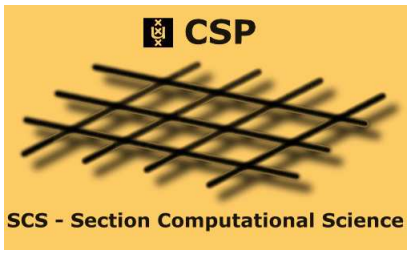

Implementing microthreaded microprocessors in VHDL

Jun Wu

Master's Thesis in Computer Science

Prof. dr. Chris Jesshope, Supervisor

February 27, 2006

Universiteit van Amsterdam
Faculty of Science
Informatics Institute

This thesis, ”Implementing microthreaded microprocessors in VHDL”, is submitted in partial fulfilment to the requirements for the degree Master of Science in Computer Science at the Universiteit van Amsterdam.

Date:

Author’s signature:

Supervisior’s signature:

Abstract

Micro-threads are small fragments of code based on loops, can be executed concurrently and the concurrency is described parametrically in the binary code. This thesis describes a microthreaded model of concurrency and illustrates the complete bottom-level design of its pipeline compared with the conventional MIPS pipeline. The simulations from a single microthreaded processor implemented in VHDL are also presented in this thesis. This simulation concentrates on the first stage of this pipeline which includes micro-threads create and dynamically schedule or reschedule. This stage is also the most important stage of the microthreaded pipeline.

Acknowledgments

I have been work five years as a space computer engineer in China Academy of Space Technology. I am really honored I can be a master student of computer science in University Van Amsterdam.

I would like to acknowledge my daily supervisor, Prof. dr. Chris Jesshope, who has provided me with the chance to the subject and guiding me throughout the whole project. I have for a long time been interested in low-level programming of microprocessors. During the design of the microthreaded processor. Professor Chris Jesshope teaches me from the cpu's basic theory to core design and simulation. From him, I learned a lot on microprocessor design that will be useful in my future job. I have also learned a lot on digital design in general. He gave me a lot of help for not only knowledge, but also daily life. If no his kindly help, to complete my master program is definitely unbelievable.

I would like to take this chance to thank to the University Van Amsterdam and the Faculty of Computer Science for offering so nice master program in computer science to international students. Also I would like to express my gratitude to Niels Molenaar in international office. He supported me in all possible ways during these years.

My special appreciation is to Prof. Peter Sloot for organizing the Master of Computational Science program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Micro-threads are lightweight threads drawn from a single context and were first introduced in 1996. They are small fragments of code based on loops, can be executed concurrently and the concurrency is described parametrically in the binary code (2; 3; 4; 5; 6; 7; 8; 9; 10). In this thesis, we describe the implementation of the micro-thread pipeline, which is derived from a conventional MIPS pipeline and supports addition to its ISA-μt[1] (see Chapter 2) and makes use of VHDL [2] to realize this pipeline model. The implementation supports dynamically scheduling of micro-threads and enables scalable implementations of various types of wide-issue multiprocessors. This model is tested on different assembly codes, providing a verification of the applicability of this model based on those simulation results.

1.1 Moore's law

Moore, one of the founders of Intel®, made a powerful prediction in an article in the April 19, 1965 issue of Electronics magazine that the total number of transistors on the cheapest CPU will grow exponentially at a constant rate and that this constant rate produces a doubling every 12 (or 18, or 24) months.

> "The first microprocessor only had 22 hundred transistors. We are looking at something a million times that complex in the

[1] ISA-μt:Instruction set architecture of micro-threads. It consists 5 more instructions for explicit concurrency control — Create, Swch, Kill, Break and Bsync.

[2] VHDL:Very High Speed Integrated Circuit Hardware Description Language.

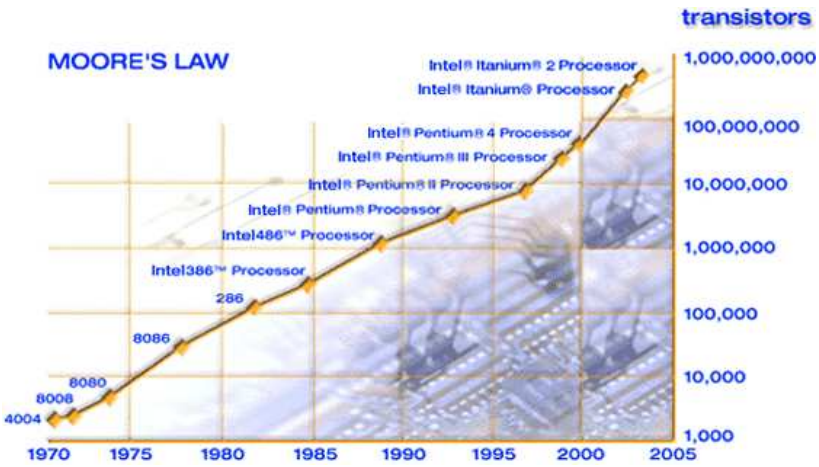

Figure 1.1: Moores Law Means More Performance. Processing power, measured in millions of instructions per second (MIPS), has risen because of increased transistor counts. Courtesy of Intel®.

> next generations-a billion transistors. What that gives us in the way of flexibility to design products is phenomenal."

What is less well-known is that Moore also stated that manufacturing costs would dramatically drop as the technology advanced. If we look back in time, Gordon Moore's prediction of exponential growth of the number of transistors on a chip has not only been achieved but in some cases exceeded. Conventional microarchitectures have been improving in performance by approximately 50-60% per year, improving the instructions per cycle (IPC) using more transistors on a chip and increasing the clock speed. In 1965, a single transistor cost more than a dollar. By 1975, the cost of a transistor had dropped to less than a penny, while transistor size allowed for almost 100,000 transistors on a single die. From the roadmap of the Intel® microprocessor, we can see that designs are now topping out at well above 1 billion transistors, running at 3.2 GHz and higher, delivering over 10,000 MIPS, and can be manufactured in high volumes with transistors that cost less than 1/10,000th of a cent (11).

1.2 Problems facing current architectures

The key metrics for characterizing a microprocessor include: performance, power, cost (die area), and complexity(12). One of the biggest challenges,

which the current designer is facing is the design of a billion-transistor architecture, which achieve high performance, low power, low cost and low design complexity. During the last three decades, even the microprocessors have gone through significant changes, however, the basic computational model has not changed much. The instructions and data are the only components of a program. The instructions are encoded in a specific instruction set architecture (ISA). The computational model is still a single instruction stream, sequential execution model, operating on the architecture states (memory and registers) (12).

At the end of 2004, it was well know that there would not be a version of the Pentium 4 running at 4.0GHz. Intel has dropped the plans for this processor which should be released before Oct, 2004, according the previous microprocessor roadmap. The explanation from Intel was that this decision to drop plans for a 4.0GHz CPU is a result of Intel's change in focus with regards to both processor strategy and marketing. However, in fact, the real reason why Intel cancelled the plan is merely a compromise with the challenges of the conventional microprocessor architecture we have been facing from the past several decades. Like Intel, other semiconductor manufacturers are also having similar issues in producing faster single-core processors because of the following challenges, which all manufacturers have been facing in the past or will face in the future:

- Memory wall

  All modern microprocessors employ memory hierarchy. The growing gap between the frequency of the microprocessor that doubles every two to three years and the main memory access time that only increases 7% per year imposes a major challenge(12). The latency of today's main memory is approximate 30 ns, which approximately equals 100 microprocessor cycles. The two most obvious reasons for this limitation are

  1. Memory has scaled in size but not in access time
  2. Speed of physical interconnects remains bounded by the laws of physics.

- Power challenge

  Dynamic power and static (leakage) power are both key issues. Static power will surpass dynamic power at 65nm. It is implicated that in the future chips will have billions of transistors, but can only power a fraction of them at once(13).

Power density has also reached a limit, where conventional techniques used to package and to dissipate heat are no longer able to cope. Physical demands on the silicon will cause excessive heat and eat up more power than any semiconductor manufacturers can accept. If the trends continue, power dissipation will increase from 100W in 1999 to about 2,000W in 2010(14). Chips in 0.6-μm technology used to have power density similar to a (cooking) hot plate (10 W/cm^2), we may experience a chip with the power density of a nuclear power plant or even a rocket nozzle soon if the trend continues(12). In(12), the author also mentions three general approaches to power reduction:

1. increase energy efficiency;
2. conserve energy when a module is not in use;
3. recycle and reuse;

These approaches can all be exploited by the microthreaded model in reducing the power dissipation and these are addressed in chapter 2

- Speedup

  Performance depends only on IPC and Frequency for a given executable program. In order to keep the pace of performance growth, one of the challenges has been to increase the frequency without negatively impacting the IPC. From the year 1998 to 2005, for instance, microprocessor manufacturers obtained got enormous gains in commercial Superscalar Processor's frequency from 500MHz to 3800MHz. However, the IPC has remained unchanged at around1.0-2.0 for typical applications. Furthermore, both conventional strategies will fail for future technologies (50nm and below), with clock speed growth slowing down because of fundamental pipelining limits and wire delays making architectures communication bound(15).

- Use of frequency versus Concurrency

  Improvement in clock speed to obtain better performance can not be continued infinitely, as power density is a function of frequency and is becoming a critical design constraint, which is already mentioned above. Using concurrency as a means of increasing performance without increasing power density is a much better strategy based on the assumption that all models and implementations are completely scalable. However, concurrency management is inherently arduous and will increase the design complexity. It requires

additional complexity that involves additional work (e.g,dependency tracking, speculation, scheduling) to be done on each instruction.

- Scalability

  As mentioned above, using concurrency will be the better choice than just increasing aggressive clock frequency. However, speedup can not to be achieved from concurrency if the performance, area and power dissipated are not all scale linearly with issue width. Today's architectures will not scale, showing diminishing returns in IPC even with increasing chip transistor budgets. The ILP of conventional architectures such as superscalar core is limited by the issue window, whose logic complexity grows as the square of the number of entries(16). The register files, re-order buffers and issue windows, which are frequently accessed global structures and relied by the conventional architectures, become bottlenecks limiting clock speed and pipeline depths. Thus the performance is not scalable with the current technology. The power and area are not scalable with the conventional strategy as well. Maximum power consumption is proportional to voltage^2 and frequency as follows: Power = C $\times$ V^2 $\times$ Frequency, C is the effective load capacitance. The area can not scale down with transistor widths as the wire delays and the on-chip power dissipation for the long wires communications(communication-bound) do no scale at the same rate. Recent research demonstrates that the inter communication network accounts for 36% in Raw processor(17) or 50% in Intel processor(18) of the total chip power. One of the key constraints faced by designers is how to alleviate the high performance penalty of long wire delays compare with the high clock cycle at future technologies.

1.3 Alternate approaches

- Out-of-order

  The current approach in attempting to boost IPC is *out-of-order execution*. Out-of-order execution is a restricted form of data flow computation. The first machine to use out-of-order execution was probably the CDC 6600 (1964) which used a scoreboard to resolve conflicts. The key concept of out-of-order processing is to allow the processor to hides some of the stalls that occur when the data needed to perform an operation is not available(e.g. cache miss). The microproces-

sor can schedule new instructions as long as they are independent. A superscalar out-of-order microprocessor can achieve higher IPC than a superscalar in-order microprocessor. Out-of-order execution involves dependency analysis and instruction scheduling. Therefore, it takes a longer time (more pipe stages) to process an instruction in an out-of-order microprocessor(12). The size of the instruction window and logic complexity increase quadratically with the issue width, which results in an out-of-order microprocessor, especially a wide-issue one, that is much more complex and power hungry than an in-order microprocessor(16).

- VLIW

  A Very Long Instruction Word or VLIW CPU architecture implements a form of explicit instruction level parallelism, where multiple functional units are used concurrently as specified by a single instruction word. The original concept of VLIW is basically developed from the superscalar processor, which tries to achieve improvements in the quality of the control unit. One potential solution to this problem is to move the dispatcher logic out of the chip and into the compiler, which can spend considerably more time and effort on making the best decisions possible. This is the basic premise of very long instruction word (VLIW) CPU designs, which is also known as *static superscalar* or *compile time scheduling*(19).

  Early VLIWs operated in lockstep; There was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus, different numbers of functional units and unit latencies require different versions of the code(1).

- EPIC

  HP introduced one instruction set architecture called EPIC (Explicitly Parallel Instruction Computing) which builds upon VLIW and pre-VLIW work performed over the past two decades. EPIC is a 64-bit microprocessor instruction set, that provides up to 128 general and floating point unit registers. The specific jointly designed instruction set architecture was named IA-64 which was introduced by Intel.

More recently, Intel has preferred to use IPF (Itanium processor Family) as the name of the first implementation.

In particular, EPIC builds upon the architectural ideas pioneered at Cydrome and Multiflow. Explicit information on independent instructions in the program is a major distinguishing feature of EPIC architectures. It uses *Unbundled branches* and *Predicted execution* to deal with eliminating and/or speeding up branching, *Cache spedifiers* and *Data speculation* to deal with cache locality management, *Control speculation* and *Predicated code motion* to deal with starting load instructions as early as possible(20). EPIC needs very large register file, as a smaller register set limits performance. Current efficiency of utilizing the register files is low, the processor has to shuffle data in and out of registers instead of doing the work required for the program(21). And EPIC still can not avoid vulnerable to speculation hazards, which is inhered with predicated execution.

- TRIPS

  The TRIPS architecture is the first instantiation of an EDGE (Explicit Data Graph Execution) instruction set. Direct istruction communication is one main characteristic of EDGE architecture. irect instruction communication means that the hardware delivers a producer instruction's output directly as an input to a consumer instruction, rather than writing it back to a shared namespace, such as a register file. Only block outputs written back to register file. Compiler structures program into sequence of hyperblocks which specify explicit instruction placement in the ALU array. A TRIPS block resembles a 3D VLIW instruction, with instruction filling fixed slots in a rigid structure. The TRIPS processor is a static placement, dynamic issue (SPDI) architecture, whereas a VLIW machine is a static placement, static issue (SPSI) architecture(22). TRIPS has a totally different binary interface, so the backward compatibility can not be achieved now. The I-cache capacity and bandwidth can not be sufficiently utilized if the block size are less than normal or sufficient size(23).

- Multi-threading

  For the current generation of microprocessors, hardware multithreading is becoming a generally applied technique, which exploit *thread-level parallelism* (TLP). *thread-level parallelism* (TLP) is a coarser-grained parallelism when compared with the *instruction level parallelism* ILP. The notion of a thread which we discuss in this thesis

differs from the notion of software threads in multithreaded operating systems. The thread in Multi-threading processors can be operating system thread, a compiler-generated thread or even a hardware-generated thread(24).

From the first multithreading processor DYSEAC (1954) which use the technique *multiple sequence*, there were a number of other types of multithreading, such as *fine-grain MT* (FGMT), *coarse-grain MT* (CGMT), *simultaneous MT* (SMT), *implicit MT* (IMT), and *dynamic MT* (DMT).

Within the processor, a multithreaded processor can control at least two threads in parallel. *Explicit multithreaded processors* interleave the execution of instructions of different user-defined threads (operating system threads or processes) in the same pipeline and differ from the *implicit multithreaded processors* and *dynamic multithreaded processors*, which increase the performance of sequential programs by applying compiler-based or hardware-based thread-level speculation. The implicit and dynamic multithreaded processors can not be scalable since they suffer from the misprediction. In case of misspeculation, all speculatively generated results must be discarded. More details are given in(24).

Many forms of *explicit multithreaded processors* have been introduced from 1960's, such as the *Interleaved multithreading* (IMT), *Blocked multithreading* (BMT), instructions can be issued only from a single thread in a given cycle, and *Simultaneous multithreading* (SMT), instructions can be issued from multiple threads in a given cycle. SMT is widely implemented by some modern commercial processors, such as Intel Pentium 4 *Hyper-Threading* (HTT), DEC Alpha EV8, and MIPS MT. RMI.

Interleaved multithreading (IMT) is one type of *fine-grain MT* (FGMT), it switch threads on each cycle. Some processors such as Heterogeneous Element Processor (HEP), the Horizon, and the Cray Multi-Threaded Architecture (MTA) multiprocessors are the most well-known examples of IMT. It apply the techniques such as *dependence look-ahead technique* and *interleaving technique* try to overcome the processing power accessibility. The microthreaded processors described in this thesis is closest to the IMT. Microthreaded model and IMT, both processors are in-order core, can eliminate control and data dependences between instructions in the pipeline, and pipeline hazards

cannot arise and the processor pipeline can be easily built without the necessity of complex forwarding paths. This leads to a very simple and therefore potentially very fast no hardware interlocking or data forwarding pipeline. Moreover, the context-switching overhead is zero cycles. Long memory latency can be tolerated by both schemes which can cover memory latency and improve overall throughput (24). The difference between microthreaded model and IMT will be discussed in chapter 2.

- Chip multiprocessors

  In order to achieve better utilization of *thread-level parallelism* (TLP), especially for some application that lack sufficient *instruction-level parallelism* (ILP), some current commercial processors including Intel Itanium 2, PA-RISC (PA-8800), IBM POWER (POWER4 and POWER5), and SPARC (UltraSPARC IV) adopt multi-cores on a single chip. Current CMP integrate superscalar cores which share a common second- or third-level cache and interconnetc. The CMP will be widely used to achieve better scalbility and advancer power and thermal management, at last get obvious performance improvement in the next generation microprocessors(25).

  There are several challages for CMP, because memory access time is much slower than the processor speed, and even a single-core processor still faces the increasing gap between memory and processor speed. CMP can make this case worse, if, as normally, just one processor can access the shared memory once. The other big problem is how to get the performance improvement for the legacy or general programs which are usually not programmed for multithreaded processing. In fact, most consumer softwares are not written in such a manner that they can gain large benefits from CMP systems. Writing correct and efficient multithreaded programs with conventional programming models is still an incredibly complex task limited to a few expert programmers. The performance potential of CMP is just limited to multiprogramming workloads and a few server applications(26).

1.4 Thesis outline

This thesis describes the research into microthread processor operation and the design of a microthread pipeline in VHDL. Chapter 2 shows the whole bottom-level design of the microthread pipeline in detail. It introduces

the conventional MIPS pipeline briefly, and then compares microthread pipeline with traditional one to show how the microthreaded processor works and how it can get improvement in TLP and ILP. In chapter 3 the design of the first stage of the pipeline is presented, which includes instruction fetch, local scheduling logic and Resigter Allocation Unit (RAU). Chapter 4 describe the simulation results using VHDL. Finally, chapter 5 gives the conclusion of this thesis and discussion about future work.

Chapter 2

Microthreaded Processor

In this chapter, we deliberate the microthreaded concurrency model in more detail and explore the whole up-bottom design of the Microthreaded processor which comprises a pipeline, a local scheduler, a asynchronous communicator, a large register file and a local I-cache. The microthreaded pipeline is based on the conventional in-order execution pipeline microprocessor — MIPS R2000. The MIPS R2000 was announced in 1985, and was the first commercial MIPS CPU model. It offers a very clean instruction set and pipeline. We choose this simple prototype since this model can be easily implemented as the core of the microthreaded processor and extended to build more complex system. The objective of the design is to be able to run the microthreaded instructions, and as the same as the MIPS R2000, all instructions can be run in one cycle. All the microthread instrcutions and most of the MIPS R2000 instructions are supported, except multiply, division and floating point instructions. Besides the support of the single-core architecture, an interface for the CMP (Chip Multiprocessors) architecture will also be included in this design.

2.1 Conventional MIPS Pipeline

2.1.1 An Overview of Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. It improves instruction through-put rather than individual instruction execution time. Today, pipelines are key to making processors

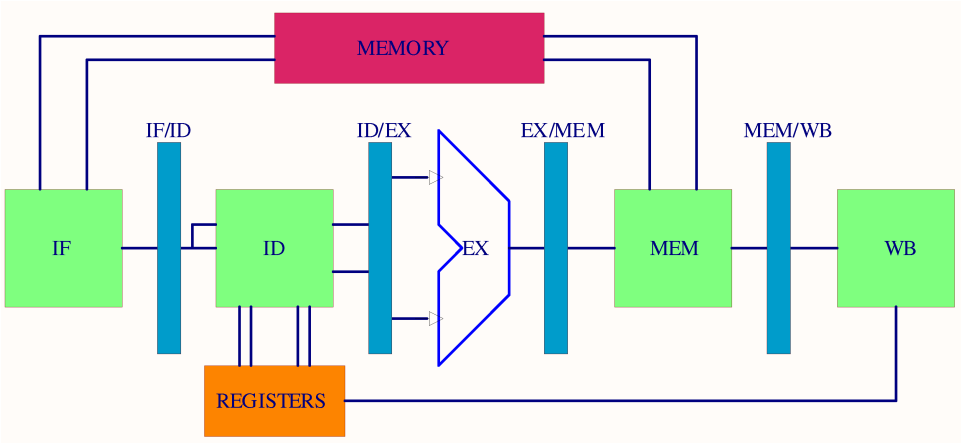

Figure 2.1: Typical simple five stages MIPS Pipeline.

fast. More details of pipelining are given in (1).

The classical MIPS pipeline has five stages. Figure 2.1 shows the typical simple MIPS Pipeline.

- IF(Instruction Fetch): Instructions are fetched from the instruction memory (cache).
- ID(Instruction Decode): Read registers, decode the instruction, generate the control signals and calculate branch address. Reading and decoding are allowed to occur simultaneously.
- EX(Execution): Arithmetic and logic operation execution or address calculation.
- MEM(Memory access): Access an operand in data memory (cache) on load and store instructions.
- WB(Write back): The result is written into a register.

2.1.2 Datapath of Pipeline

- Instruction fetch:

  The value of register PC is the address of the instruction which is being read from the memory. After this instruction is fetched from memory, the PC address is incremented by 4 and then written back

into the PC to be ready for the next clock cycle. The instruction is also placed in the IF/ID pipeline register which also saves the incremented PC in case it is needed later for an instruction, such as branch instruction. When it has an exception, the instruction will be fetched from an identical location. In this case, the PC multiplexor sends 0x4000 0040 to the PC.

- Instruction decode and register file read:

  The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate to 32-bit. These three 32-bit values are all stored in the ID/EX pipeline register along with the incremented PC address, the source and destination register number. The sign-extended immediate is shifted left two and added with the incremented PC. That sum is the effective branch address which is sent back to the PC multiplexor.

- Execute or address calculation:

  The instruction reads the contents of two registers and does the arithmetic/logic operation, or it gets an address by adding the register 1 with a sign-extended immediate. This address is used in MEM stage. Those results are placed in the EX/MEM pipeline register. The bypass network is described in section 2.1.4.

- Memory access:

  The instruction reads the data memory using the address from the EX/MEM pipeline register and loads the data into the MEM/WB pipeline register.

- Write back:

  The data is read from the MEM/WB pipeline register and written into the register file.

2.1.3 Pipelined control

The control information is created during instruction decode stage and propagated through the pipeline. The control signals are used to select the Result register, the ALU operation, either Read data 2 or a sign-extended immediate for the ALU, branch assertion, pipeline flush, memory read or write, and register file manipulation.

2.1.4 Pipeline hazard

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards. There are three types of hazards in a conventional MIPS pipeline(1).

- Structural hazards

  Structural Hazards means that the hardware does not have enough resource to support the combination of instructions which are executed in a clock cycle. For instance, the MIPS pipeline structure does not support accessing the same memory twice during one clock cycle, thus if only one memory is used it will be impossible to solve a store or load instruction without stalling the pipeline. Structural hazards are easy to eliminate. The simplest method is increasing the number of resources(for example, using two memories, one for instructions and one for data).

- Control hazards

  A *control hazards* is when instruction fetch cannot continue because the execution path is dependent on a decision based on the results of an earlier instruction(branch). It can be considered as special case of a data hazard. But they are separated category because they are treated in different ways. This can be applied to the branch instruction. Stall and prediction are the two simple techniques to resolve the control hazards. In 2.2.3, it is indicated how the microthreaded pipeline resolves control hazards.

- Data hazards

  If an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction, it has a *data hazard*. A solution is to get the result from the pipeline before it reaches the write back stage. This solution is called forwarding or bypassing. Reference (1) introduces the forwarding logic for two cases:

  – EX hazard:

if(EX/MEM.RegWrite and (EX/MEM.RegisterRd≠0) and (EX/MEM.RegisterRD=ID/EX.RegisterRs)) Bypass data from register EX/MEM.

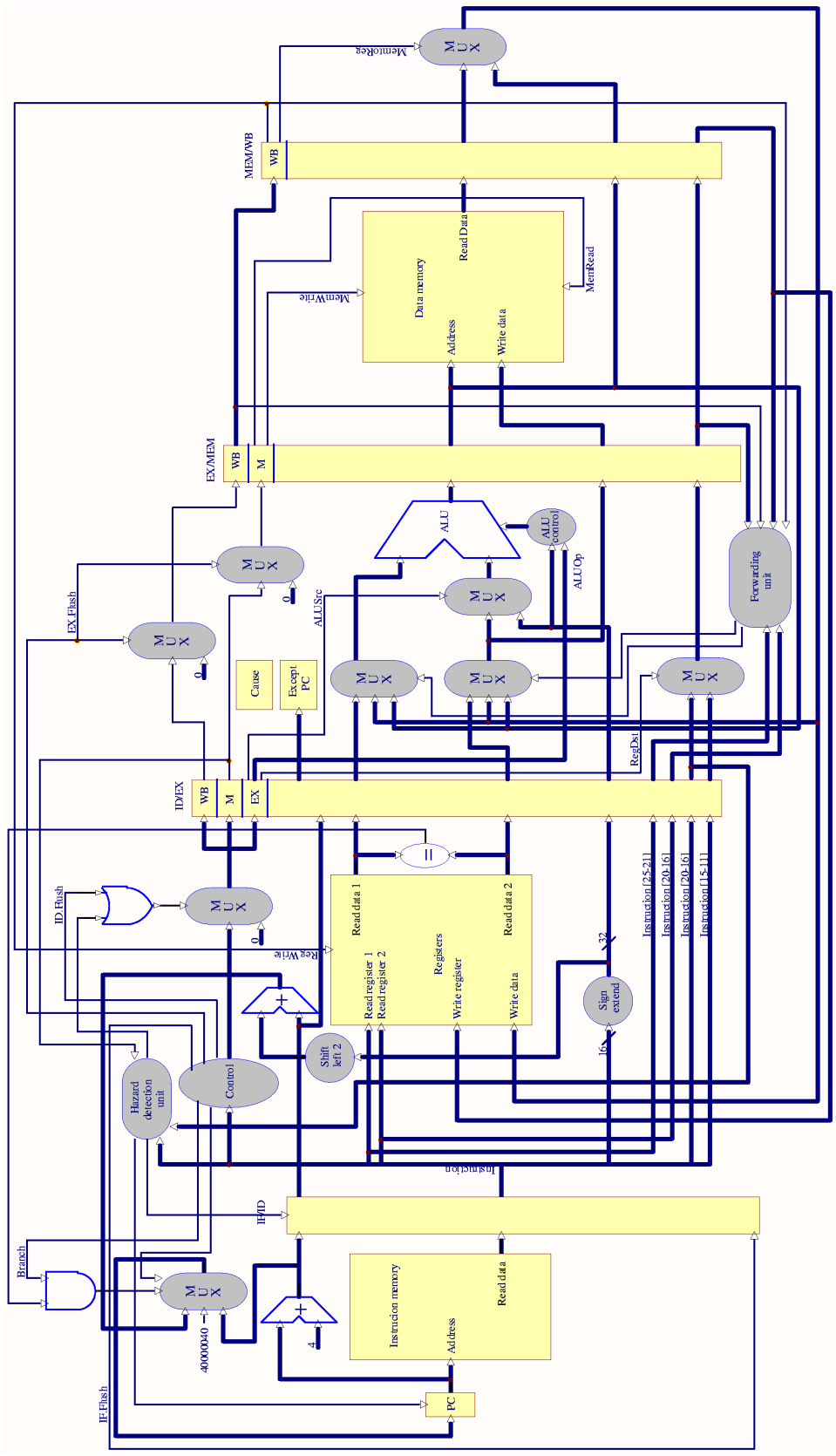

Figure 2.2: The datapath and control for MIPS pipeline. Courtesy of Hennessy and Patterson (1)

– MEM hazard:

if(MEM/WB.RegWrite and (MEM/WB.RegisterRd≠0) and (EX/MEM.RegisterRd≠ID/EX.RegisterRs) and (MEM/WB.RegisterRd=ID/EX.RegisterRs)) Bypass data from register MEM/WB.

Section 2.2.3 indicates the difference between microthreaded pipeline and MIPS pipeline forwarding logic.

Figure 2.2 shows the R2000 MIPS pipeline datapath and control. A description of this figure can be found in (1).

2.2 Microthreaded processor architecture

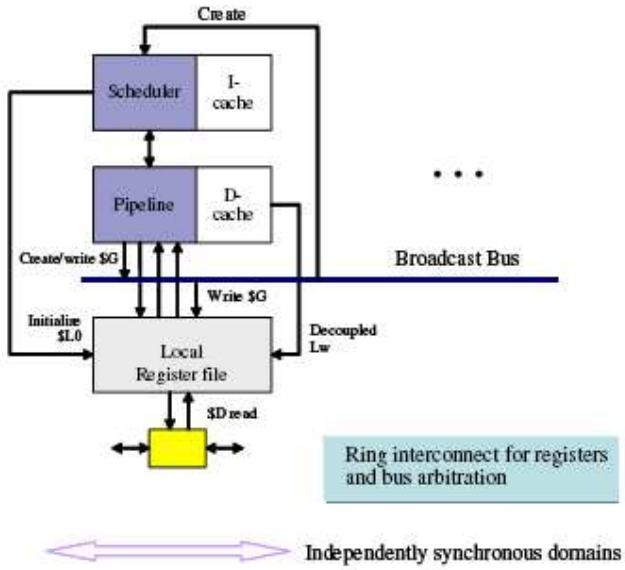

Figure 2.3: Microthreaded processor includes a broadcast bus and ring network. Courtesy of Chris Jesshope

Figure 2.3 gives an overview of the Microthreaded processor, its internal structure and interconnections. This processor comprises a pipeline, a local scheduler, an asynchronous interface, a large register file and a local I-cache (27). In this project, this memory architecture eliminates the L1

D-cache completely. In paper (27; 28), it mentions that the Microthread processor may or may not have a local D-cache because of the constraint of area and power, and the inherent advantage of Microthreaded processor, which can tolerant and hide high memory latency. The authors compare two cases, the first set of results are for a relatively complex level-1 D-cache of 64Kbyte, with 8-way associativity and 64 byte line size, the second set of results are for a 1Kbytes with direct mapped cache lines. It can be seen that number of instructions executed, time to solution and IPC are all virtually unchanged (28). This scheme reduces the complexity of the processor by omitting the L1 D-cache.

The microthread model can be applied to an arbitrary ISA by the implementation of just 5 instructions that provide concurrency controls. These instructions are shown in table 2.1. The Microthread instructions are only executed in the first stage of the pipeline and only control the behavior of the *Local scheduler*.

The *Local scheduler* is therefore the most important component of a microthreaded processor. The understanding, design and implementation in VHDL of this components is the major contribution of this thesis.

Table 2.1: Concurrency-control instructions

Instruction	Instruction Behavior
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

Table 2.2: Microthreaded instructions' opcode map

	op(31:26)	
18	Swch	011000
19	Kill	011001
1a	Bysnc	011010
1b		011011
1c	Cre	011100
1d	Brk	011101

2.2.1 Local scheduler

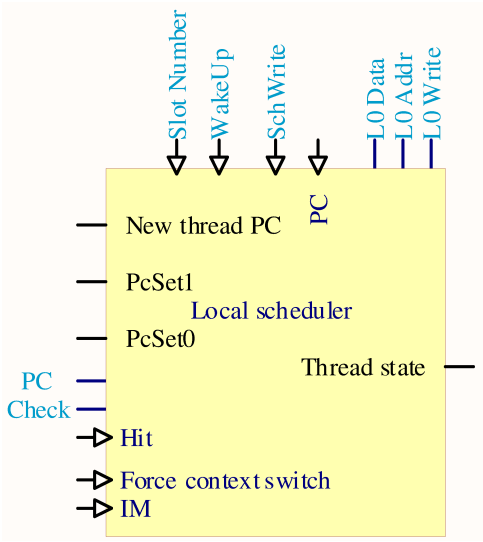

Figure 2.4: The interface to the Local Scheduler

The local scheduler determines which iterations from a family of microthreads is to be executed and manages the state of them. In chapter 3, we will illustrate the design details of the Local Scheduler. Below we just illustrate the interface and control logic of this Local Scheduler.

The *Register File* is responsible for synchronizing an instruction's execution with the production of its data. To do this it suspends instructions until their data is available. To reschedule the suspended thread it will send the signals *Slot Number* and *WakeUP* when data is written to a register in which a thread reference (*Slot number*) is stored in. A reference of the thread is written in a register on a read failure and that thread is rescheduled only when data is satisfied. Once the signal *WakeUP* is asserted, the scheduler reads the *Slot Number* to find the location in the data structures and activate this waiting thread immediately.

The register read stage also determines the value of the thread's *PC* following a context switch and then assert the signal *SchWrite* to let local scheduler change the corresponding thread's PC. This *PC* written to the scheduler could possibly be current instruction PC, when a read register fails, branch target instruction PC, when the branch is taken, or instruction PC+8 when the branch is not taken.

IM is the pre-fetching port which input the instruction that has the ad-

Table 2.3: Interface to the Local Scheduler

Slot Number	9 bits	input
WakeUp	1 bit	input
SchWrite	1 bits	input
PC	32 bits	input
IM	32 bits	input
PCSet0	1 bit	output
PCSet1	1 bit	output
New thread PC	32 bits	output
L0 Data	32 bits	output
L0 Addr	10 bits	output
L0 Write	1 bit	output
Thread state	35 bits	output
PC	32 bits	output
Check	1 bit	output1
Hit	1 bit	input
Force context switch	1 bit	input

dress PC+4. The pre-fetching mechanism is the key why the *Michthreaded* pipeline does not need extra pipeline cycles to execute the context switch instructions. A concurrency control instruction always follows an executable one and is prefetched and executed concurrently with it as there are no structural hazards in doing so.

PcSet0 is used for decoding whether the opcode of the prefetched instruction, *IM[PC+4]* is a ISAμt or not.

$$PcSet0 = MF_{29} * MF_{30} * \overline{MF_{31}}; \tag{2.1}$$

PcSet1 is the signal for decoding the opcode of *IM[PC+4]* that determines if it is a context switch instruction or not.

The signals *L0 Data*, *L0 Addr* and *L0 Write* are used for loop index initialization.

$$PcSet1 = \overline{MF_{28}} * MF_{29} * MF_{30} * \overline{MF_{31}}; \tag{2.2}$$

The local scheduler latches the new thread PC when a context switch occurs. It needs these three signals: *PC*, *Check* and *Hit* to test if the thread instructions PC are not beyond the boundary of instruction cache line. Forced context switch is a special case when the PC increments over a cache-line

boundary. All cache misses will cause a context switch if processor has more active threads or stall on waiting the cache line re-flush otherwise.

Table 2.4 is the thread state. It comprises five components and is utilized by the next stage of the pipeline. The Local/Remote bit can catenate with D-base as one component.

Table 2.4: Thread state

Field Name	Size	Description
Local/Remote	1 bit	Location of the microcontext
D-base	10 bits	Base address of the dependent register window
L-base	10 bits	Bsse address of the local register window
S+L	5 bits	Size of the dynamic windows
Slot number	9 bits	Location of the thread inside the scheduler
Producer slot no.	9 bits	Location of thread required for dependence.

2.2.2 Register File Partitioning and Addressing

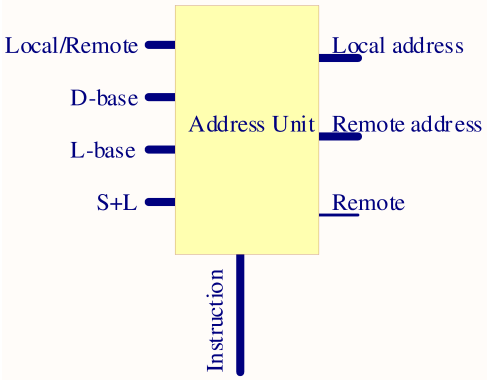

Figure 2.5: The interface to the Address Unit.

In the register read stage, the conventional MIPS ISA supplies the address of an instruction's operands directly for reading or writing to the register file. A microthreaded processor can not satisfy this need of register address from the instructions. It has a large register file which contains several different register windows. The first register window is the *Global register* ($Gi) and comprises the lower 16 registers. These registers are used to store loop invariants or any other data that is shared by all threads (28). There are three other register windows which are mapped to the upper 16

registers for addressing the microcontext of each iteration. These are the local window ($Li), the shared window ($Si) and the dependent window ($Di). The sum of the size of these three windows must be less than or equal to 16. More details about register file partition and distribution can see (28)

Table 2.5: Register file partition

Share($S3)/Dependent($D4)
Local($L3)
Share($S2)/Dependent($D3)
Local($L2)
Share($S1)/Dependent($D2)
Local($L1)
Global($G)

The thread's state (S+L, Remote/Local, D-base and L-base) and the register specifier in the instruction are required to generate a register address into the appropriate context. The following table and Figure 2.6 are the logic to achieve this.

If the ms bit of RS equals 0, it means RS is less than 16, then the register is a $G and RS is used directly. If RS is less than 16+S+L, the register is $L or $S. In both cases all registers are on local processor and the remote signal is set to zero. When RS is larger or equal than 16+S+L, then the register is $D. In this case it is possible to read this register from remote processor if the signal of $D remote/local is 1 and the register address is formed from producer's D_base. Otherwise the register is read locally.

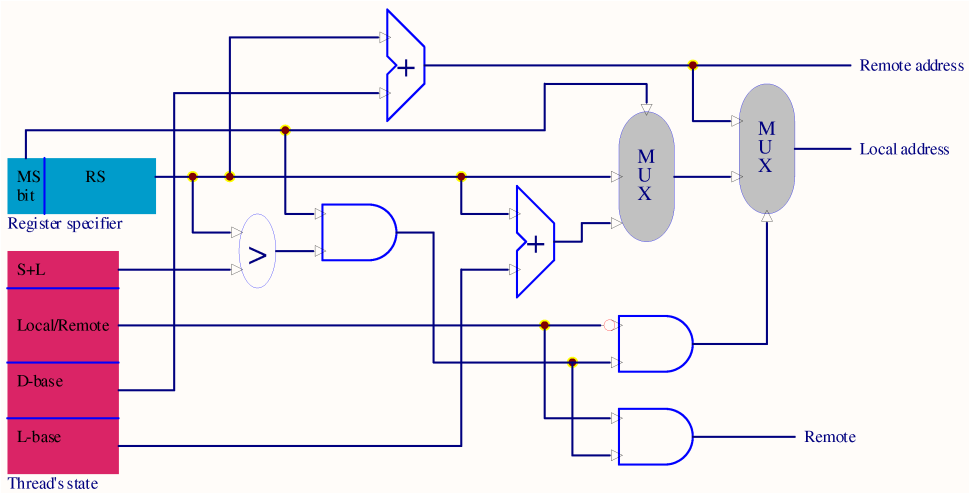

Figure 2.6: The logic of register address decoding.

```
if ( RS < 16)
{
      Address = RS;
      Remote == 0
}
elsif (rs < 16 + S + L)
{
      Address = L_base + RS;
      Remote == 0
}
elsif (rs ≥ 16 + S + L) && (Remote $D == 1)
{
      Remote = 1;
      Address = D_base(producer) + RS;
}
elseif (rs ≥ 16 + S + L) && (Remote $D == 0)
{
      Remote = 0;
      Address = D_base + RS;
}
```

2.2.3 Microthreaded Pipeline

High level architecture of Microthreaded Pipeline

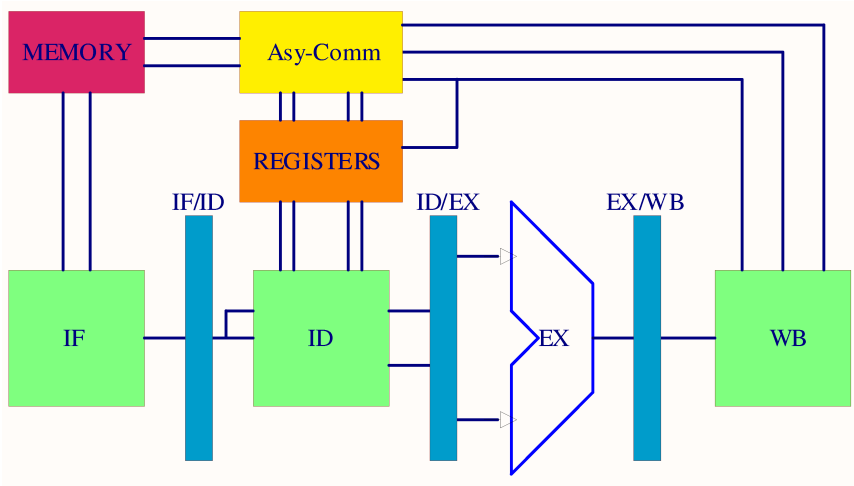

Figure 2.7: Microthreaded Pipeline.

The concept of this Pipeline is formed from a combination of microthread theory and the general MIPS pipeline. Figure 2.7 gives an overview of a simple microthreaded, in-order pipeline with its four stages and the communication interfaces required to implement this model in a distributed manner. The pipeline stages are:

- IF(Instruction Fetch): Thread control and instruction fetch.
- ID(Instruction Decode): Instruction decode/register read and reschedule
- EX(Execution): Arithmetic and logic operation execution or address calculation.
- WB(Write Back): Result is written into a register or written asynchronously to the register file.

Microthreaded Pipeline Datapath & Control

- Instruction fetch:

The main task of IF has no difference between Microthreaded and MIPS pipeline (2.1.2), which is get the proper instruction with the correct address PC. The differences between the Microthreaded and MIPS pipeline IF stage are the update of the PC and a local schedule, which select the new PC either *PC+4*, *PC+8* or a new thread's PC address. It also keeps all threads' states. All instructions to support the microthreaded model requrie only the first stage of a conventional pipeline as they provide control to the local scheduler at the IF stage.

Two instructions at PC and PC+4 are fetched simultaneously. One instruction is launched into the pipeline register and transferred to the rest of pipeline and the other one, if it is a concurrency control instruction, is launched into the local scheduler for the concurrency controls. There is one constraint for this case, the lower address instruction (*IM[PC]*) of the two instructions should not be the thread-control instruction and the compiler must guarantee this.

For the case the instruction *IM[PC+4]* is a conventional ISA, then the new PC will be PC+4 in the next cycle. Otherwise it means the instruction is a microthreaded instruction. In this case there are two possibilities. If the instructions following the current one is a Swtch, Kill or Bsync, then the instruction can force a context switch and also if there is at least one thread ready to be run in the local scheduler, the new PC will be the new thread PC and transferred to the instruction memory. Otherwise if the instruction can not cause a context switch, the new PC will be *PC+8* instead of *PC+4* (Create or Break). Following is the pseudo code to describe this:

```
if ( IM(PC+4) ∈ RISC ISA )
      PC = PC + 4;
elsif ( IM(PC+4) ∈ { Swch, Kill, Bsync } )
      PC = new thread PC;
else
      PC = PC + 8;
```

- Instruction decode:

  This is similar to the MIPS Pipeline(see section 2.1.2), the instruction stored in the IF/ID pipeline register supplies the register numbers which are decoded through the *address unit* (see section 2.2.2) for reading/writing registers.

There are two special cases in reading and writing the register file that require explicit communication. Writing to the global window is an action which not only writes locally but also the address and data must be broadcast to all other processors to update the same location on all processors. This broadcast communications is triggered by a write to a register with a specifier in the range$\{0...15\}$. In this case the data is written locally as normal but also sent to a broadcast bus, which duplicates the write in every other processor. The second special case is for reads to the dependent window, when producer and consumer are mapped to different processors. A failed read to the dependent window, will also trigger a remote read of the data via the ring network. If the data is not immediately available on the remote processor, the request is suspended in the producer's shared window until the data is produced. After it returns, via the network, wakes up the suspended threaded in the dependent window of consumer thread. All these communication are asynchronous and independent of the pipeline's operation.

The compiler places a *Swch* instruction after any branch of control and an instruction whose register operands are not statically guaranteed. When the *Swch* follows a branch instruction, it has an operation that sends the resolved PC, which is possible the branch/jump address or the *PC+8* back to the *Local Scheduler* along with a write request signal. In the other case, when the register operand is read fail, it has two operations, one is writing the current PC to the scheduler to let this thread wait for the data until it is available. Another operation is that to let the *slot number* pass down the pipeline with the thread's state and write it back to the register using the normal write-back process. In order to realize this operation and not make any changes to later pipeline stages, the control logical just change the *Execution* stage to do an logic *and* operation of the two *Slot Number* as the operands. The logic is following:

```
if(Instruction == Jump)
      PC = Jump Address;
elsif(Registers == Empty)
{
      PC = Current PC;
      Registers = Slot number;
}
elsif(Branch == taken)
      PC = Branch Address;
else
      PC = PC+8;
```

The control of the Microthreaded Pipeline is very simple if it is compared to the conventional MIPS pipeline (see section 2.1.3). It eliminates all the control logic to deal with the pipeline bubbles. It does not need the complex logic for example for *Branch prediction* and *Pipeline Flush*.

Until now we can easily understand why the Microthreaded Model can obviously improve the instruction-level prarllelism (ILP) and thread-level parallelism (TLP). Once there are threads available, the processor will never be suspended for resolving the branch address or waiting the data to be available. For the branch, the context switch avoids having to implement branch prediction and fills the pipeline with instructions from other threads. The thread has a deterministic delay before being rescheduled. Also once the data dependencies for an instruction are resolved, the scheduler can be signalled to resume execution of the suspended thread immediately.

- Execution & Write back:

  In theory, these stages do not have a big difference with the MIPS pipeline, which is described in section 2.1.2. The obvious difference is that the Microthreade pipeline communicate with the data memory through an *Asynchronous Communication*.

Microthreaded Pipeline Hazards

Because of the reasons already mentioned above, the Microthreaded Pipeline does not have Branch Hazards which is described in section 2.1.4. For the Data hazards and forwarding, the bypass can be just simplified as follows:

```
if(EX/WB.RegWrite
and (EX/WB.RegisterRd≠0)
and (EX/WB.RegisterRD=ID/EX.RegisterRs))
Bypass data from register EX/WB.
```

Figure 2.8 shows the final evolved datapath and control for Microthreaded pipeline which includes a local scheduler, address unit and asynchronous interface.

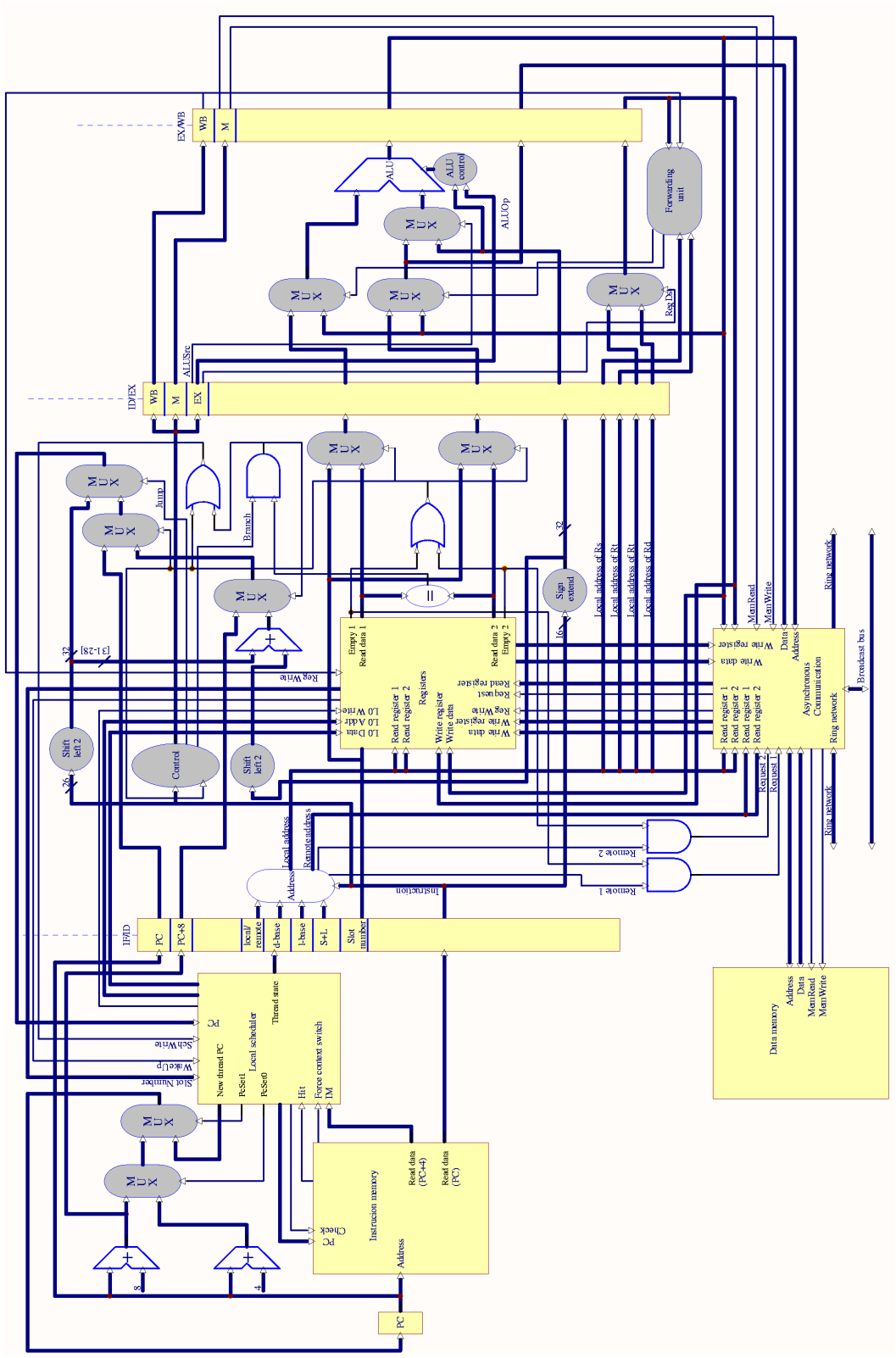

Figure 2.8: The datapath and control for Microthreaded pipeline including a local scheduler and an asynchronous interface.

Chapter 3

Design and Implementation of the Microthreaded processor

Microthreaded model defines instruction level parallelism. It provides latency tolerance through explicit context switching (swch) for control and data hazards. In Chapter 2, we describe the high-level architecture of the microthreaded processor and the low-level pipeline design. In this chapter and chapter 4 we illustrate the design details of this processor realization in VHDL language. And this thesis focuses on the implementation of local scheduler and the whole IF stage.

3.1 Overview

We have indicated the top-level architecture of an IF stage comprising PC selection logic, an instruction memory (cache) and a local scheduler. As described in previous chapter, the parallelization information of the microthreaded is produced by the compiler and kept in the thread control block (TCB). In IF stage, it uses local scheduler to read the information through the TCB to take charge the management when a create instruction executed. The TCB stores the information which determines the number of required registers per thread. A set of registers local($L), shared($S), and dependent($D) should be allocated by each thread before execution. The instruction memory has two blocks which are a code block storing the microthreaded instructions and a TCB containing parameters that describe the family of threads. A family of threads is defined by an iterator including a triple of *start*, *step* and *limit*. And a *thread pointer* (TP) defines the first executable instruction of a thread and it is terminated by the instruc-

tion *Kill*. One or more *Kill* instructions are possible required if the thread has branch (28). The parameters defining the information of micro-contex is also required. All threads share a single contex and each thread has its own register window which is allocated dynamically by the Register Allocation Unit (RAU). We call this dynamical window as *microcontext*. Table 3.1 describes the variables in the TCB that defines the thread family.

Table 3.1: The parameters in the thread control block (TCB) which defines a family of microthreads.

Name	Description	Size
Threads	Cardinality of the set of threads representing an iteration (n)	2 bytes
Dependency	Iteration offset for any loop carried dependencies	2 bytes
Pre-ambles	Number of iterations using pre-amble code	2 bytes
Post-ambles	Number of iterations using post-amble code	2 bytes
Start	Start of loop index value	4 bytes
Limit	Limit of loop index value	4 bytes
Step	Step between loop indices	4 bytes
Locals	No. of local registers dynamically allocated per iteration	1 byte
Shares	No. of shared registers dynamically allocated per iteration	1 byte
$Pre - pointer^*$	One pointer per thread in set for pre-amble code	0/4 bytes
$Main - pointer^*$	One pointer per thread in set for main loop-body code	4n bytes
$Post - pointer^*$	One pointer per thread in set for post-amble code	0/4 bytes

The code block and TCB are all saved inside the instruction memory (I-Cache) due to the simplification. Of course, the TCB block is *.data* type also can be stored in the data memory (D-Cache). The following dependent example loop and pseudo-code can illustrate how to partition these two blocks and the structure of the data and code. These two block has separated address and the TCB area is just read once when creating the microthreads. In chapter 4 we use a similar independent loop to demon-

strate how the microthreaded microprocessor implements depend on the data and code of these two blocks.

$$\boxed{\begin{array}{l} for(i = 1; i < n; i++) \\ \quad M = (M + C[i]) * (A[i] + B[i]); \end{array}}$$

```
Thread control Block:
.data
loop:   .word    2          # threads per iteration
        .word    1          # dependency distance
        .word    1          # loop start
        .word    n          # loop limit
        .word    1          # loop step
        .word    6          # number of local registers
        .word    1          # number of shared registers
        .word    p          # point to code fragment
        .word    q          # point to code fragment
```

```
Code Fragments:
.code
main:   cre      loop       # create family of threads
        Bsync
p:      Lw       $L1 A($L0)
        Lw       $L2 B($L0)
        Add      $L3 $L1 $L2
        Kill
q:      Lw       $L4 C($L0)
        Add      $L5 $D0 $L4
        Swch
        Mul      $S0 $L3 $L5
        Kill
```

3.2 Local scheduler

3.2.1 Register Allocation Unit

The hardware supporting dynamical allocation and de-allocation registers to families of microthreads is described in this section. As mentioned above, each thread must allocate a set of registers before thread starting execution. Register allocation unit (RAU) within each scheduler models the allocation

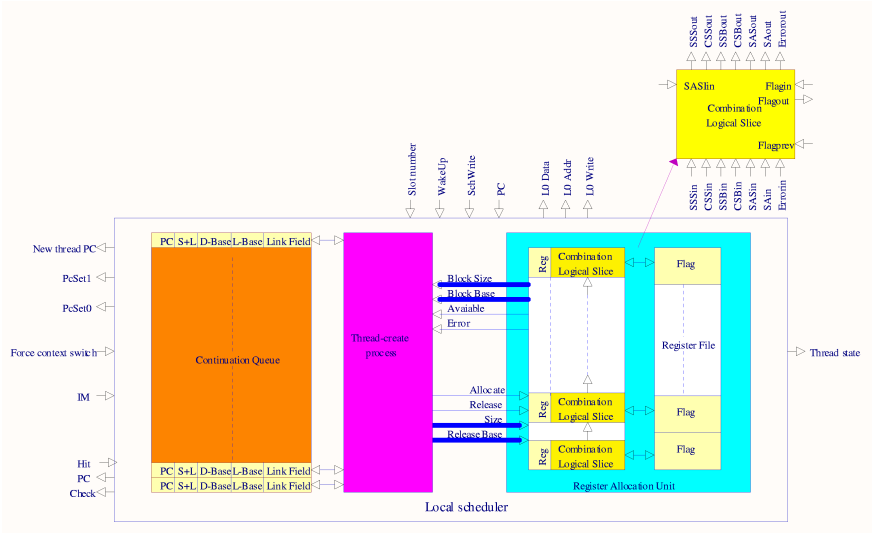

Figure 3.1: Block diagram of the RAU, CQ and the interface with the thread create process.

of micro-contexts to the local register file and determines when new microthreads may be allocated. It uses the dynamical mechanism to check if there are enough required number of registers for each thread prior to thread scheduling and deallocate registers when the thread is terminated. If registers are available it will allocate a micro-context and then create entries in the continuation queue (CQ). It needs to find a group of contiguous free registers to satisfy the request from the thread-create process and release it after *Kill* instruction. When all the thread associated with a micro-context have been killed, its registers will be relinquished and the RAU will update its allocation model (27). The RAU utilizes a set of 1-bit flags to model the allocation states of the registers and it is initialized to be free (set to 0) when processor resetted.

Initially there is one free block and it contains all available free registers can be used by other threads. These registers do not include the register 0-31 which is used by the main thread. The free block is split to allocated blocks with the requested size from the thread-create process and mark remaining blocks as free. Figure 3.1 shows the structure design of RAU and the (de-)allocation interface with the rest part of the local scheduler. As show, the RAU has an iterative array of allocation slices, one slice per n registers. In this design, we just choose n=1 to realize a RAU and we can evaluate $n > 1$ after full development. RAU responses the request action

and gets the information from the thread-create process and the needed information, i.e the base address and slice available, will be sent back after (de-)allocation. The information propagates through the slices to determine the base address and the size of the largest free block, the base address and size of the current free block and whether it has free block. When the space is available and its size is larger than the size of the required registers for a given thread, the thread-create process supplies the required block size and an allocate request. An error flag is also propagated through the slices to indicate whether the inputs are appropriate or not (27). The data manipulated and propagated between slices is listed in table 3.2.

Table 3.2: Allocation parameters.

Signal	Description
SSS	Selected Slice Size
CSS	Current Slice Size
SSB	Selected Slice Base
CSB	Current Slice Base
SAS	Set allocate Size
SASI	Slice allocate Size in
SA	Slice Available
Error	Error signal
Flagin	Current flag state
Flagprev	Previous flag stage
Flagout	New flag state

There are two challenges to design this hardware unit. The first challenge is that the registers should be fully utilized. The second one is how to perform the (de)allocation in a minimum/constant number of cycles. For the first challenge, this is also the issue that we should always concern how to balance the design complexity with efficiency. The complexity of this allocation scheme is proportional to $O(\frac{R}{n})$, R is the size of the register file and n is the number of registers allocation in a unit of allocation. As mentioned above, we use one slice per register (n=1) in our VHDL implementation and this has top efficiency and the highest complexity as well. In order to reduce the complexity of the allocator, we can use n greater than 1, and it also reduce both area and propagation delay. However, we need consider the inefficiency in register use because of the inherent disadvantage of block algorithm. It causes the unused register fragments unless these registers are released by the thread or these unused contiguous registers

are large enough for allocating to new thread. In (27), the authors also indicate that any low efficiency to allocate registers can be minimized by optimization of compiler, which can enable this allocation scheme to fully manage the overhead associated with dynamic allocation. Cycles for implementation is always a critical issue for allocator design. This allocation design is straightforward. We use signals which propagate though slices of allocation logic directly. The signal propagation can get minimal delay which is one cycle when this allocation scheme allocates one micro-context. When this allocator has no any action is being operated, the RAU still can calculate the needed information, so that it is available before next request.

The algorithm implemented by the RAU is described below (27), where we have N combinational logic slices, note that the description of the abbreviations we used are illustrated in table 3.2.

- Initialization.
  To decide if the space is available or no error in pervious process. If it is available and correct, to find the base address and size of the largest free block in the register file.
- When to do a allocation.
  If the space is available and the size of the largest 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 block. Otherwise, just wait and calculate the needed information for next allocation.
- When to do a release.
  To set the flags as empty start from the beginning (base) address.

Below is the simplified pseudo-code of register allocation algorithm.

```
if((Do_allocate=1)and(SASI>0)and(SASin=0)and(flagin=0))then
    SASout = SASI - 1;
    flagout = 1;
elsif((Do_allocate=1)and(SASI=0) and (SASin>0)and(flagin=0))then
    SASout=SASin-1;
    flagout=1;
elsif((Do_allocate=1)and((SASI>0)or(SASin>0))and(flagin=1))then
    Errorout=1;
end if;
if((Do_release=1)and(SASI>0)and(SASin=0)and(flagin=1))then
    SASout=SASI-1;
```

```
    flagout=0;
elsif((Do_release=1)and(SASI=0)and(SASin>0)and(flagin=1))then
    SASout=SASin-1;
    flagout=0;
elsif ((Do_release=1)and((SASI>0)or(SASin>0))and(flagin=0))then
    Errorout=1;
end if;
if(Flagin=0)and(Flagprev=0)and(Do_allocate=0)and(Do_release=0)then
    if(CSSin≥SSSin) then
        SSSout = CSSin + 1;
        SSBout = CSBin;
    else
        SSSout = SSSin;
        SSBout = SSBin;
    end if;
    CSSout = CSSin + 1;
    CSBout = CSBin;
    SASout = 0;
    SAout = 1;
elsif(Flagin=0)and(Flagprev=1)and(Do_allocate=0)and(Do_release=0)then
    if(CSSin>SSSin) then
        SSSout = CSSin;
        SSBout = CSBin;
    else
        SSSout = SSSin;
        SSBout = SSBin;
    end if;
    CSSout = 1;
    CSBout = slice_id;
    SASout = 0;
    SAout = 1;
elsif(Flagin=1)and(Flagprev=0)and(Do_allocate=0)and(Do_release=0)then
    if(CSSin>SSSin) then
        SSSout = CSSin;
        SSBout = CSBin;
    else
        SSSout = SSSin;
        SSBout = SSBin;
    end if;
```

```
    CSSout = CSSin;
    CSBout = CSBin;
    SASout = 0;
    SAout = 1;
elsif(Flagin=1)and(Flagprev=1)and(Do_allocate=0)and(Do_release=0)then
    if(CSSin>SSSin) then
        SSSout = CSSin;
        SSBout = CSBin;
    else
        SSSout = SSSin;
        SSBout = SSBin;
    end if;
    CSSout = CSSin;
    CSBout = CSBin;
    SASout = 0;
    SAout = SAin;
end if;
```

3.2.2 Continuation Queue

The microthreaded execution model dynamically creates threads via the local scheduler as and when required by executing instructions generated at compile time. Figure 3.1 shows the intra-actions between Continuation Queue (CQ) with Thread-create process and RAU within the Local scheduler. The continuation queue can accept one new thread per cycle from the I-cache if RAU is available. A continuation queue has a table, a link memory and several processes, which hold and manage the state of all allocated threads. The slots are addressed by slot number which is used as the reference to a thread. Each slot of the table comprises a program counter, two base addresses for the dynamically allocated registers and the sum of the number local and shared registers allocated. The register base addresses are local base which is the base address of its micro-context, dependent base which is the base address of a dependent micro-context if used, and includes a flag to specify this is local or remote address. It also includes a link field which is a pointer to another slot in the table. This point is used to build continuation queues which are empty queue, ready queue and a queue for threads suspended on a register. All these components are shown in table 3.3. In our VHDL realization, for a 32 bits PC, a 1K register file and a 512 slots CQ, each slot in the continuation queue needs 67 bits.

Table 3.3: Components of a slot in the continuation queue for 512 slots and 1K entry register file.

Field Name	Size
PC	32 bits
Local Base	10 bits
Dependent Base	11 bits
L+S	5 bits
Link Field	9 bits

Thread state is stored within a slot and maintained by several processes and link queues. All the thread slots are organized into three queues which are used to manage the empty, active and suspended slots. Each queue has two registers used to maintain pointers. The *head* and *tail* registers which are the first and last pointers of the chain. For example, for the empty queue, the *head* register points to the first empty slot number. The link field of each empty slots is used to point to the next empty slot within the queue. And another one is the *tail* register which indicates the last empty slot number of this queue. By default, any operations, i.e. create or kill, just can operate on these two slots. We can simplify this operation into how to maintain the pointer within the link queues. It is also one of the main tasks of the continuation queue.

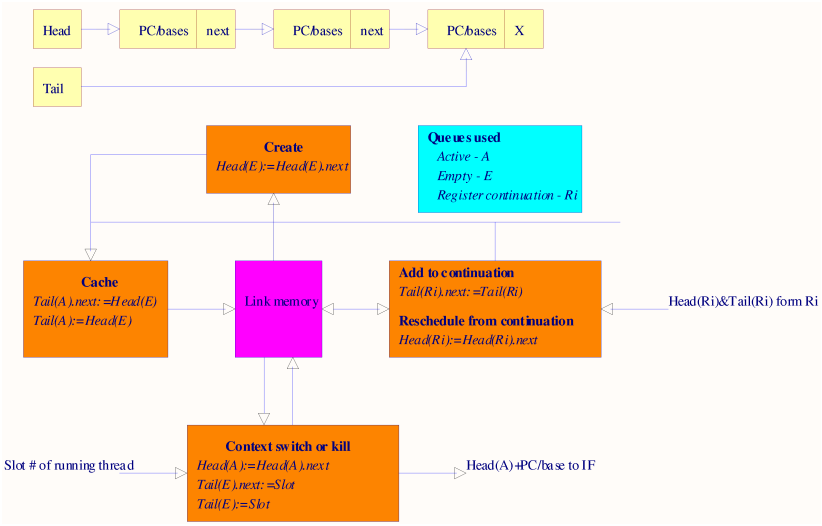

Figure 3.2: The processes and interactions within the continuation queue.

The continuation queue has several processes which are *code prefetching*, *thread creating*, *thread kill* and *context switch*. The process *code prefetching* runs during the whole execution period. When the prefetched instruction is conventional ISA, it sets the next PC as PC+4 and has no difference with a conventional processor. And when this prefetched instruction is ISA-μt, it sets the next PC as PC+8 or new thread PC. The processes *thread creating*, *thread kill* and *context switch* can all be trigged by *thread creating* process in IF stage. But for the *context switch* process, it also can be trigged by the results occured from the ID stage when the cases, i.e. branch or register ready, to wake-up and reschedule the threads. This case is also a special case of *context switch* process. In this case, the process also manages the continuation queues of threads suspended on a given register. Figure 3.2 shows the relationship between those processes. The logic of them are described in more detail below.

- Code Prefetching

```
if (instruction = ISA) then
    PC=PC+4;
elsf (instruction = Cre) then
    Stall the pipeline;
    PC=Address of TCB;
    Trigger thread creating process;
elsf (instruction = Bsync) then
    Synchronization for thread creation;
    Read all parameters of TCB from D-cache;
    Resume the pipeline;
elsf (instruction = Swch) then
    Trigger context switch process;
elsf (instruction = Kill) then
    Trigger thread kill process;
end if;
```

- Thread Creating

```
if (Create process) then
    Check PC whether within the I-cache;
    if (PC_ hit = 1) then
        if ((L0 ≤ Loop_Limit) and (Space_Available = 1) and (L+S ≤ Slice_Size)
            and (input_error = 0) and (Empty_Link_Available = 1)) then
            Tail.A.next = Head.E
            Tail.A = Head.E
            Head.E = Head.E.next
            RR_Write = 1;          - - Write value of $L0 to register file
            RR_L0 = L0;
            RR_L0_Address = Allocate_Base;
            Write PC to continuation queue;
            Write S+L to continuation queue;
            Write L base to continuation queue;
            Write D base to continuation queue;
            Do allocate = 1;       - - Let the RAU allocate the register file
            Required_Alloc_Size = Local_registers + Shared_registers
            if (Empty_Link_Head = Empty_Link_Tail) then
                Empty_Link_Available = 0;
            end if;
        end if;
        L0 = L0 + Step_loop;
    end if;
end if;
```

- Thread Kill and Context Switch

```
if (Context switch or Kill process) then
    if (Active_Link_Head ≠ Active_Link_Tail) then
        LS_Stall = 0; – Latch a new thread
        PcSet0 = 0;
        PcSet1 = 0;
        New_thread_pc = PC(Active_Link_Head);
        Thread_Local_Remote = 0; – Thread state output
        Thread_D_base = D_base(Active_Link_Head);
        Thread_L_base = L_base(Active_Link_Head);
        Thread_S_L = S_L(Active_Link_Head);
        Thread_Slot_number = Active_Link_Head;
        Head.A = Head.A.Next
        if (Kill process) then
            Tail.next:=Slot
            Tail.E:=Slot
        end if;
    elsif Active_Link_Avaiable = 1 then –If it is the last active link
        LS_Stall = 0; – Latch a new thread
        PcSet0 = 0;
        PcSet1 = 0;
        New_thread_pc = PC(Active_Link_Head);
        Thread_Local_Remote = 0; – Thread state output
        Thread_D_base = D_base(Active_Link_Head);
        Thread_L_base = L_base(Active_Link_Head));
        Thread_S_L = S_L(Active_Link_Head));
        Thread_Slot_number = Active_Link_Head;
        if (Kill process) then
            Tail.next:=Slot
            Tail.E:=Slot
        end if;
        Active_Link_Avaiable = 0;
            – The last active link is used up, all the slots are
empty.
    end if;
end if;
```

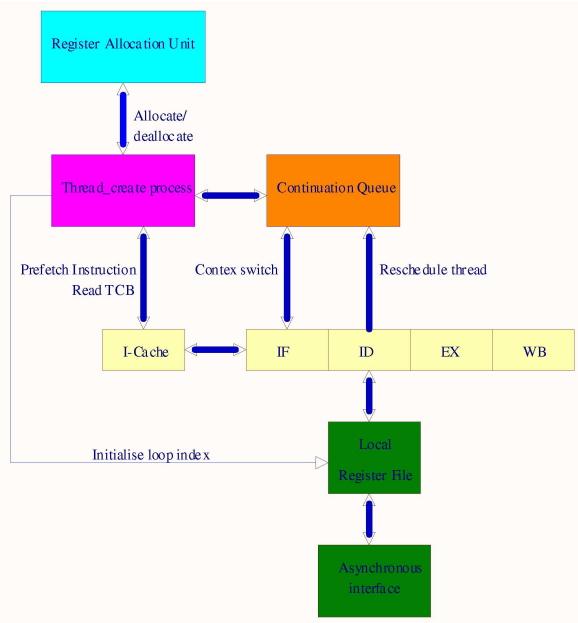

Figure 3.3: The datapath and detail of Local scheduler.

3.3 Put all together

In this project, we finish designing the low-level architecture of Microthread pipeline and implementing the first stage of it. The design and implementation of a local scheduler is the main contribution to the microthread processor. Fingure 3.3 shows the detail of local scheduler. It shows its main components and the datapath between it and other stages of the pipeline. The simulation results of those separated components and a full simulation are illustrated in chapter 4.

Chapter 4

Simulation results

In this Chapter, we show some preliminary simulation results of this project. We simulated the IF stage of the microthreaded pipeline including the local scheduler which is the most important component within a microthreaded processor. The results are preliminary as we just show the relationship between independent threads and can not show the results when threads running on multi-processors. And also a microthreaded compiler is required before we can simulate complete applications.

4.1 Register Allocation Unit

What we need to test the behavior of Register Allocation Unit (RAU) which is described qualitatively below:

- If the RAU is ready, the scheduler can initiate an allocation immediately, which completes in a single cycle.
- Find the start address and the size of the first and largest free block in the register file.
- When an allocation occurs, flip the flags from the start address with the request size.
- When to do the release, set the corresponding flags of that block to be free in the register use model.
- Set available signal when it has free registers.

In order to illustrate the results as simply as possible, we just use 32 registers which is a large enough size for the test. The test bench that we design is described with following figures:

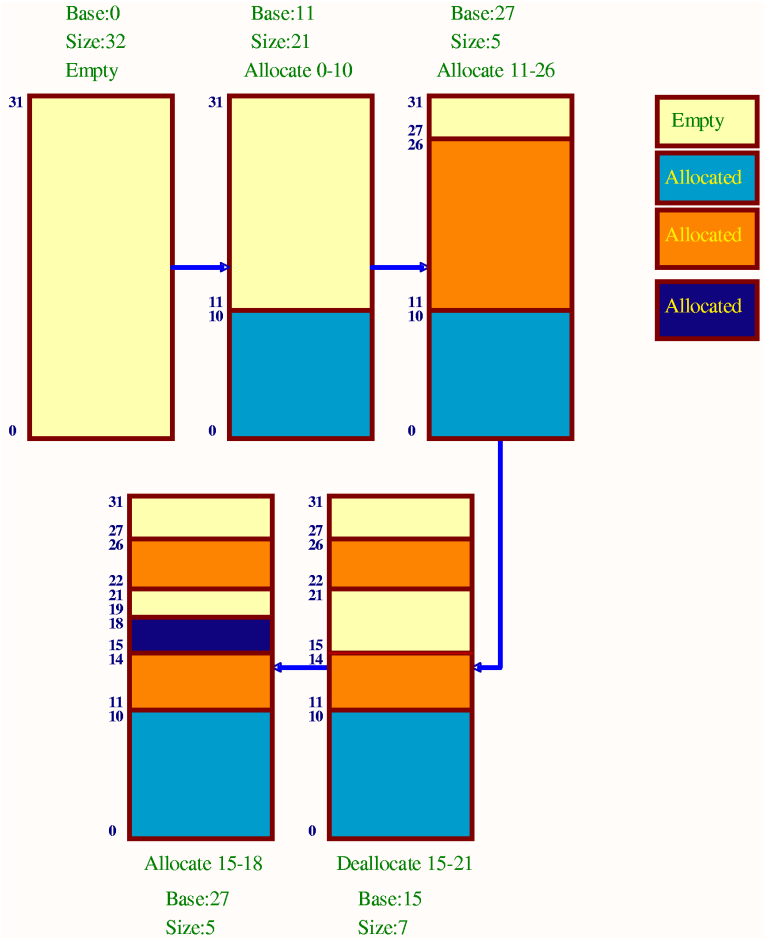

Figure 4.1: The test bench of Register Allocation Unit, showing the sequence of allocation states in the test

Figure 4.2 and 4.3 show the last VHDL simulation results of RAU. We can see those sequent results are the same with the prediction from the test bench. First, we have a big free registers file. All of the flags are set to be 0 and the slice size output of RAU is set to 32. And then it gets a allocation request in size 11 and 16. After those requests, the largest address is 27 and size is 5. Later it gets a release request in size 7 and based on 17. Now the

Table 4.1: The sequence of allocation states.

Event	Size	Release base	Base address returned
Allocate	11	/	11
Allocate	16	/	27
Deallocate	7	15	15
Allocate	4	/	27

current largest address and size are 17 and 7. The last request is allocation in size 4, so after this case the largest address is changed back to 27 and the largest free size is 5. Figure 4.2 shows the result of input and output ports. Figure 4.3 shows the change of the flags. The cycle time is 10*ns*. In a single cycle, it creates the correct information 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.

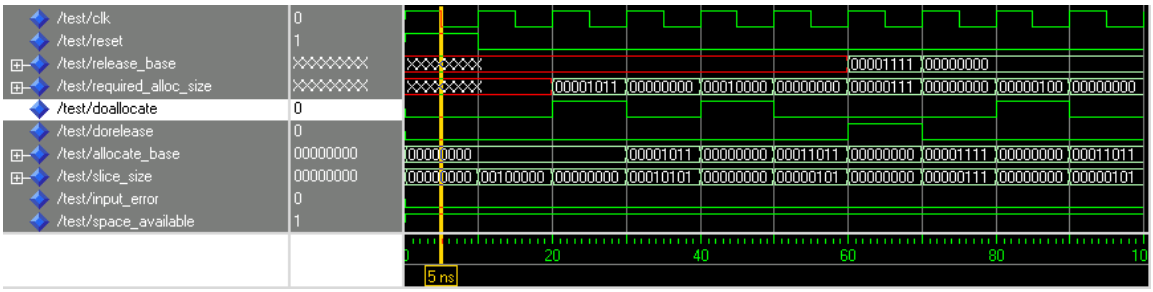

Figure 4.2: The VHDL simulation result of the Register Allocation Unit. This graph displays the changes to the interface to the register allocation logic.

4.2 Local Scheduler

The continuation queue does not need to be test independently. We can test it by testing the whole local scheduler. Before testing the local scheduler, we will illustrate our test bench. It also shows how both loop and in-line forms of concurrency are expressed in microthreaded model. Consider the code generated from a simple loop given below in C and assemble language:

$$\begin{array}{|l} for(i = 1; i < 10; i++) \\ \quad C[i] := A[i]^2 + B[i]^2; \end{array}$$

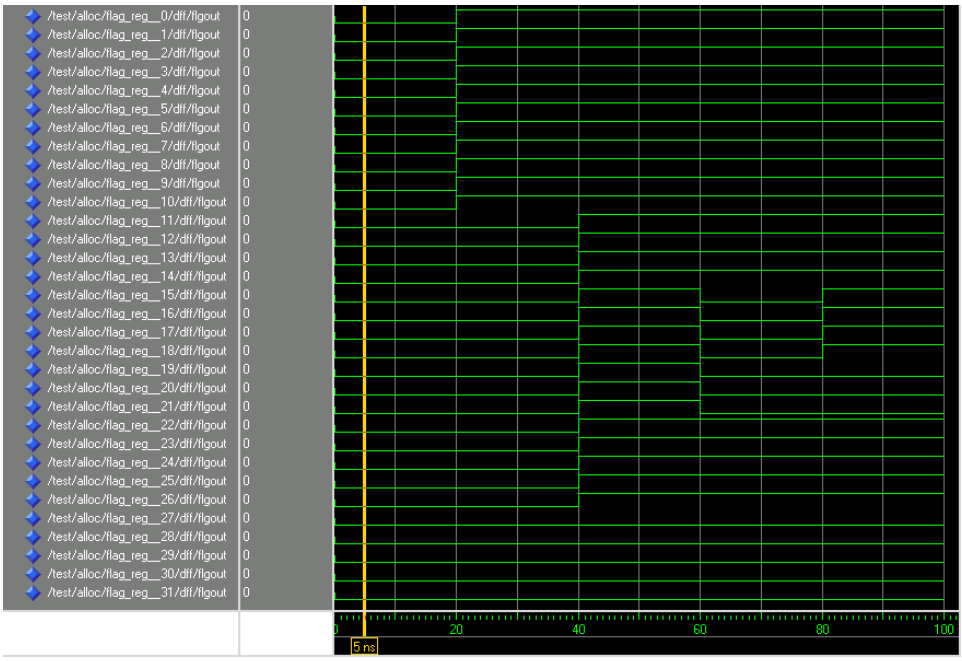

Figure 4.3: The VHDL simulation result of Register Allocation Unit. This graph displays the value of each of the 32 flags in the register allocation model. When the flag is at logic low, the corresponding register is not allocated. A transition to logic high shows an allocation and a transition from high to low shows an unallocation.

```
Thread control Block:
.data
loop:   .word   2       # threads per iteration
        .word   0       # dependency distance
        .word   1       # loop start
        .word   10      # loop limit
        .word   1       # loop step
        .word   5       # number of local registers
        .word   0       # number of shared registers
        .word   p       # point to code fragment
        .word   q       # point to code fragment
```

```
Code Fragments:
.code
main:   cre     loop                    # create family of threads
        Bsync
p:      Lw      $L1 A($L0)
        Mul     $L2 $L1 $L1
        Kill
q:      Lw      $L3 B($L0)
        Mul     $L4 $L3 $L3
        Swch
        Add     $L3 $L2 $L4
        Swch
        Sw      $L3 C($L0)
        Kill
```

First note that both A[i] and B[i] are squared independently. The loop can therefore be executed using a family of microthreads comprising 20 threads. For each i, the thread p loads and computes $A[i]^2$, and thread q computes $B[i]^2$, waits for $A[i]^2$, completes the summation and stores the result. Instructions in these threads execute in order and no branch prediction is required. Below is the binary code that is hand compiled from above test program fragment. The *.Code* segment is from address *0x100* and the *.TCB* segment begins from address *0x400*.

```
clear
address 64
00000000                - -00000100 nop
70000400                - -00000104 Main:Cre loop
00000000                - -00000108 nop
68000000                - -0000010c Bsync
8c110010                - -00000110 p:Lw $L1 A($L0)
02320018                - -00000114 Mul $L2 $L1 $L1
64000000                - -00000118 Kill
8c310010                - -0000011c q:Lw $L3 B($L0)
02740018                - -00000120 Mul $L4 $L3 $L3
60000000                - -00000124 Swch
02549820                - -00000128 Add $L3 $L2 $L4
60000000                - -0000012c Swch
ac530010                - -00000130 Sw $L3 C($L0)
64000000                - -00000134 Kill
address 256             - -Loop:Thread control block
00000000                - -00000400 Empty
00000002                - -00000404 Threads per iteration
00000000                - -00000408 Dependency distance
00000001                - -0000040c Loop start
00000010                - -00000410 Loop limit 4096
00000001                - -00000414 Loop step
00000005                - -00000418 Number of Local registers
00000000                - -0000041c Number of shared registers
0000011c                - -00000420 Pointer to code fragment p
00000110                - -00000424 Pointer to code fragment q
00000108                - -00000428 Return to instruction memory
```

Because the time reason, we restrict the qualification of this local Scheduler that it runs independent in the IF stage on a single processor. And no dependence between those threads. The test elements about the behavior of Local Scheduler are described below:

- If the I-cache is ready, the scheduler can initiate an allocation immediately, which completes in a single cycle.
- The local scheduler can read the binary code from the I-cache. And can partition different segments of the code.
- It can read all parameters in the Thread Control Block (TCB). And completes reading one element in one cycle

- When the scheduler is reading the TCB, it can stall the pipeline. And after it finishes reading TCB it can resume the pipeline.
- After reading the TCB, it also can trigger the thread-create process. The program begins to run from .code segment.
- Send the request to the I-cache to test if the code inside the cache. And pre-fetch the instruction on every cycle.
- Schedule and re-schedule the threads, store the thread inside continuation queue and manage the corresponding *head* and *tail* registers.

The figure 4.4, 4.5 and 4.6 show the last VHDL simulation results for the local scheduler. The results presented in Figure 4.4 which shows the first microsecond results. It shows that the microprocessor begins to read the TCB from the I-Cache after initialization. Figure 4.5 is the second microsecond simulation result. This graph displays that the microprocessor begins to create the threads after half cycle when it finishes reading the TCB parameters. From this figure we can see that reading TCB parameters needs ten cycles. And half cycle later it begins to latch the first active thread. So it is just one cycle delay for starting the threads. The total cycles delay for initialization are 11. In this figure, it also shows the local scheduler begins to do the first register allocation at the same time with the thread creation. Figure 4.6 is the third microsecond simulation result. It shows the behaves which are latching and switching the threads. It also shows the time for create the last thread. Meanwhile, it finishes sending the request to the RAU for register allocation. This figure also proves that the we can create one thread just using one cycle. For all of those figures, we do not show the *kill* action and the *deallocation* request to the RAU. When the context switch happens, all the threads are suspended on the register file. The wake up mechanism is realized on this stage. Because we do not implement the second stage of microthread pipeline, we do not simulate the signal for waking up the suspended threads. But these behaves can be easily simulated later after the full development. From what we mentioned above, we can see that the functions of this local scheduler satisfy the desire of this project.

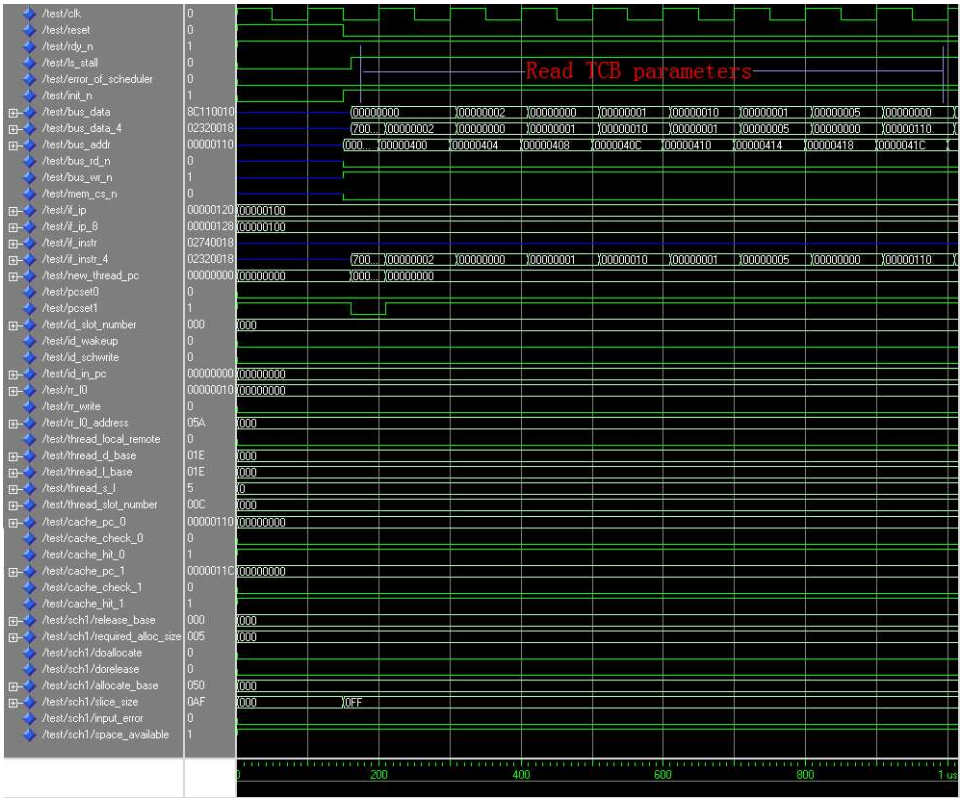

Figure 4.4: The first microsecond VHDL simulation result for the Local Scheduler. This graph displays that the microprocessor begins to read the TCB from the I-Cache after initialization.

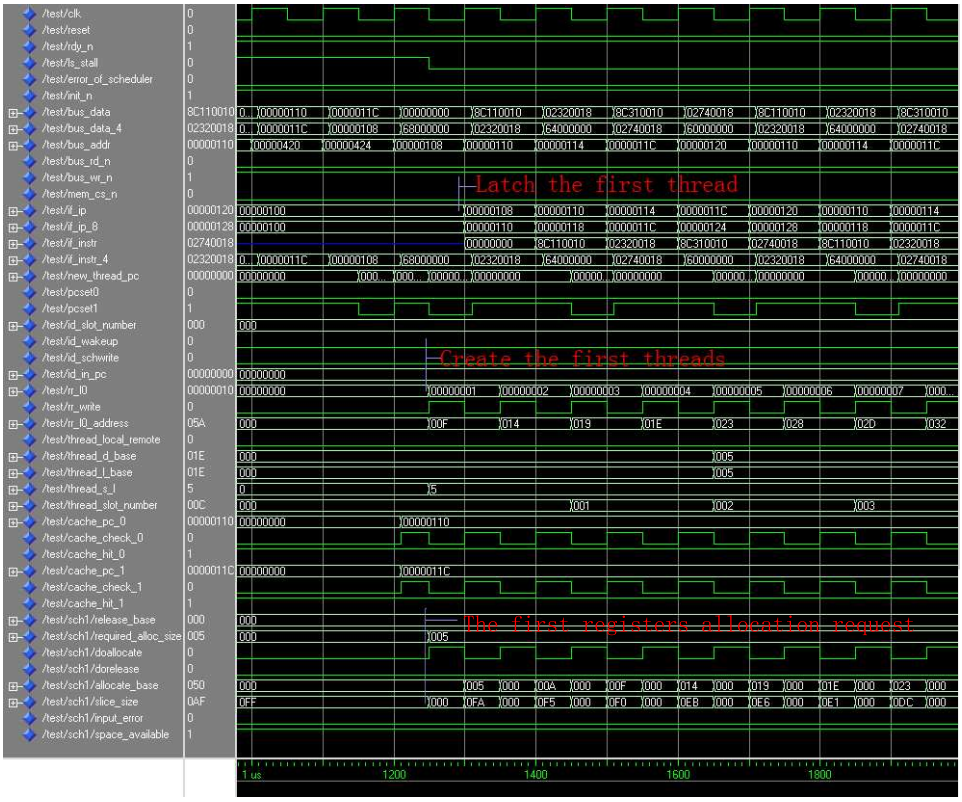

Figure 4.5: The second microsecond VHDL simulation result for the Local Scheduler. This graph displays that the microprocessor begins to create the threads after half cycle when it finishes reading the TCB parameters. And half cycle later it begins to latch the first active thread. So it is just one cycle delay for starting the threads. It also shows the local scheduler begins to do the first register allocation at the same time with the thread creation.

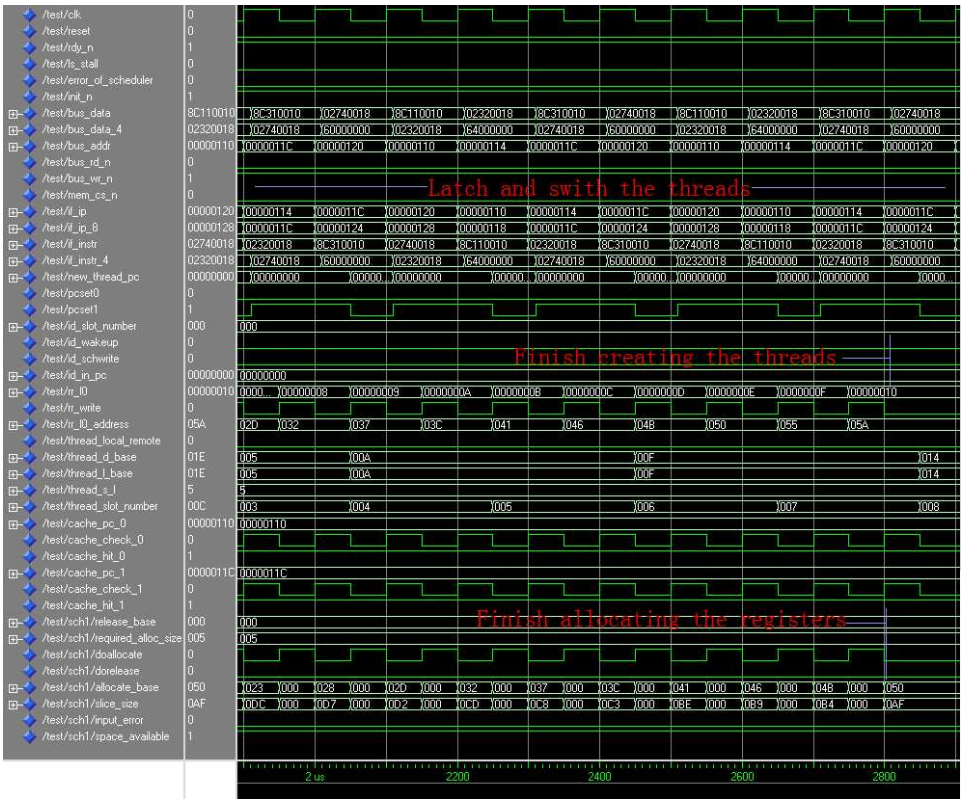

Figure 4.6: The third microsecond VHDL simulation result for the Local Scheduler. It shows the behaves which are latching and switching the threads. It also shows the time to create the last thread. Meanwhile, it finishes sending the request to the RAU for register allocation.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This project is the implementation of a microthreaded microprocessor in VHDL. This process is based on a model which decomposes a sequential program into samll fragments of code called microthreads. This model can obviously improve the instruction-level prarllelism (ILP) and thread-level parallelism (TLP), as it can eliminate speculation and memory lantency by scheduling and rescheduling new threads if it has available one. The microthreads are scheduled dynamically and can communicate and synchronise with each other efficiently. The pre-fetching mechanism also can avoids many instruction-cache misses in the pipeline.

During this project, we finished designing a microthreaded processor's pipeline in detail. Microthreaded pipeline has 4 stages. We described its whole architecture which includes a complete data and contrl path. After that, we implemented the first stage of this pipeline, which comprises a local scheduler that is also the most important component of a microthreaded processor. As limited by time, this implementation is just based on a single processor, which has no commnication between neighbour processors. And the program fragmentes executed are independent, which do not have any dependence between each other.

We implemented those functions of this microthreaded processor, which can be summarized as follows:

- It can load the microrthreaded program fragmentes including thread control block (TCB) into the instruction-cache.
- The local scheduler can be initialized by the parameters in the TCB.

- By the pre-fetching mechanism, the local scheduler can test wehether the instruciton-cache misses or not. And also can decide whether the instruction following the current one is ISA-μt or not.
- Select the corresponding program counter (PC) depending on the prefetched instruction.
- It can dynamically allocate and de-allocated registers to families of microthreads.
- Those threads can be dynamically scheduled and re-scheduled on demand.
- It can propagate the thread state from IF stage to the next stage.

5.2 Future Work

Untill now, we already designed the complete structure of the microthreaded pipeline and realized its first stage which comprises the most important component — local scheduer. It is a good start of this ambitious project. Later work can implement other components including the register file, asynchronous communicator. It can be implemented under much more complex conditions. The threads can be distributed into multi-processor. After the design pass the simulation, it should be synthesised into FPGA, which we can test it under real environment. This design should also can be implemented as a SOC system or become a large CMP system in the future.

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[2] A. Bolychevsky, C.R. Jesshope, and V.B. Muchnick. Dynamic scheduling in risc architectures. *IEE Trans. E, Computers and Digital Techniques*, 143:309–317, 1996.

[3] Jesshope C. R. Microthreading - a model for distributed instruction-level concurrency. unpublished, http://staff.science.uva.nl/∼jesshope/Papers/%c2%b5-thread.pdf, 2005.

[4] Jesshope C. R. Scalable instruction-level parallelism. *Computer Systems: Architectures, Modelling and Simulation*, Proc 3rd and 4th Int, July 2004. l. Workshhops, SAMOS 2003, SAMOS 2004, (LNCS 3133, Springer), ISBN 3-540-22377-0, pp383-392, presented Samos, Greece, July 2004.

[5] Jesshope C. R. A concurrency model for instruction-level distributed computing. unpublished, http://staff.science.uva.nl/∼jesshope/Papers/NPC%20paper.pdf, 2005.

[6] Chris R. Jesshope. Multi-threaded microprocessors - evolution or revolution. In *Asia-Pacific Computer Systems Architecture Conference*, pages 21–45, 2003.

[7] C. R. Jesshope and Luo B. A microthreaded chip multiprocessor with a vector instruction set. unpublished, http://staff.science.uva.nl/∼jesshope/Papers/isca2002.pdf, 2002.

[8] Bing Luo and Chris Jesshope. Performance of a micro-threaded pipeline. In *CRPITS '02: Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, pages 83–90, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[9] Chris Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 80–88, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Chris Jesshope and Bing Luo. Micro-threading: A new approach to future risc. In *ACAC '00: Proceedings of the 5th Australasian Computer Architecture Conference*, page 34, Washington, DC, USA, 2000. IEEE Computer Society.

[11] Radhakrishna Hiremane. From moore's law to intel innovation-prediction to reality. *Technology@ Intel Magazine*, pages 4–9, April 2005.

[12] R. Ronen, A. Mendelson, K. Lai, S. Lu, F. Pollack, and J. Shen. Coming challenges in microarchitecture and architecture. *Proc. IEEE*, 89(3):325–340, March 2001.

[13] Doug Burger. Tiled architectures, ACACES 2005 (L'Aquila).

[14] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[15] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.

[16] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[17] H. Wang, L. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks, 2003.

[18] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *SLIP '04: Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7–13, New York, NY, USA, 2004. ACM Press.

[19] Wikipedia. http://en.wikipedia.org/wiki/Superscalar.

[20] Michael S. Schlansker and B. Ramakrishna Rau. Epic: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[21] Walter A. Triebel. *Itanium Architecture for Software Developers*. Intel press, July 2000.

[22] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[23] Stephen W. Keckler, Doug Burger, and Chuck Moore. Trips: Extending the range of programmable processors. Computer Architecture and Technology Laboratory, Department of Computer Sciences, The University of Texas at Austin, www.cs.utexas.edu/users/cart.

[24] Theo Ungerer, Borut Robiĉ;, and Jurij Ŝilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[25] Wikipedia. http://en.wikipedia.org/wiki/Chip-level_multithreading.

[26] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of tcc on chip-multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 63–74, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Lan Bell, Nabil Hasasneh, and Jesshope C R. Microgrids and micro-contexts: Support structures for microthread scheduling and synchronisation. submitted to IJPP (Special issue and Proc. 1st MicroGrid Conference, Amsterdam, http://staff.science.uva.nl/ jesshope/Papers/Scheduling_Synchronisation.pdf, July, 2005.

[28] Kostas Bousias, Nabil Hasasneh, and Chris Jesshope. Instruction-level parallelism through microthreading - a scalable approach to chip multiprocessors. *The Computer Journal Advance Access*, 2005. http://staff.science.uva.nl/ jesshope/Papers/ACSAC05.pdf.