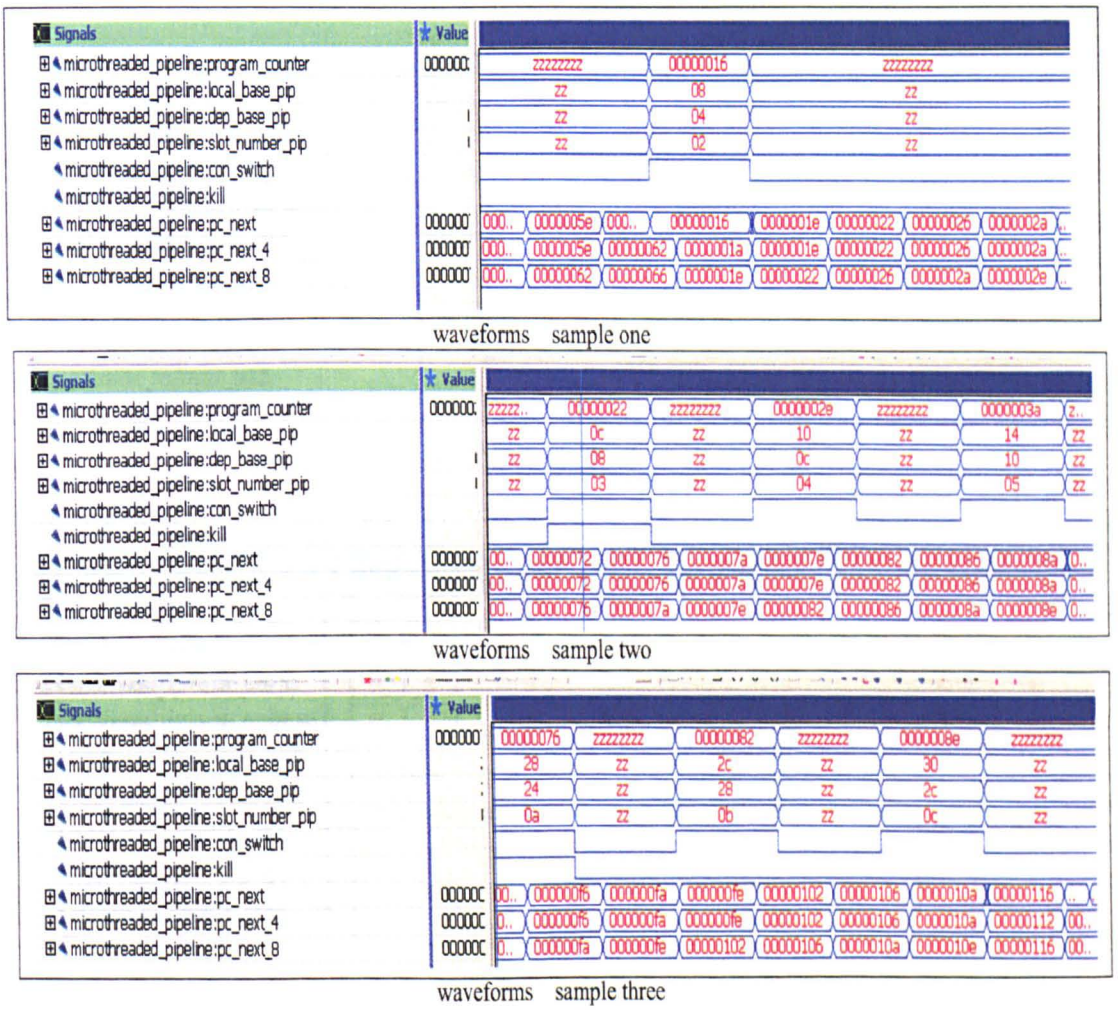
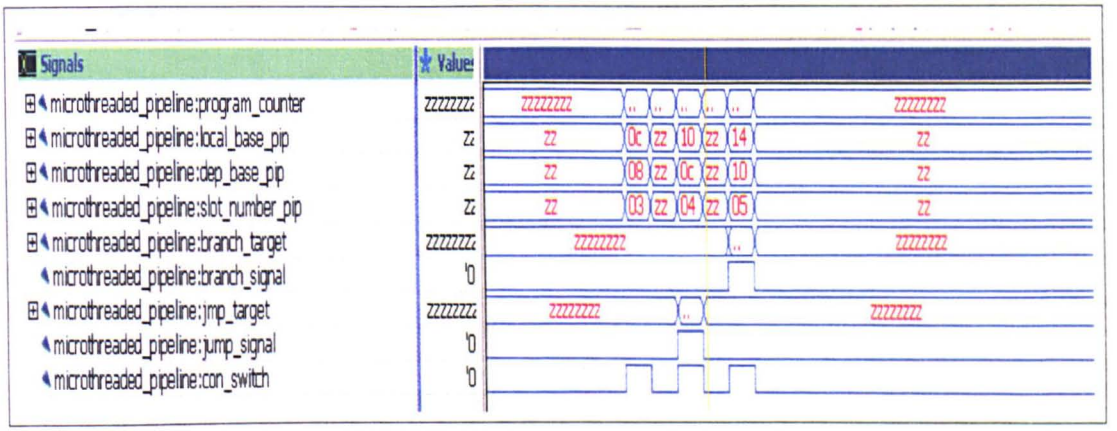
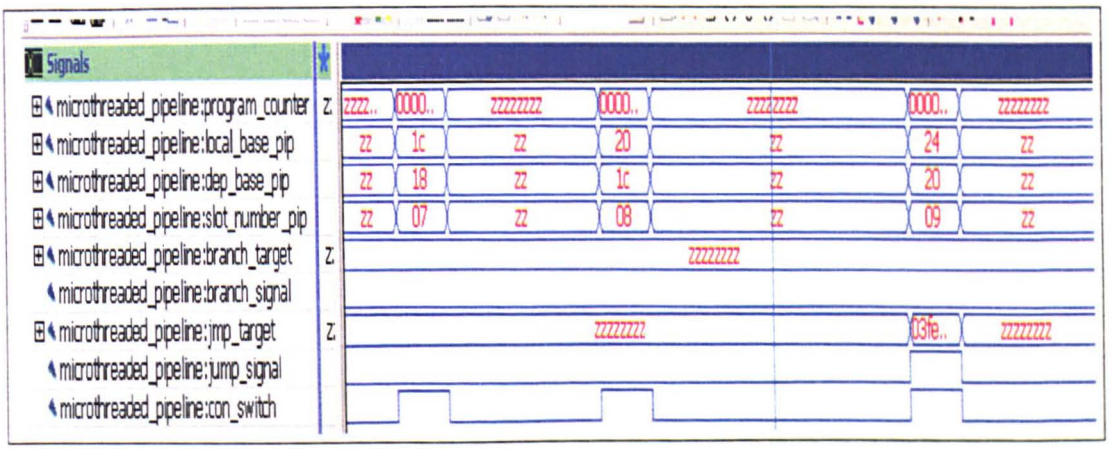


Figure D.4: Simulation waveforms showing microthreaded pipeline with context switch and kill instructions.



waveforms sample one



waveforms sampletwo

Figure D.5: Simulation waveforms showing microthreaded pipeline with branch and jump instructions.