Investigation of a simultaneous multithreaded architecture

Marc Torrant

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Investigation of a Simultaneous Multithreaded Architecture

by

Marc Torrant

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in
Computer Engineering

Primary Advisor:  
Dr. Muhammad Shaaban, Assistant Professor

Committee Member:  
Dr. Roy Czernikowski, Professor

Committee Member:  
Dr. Ken Hsu, Professor

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August, 1999
Release Permission Form

Rochester Institute of Technology

Investigation of a Simultaneous Multithreaded Architecture

I, Marc Torrant, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Marc Torrant

9/2/99
Date
Abstract

Many enhancements have been made to the traditional general purpose load–store computer architectures. Among the enhancements are memory hierarchy improvements, branch prediction, and multiple issue processors. A major problem that exists with current microprocessor design is the disparity in the much larger increase in speed of the CPU versus the moderate increase in speed accessing main memory. The simultaneous multithreaded architecture is an extension of the single-threaded architecture that helps hide the performance penalty created by long–latency instructions, branch mispredictions, and memory accesses. Simultaneous multithreaded architectures use a more flexible parallelism, which takes advantage of both instruction–level, and thread–level parallelism. The goal of this project was to design, simulate, and analyze a model of a simultaneous multithreaded architecture in order to evaluate design alternatives. The simulator was created by modifying a version of the Simple Scalar toolset, developed at the University of Wisconsin. The simulations provide documentation for an overall system performance improvement of a simultaneous multithreaded architecture. In early simulation results, performed with the same number of functional units, an improvement in the number of instructions per cycle (IPC) of between 43% and 58% was found using four threads versus a single thread. The horizontal waste rate, which measures the number of unused issue slots, was reduced between 35% and 46%. The vertical waste rate, which measures the percentage of unused issue cycles (no issue slots used in a cycle), was reduced between 46% and 61%. These results are derived from a set of four sample programs. It was also found that increasing the number of certain functional units did not improve performance, whereas increasing the number of other types of functional units did have a significant positive impact on performance.
# Contents

Abstract

List of Figures

List of Tables

Glossary

Acknowledgments

Trademarks

1 Introduction

2 Theoretical Background of CPU Architecture
   2.1 Early Microprocessors
      2.1.1 Pipelining
      2.1.2 Cache
   2.2 Instruction Set Architecture
   2.3 Superscalar Microprocessors
   2.4 Speculative Execution
   2.5 Multithreaded Microprocessors

3 Theoretical Background of Operating System Issues
   3.1 Process Introduction
3.1.1 Process Memory Management
3.2 Process Creation
3.3 Process Termination
3.4 Process States
3.5 System Calls
  3.5.1 Entering the Kernel
3.6 Context Switching
3.7 Single-Threaded OS versus SMT OS

4 Theoretical Background of Simultaneous Multithreading
  4.1 Comparison Between SMT and Single-Chip Multiprocessor
  4.2 Design Limitations of Advanced Microprocessors
  4.3 Additional Benefits
  4.4 Design Tradeoffs
    4.4.1 Thread-Priority Determination
    4.4.2 Register File Issues
    4.4.3 Cache Design
    4.4.4 Additional Resources

5 Proposed Architecture
  5.1 CPU Pipeline Modifications
    5.1.1 Register File
    5.1.2 Instruction Fetch Stage
    5.1.3 Instruction Decode Stage
    5.1.4 Instruction Dispatch/Issue Stage
    5.1.5 Instruction Execution Stage
    5.1.6 Writeback Stage
    5.1.7 Commit Stage
  5.2 Exceptions
# 6 Simulator

## 6.1 Simulated Pipeline
- 6.1.1 Register Update Unit (RUU)
- 6.1.2 Ready Queue
- 6.1.3 Event Queue

## 6.2 Simulator Memory Space
- 6.2.1 Simulator Loader
- 6.2.2 Memory Space Notes
- 6.2.3 Compiling A Program

## 6.3 Exiting the Simulator

## 6.4 Simple Scalar Instruction Set
- 6.4.1 Simple Scalar System Calls

## 6.5 Instruction Flow in Simulator

## 6.6 Approximations, Limitations, and Simulator Tricks

## 6.7 Running The Tools

# 7 Simulation Results

## 7.1 Test Programs
- 7.1.1 Applicability of Results

## 7.2 Time Complexity of Simulator
- 7.2.1 Individual Thread Performance versus System Performance

## 7.3 Issue Methods and Rates
- 7.3.1 Issue Bandwidth
- 7.3.2 Alternate Issuing Schemes

## 7.4 Reduction of Wasted Issue Slots
- 7.4.1 Reduction of Horizontal Waste
- 7.4.2 Vertical Waste

## 7.5 Functional Unit Utilization
8 Conclusions and Recommendations 94
  8.1 Future Work 95

A Simple Scalar Instruction Set 97
  A.1 Load/Store Instructions 97
  A.2 Integer Arithmetic Instructions 98
  A.3 Control Instructions 98
  A.4 Floating-Point Arithmetic Instructions 98
  A.5 Miscellaneous Instructions 99

B Sim-SMT Configuration File 100

C Test Programs 103
  C.1 MY-TEST.C 103
  C.2 MATSOLVE.C 103
  C.3 NEWTON.C 106
  C.4 FP-TEST.C 109
List of Figures

2.1 Standard Five-Stage Pipeline 5
2.2 Comparison of Memory Types, Access Latencies, and Sizes 12
2.3 Typical Superscalar Pipeline [4] 16
3.1 C Program Memory Layout in Xinu OS [2] 22
3.2 C Program Memory Layout in BSD Unix OS [7] 23
3.3 Process State Diagram 25
4.1 SMT Pipeline [4] 34
4.2 General Architecture of SMT 35
4.3 General Architecture of Single-Chip Multiprocessor 36
5.1 Fetch Unit for Proposed Architecture 52
5.2 Modified Tomasulo Approach for SMT 59
6.1 Pipeline of Simulator 67
6.2 Simple Scalar sim-outorder and sim-SMT Memory Space 71
6.3 Simple Scalar Instruction Format 74
7.1 IPC versus Number of Threads 82
7.2 Simulation Time versus Number of Threads 84
7.3 Individual Program IPCs versus System IPC 85
7.4 Issue Rate versus Number of Threads 86
7.5 IPC and Issue Rate versus Issue Bandwidth 87
7.6 Horizontal Waste Rate versus Number of Threads
7.7 Vertical Waste Rate versus Number of Threads
7.8 Functional Unit Utilization versus Number of Threads
7.9 No Functional Unit Available Count versus Number of Threads
# List of Tables

3.1 Process State Transition Description  
3.2 Comparison Between Single-threaded OS and SMT OS  
4.1 Comparison Between the Superscalar and SMT Issuing Patterns  
5.1 Register File Description  
5.2 Functional Units  
5.3 Information Forwarded Via the Writeback Stage  
7.1 Alternate Issuing Technique Results  
7.2 Alternate Functional Unit Configuration Results
Glossary

**Arithmetic Instruction**: page 16

Instruction in which a mathematical operation such as ADD, SUBTRACT, MULTIPLY, or DIVIDE is applied to either integer or floating point operands.

**BTB**: page 17

Branch Target Buffer. This is a cache that stores the predicted value of the address that will be branched to when a particular branch is taken.

**Branch Prediction**: page 17

A method used to reduce the penalty from control hazards by predicting whether a particular branch is taken or not.

**Branch Misprediction**: page 17

When a branch is incorrectly chosen as taken or not taken.

**Cache**: page 8

A smaller, faster set of memory that is used to hold the most recently used addresses. Cache helps reduce the overall memory access time.

**CDB**: page 18

Common Data Bus. Data bus used to forward the results of the functional units to all other pending instructions. Enables data forwarding in the Tomasulo approach.
CPI: page 4
Cycle Per Instruction.

Commitment: page 18
The phase of the operation where the state of the machine is changed. This is the last stage in the pipeline. At this point, the registers are updated. Also during this stage, exceptions and branch mispredictions are caught and handled.

CISC: page 4
Complex Instruction Set Computer.

Completion: page 18
An arithmetic, logical, or floating-point instruction is said to be completed once the functional unit has finished the operation.

CAM: page 12
Content Addressable Memory. Memory which is addressed based upon its data content, as opposed to the address at which it is located.

COM: page 17
Instruction Commit stage. During this pipeline stage, in an out-of-order processor, instructions results are written back to the register file.

Context: page 22
An executable image. It holds the current state of a process. Physically it refers to the registers, program counter, and the stack pointer of a particular program.

Context Switch: page 30
The process of writing the register set of a program out to memory, and bringing in another program's set of registers.
Control Hazards: page 8
Due to the branch and jump instructions, the PC is not always incremented sequentially. Branch prediction is used to reduce the penalty from a control hazard.

Control Instruction: page 16
Instructions which will alter the sequential order of a program, either conditionally (branches), or unconditionally (jumps). This class of instructions also includes instructions which will jump to a subroutine, TRAP instructions, and return from exceptions [11].

CPU: page 1
Central Processing Unit. It is also referred to as a microprocessor.

Data Cache: page 10
Specialized cache that is used to hold data, and not instructions.

Data Hazards: page 7
In pipelining, this occurs when the output operand of an instruction is either an input or output operand of another instruction that follows it, and the second instruction must wait for the first instruction to finish before it can proceed.

Data Dependency: page 7
When two instructions are related through their input and output operands.

DIS: page 17
Instruction Dispatch Stage.

Dependency: page 7
Occurs when two instructions are related through their operands. For example, instruction one’s target register is the same as one of the read operands for instruction two.
Exception: page 28
This is an error that occurs in hardware, and a manner by which operating system services are accessed.

EX: page 6
Execution Stage.

Floating Point Instruction: page 16
Instructions that perform arithmetic and comparison instructions on floating-point data/registers.

Fine-Grained Multithreaded: page 19
A processor which holds multiple contexts in hardware, and switches between the contexts every couple of cycles.

Hazards: page 7
Problems that arise with the implementation of the pipelining technique. These are data, structural, and control hazards.

Instruction Cache: page 10
A specialized cache used to hold only the instructions of a program.

ID: page 6
Instruction Decode Stage.

IF: page 6
Instruction Fetch Stage.

Instruction Latency: page 4
The number of clock cycles required to complete an instruction.
ILP: page 16
Instruction Level Parallelism. The degree to which sequential instructions can be issued simultaneously. This is limited by both data and control hazards and the number of functional units available for execution.

IPC: page 39
Instructions Per Cycle.

IS: page 17
Instruction Issue Stage.

ISA: page 15
Instruction Set Architecture.

I-count Feedback: page 44
Technique by which the different threads are prioritized by counting the number of instructions in the static part of the processor pipeline. The thread with the lowest number of instructions gets the highest priority.

Kernel: page 20
This is the core of the operating system. It usually includes memory management, process management, and I/O interface.

L1 Cache: page 11
First level of cache. This cache is the closest to the CPU, and in many cases, directly on the chip itself. It has the fastest access time.

L2 Cache: page 11
A secondary level of cache. A cache that is larger L1 cache, used to help reduce the memory access time of an L1 cache miss, and the overall average memory access time.
**Logical Instruction:** page 16
An instruction such as AND, NOT, or XOR which perform bitwise logical operations on the operands.

**Memory Instruction:** page 16
An instruction such as load or store instruction, which will either read data from memory and save it in a register, or write data from registers to memory.

**MEM:** page 6
Memory Access Stage.

**Name Dependency:** page 7
Occurs when two instructions use the same register or memory, but there is no data being shared between the two instructions.

**Operand:** page 6
Data to perform a particular function on. In the case of a CPU, it can be either a register, or an address in memory.

**OS:** page 20
Operating System.

**Pipelining:** page 5
A technique in which an instruction is broken down into smaller operations. During every cycle, a new operation can begin. This will allow for a smaller clock period and higher throughput.

**Pipeline Stage:** page 6
The individual steps that are used to perform an operation in a pipelined processor. In the traditional five-stage pipeline, the stages are: Instruction Fetch, Instruction Decode, Execution, Memory Access, and Writeback.
PC: page 3
Program Counter.

RAS: page 47
Return Address Stack.

RAW: page 7
Read After Write hazard. Occurs when an instruction attempts to read a register before a previous instruction has written to it. Also known as a true data hazard.

Register File: page 6
A small, fast memory bank that runs at the speed of the processor.

Reservation Station: page 17
Hardware used to hold instructions waiting to be issued to a functional unit in the Tomasulo algorithm.

RISC: page 4
Reduced Instruction Set Computer.

Round Robin Selection: page 29
Simple selection mechanism by which the selection is done through an n-bit counter. There is no priority given in this selection mechanism.

RUU: page 68
Register Update Unit used in the Simple Scalar toolset.

SMT: page 33
Simultaneous Multithreading. It was first introduced in 1995 by Dean Tullsen at the University of Washington.
Spatial Locality: page 10
If an address of memory is accessed, then the addresses near that address will also be accessed relatively soon.

Speculative Execution: page 17
A method to help reduce the penalty from mispredicted branches. Instructions are executed, but not necessarily committed.

SP: page 19
Stack Pointer. A section of memory that is used to store return addresses, function parameters, and function results for procedure/function calls.

Superscalar: page 16
Refers to a microprocessor which can execute more than one instruction during any particular clock cycle.

Structural Hazards: page 8
Hazards that arise when there is a combination of instructions that can’t be satisfied at some point in time by a microprocessor’s resources.

Temporal Locality: page 10
If an address of memory is accessed, then that address will be accessed again relatively soon.

Thread: page 22
"Thread of execution.” A light-weight process.

TLP: page 33
Thread Level Parallelism. The degree to which two separate threads can be executed simultaneously.
Unified Cache: page 11
Cache arrangement where both the instructions and data are held together in the same cache.

WAR: page 7
Write After Read hazard. This type of hazard occurs when an instruction attempts to write to a register before a previous instruction has read from it.

WAW: page 7
Write After Write hazard. This type of hazard occurs when an instruction attempts to write to a register before a previous instruction has written to the register.

WB: page 6
Writeback Stage.
Acknowledgements

I would like to thank a few people, without whose help I couldn’t have completed this thesis. First, I would like to thank my graduate committee members, Dr. Muhammad Shaaban, Dr. Roy Czernikowski, and Dr. Ken Hsu for their guidance and helpful suggestions throughout this process. I would also like to thank Andre Botha, for his many hours of reading and editing this document. Finally, I would like to thank my parents who have been supportive throughout my entire education.
Trademarks

**Alpha** is a registered trademark of Compaq Corporation.

**HP, PA-RISC, PA-8000** are trademarks of Hewlett-Packard Company

**Intel, Pentium** are registered trademarks of Intel Corporation.

**MC68000** is a registered trademark of Motorola, INC.
Chapter 1

Introduction

Even with the advancements in computer architecture, there is a theoretical limit that is placed on a single-threaded architecture. The most aggressive single-threaded CPU architectures involve speculative and out-of-order execution. While this approach provides a performance improvement over the traditional five-stage pipeline architecture [11], it is inherently limited by the instruction level parallelism (ILP) that a particular program can offer.

The future of microprocessor architecture is looking at how single processors can be arranged to work with other processors, in either a multiprocessor or multicomputer configuration. One of the promising new architectures is the simultaneous multithreaded architecture. Simultaneous multithreaded (SMT) architectures feature multiple contexts in hardware, where a register file, a program counter, and a stack pointer constitute a context. Traditional multithreaded architectures also include multiple contexts, but each context gets all of the CPU resources for a certain period of time, and then the resources are switched to another context. SMT contexts share all of the functional units each clock cycle, as opposed to a single-chip multiprocessor where each of the integrated processors has its own functional units. Also, in an SMT architecture, a single control unit is used for the entire processor, as opposed to each of the processors in the single-chip multiprocessor having its own control unit.

SMT can take advantage of not only instruction level parallelism (ILP), but also thread level parallelism. Thread level parallelism (TLP) is the degree to which two or more separate threads can be executed simultaneously. Tullsen et. al. [27] show that under certain configurations, SMT
can achieve up to 5.4 instructions per cycle (IPC). One of SMT's advantages is its ability to hide the latency created by cache misses, longer floating-point operations, and branch mispredictions. If a particular thread misses on an instruction fetch, then a higher priority will be given to another, non-blocking thread. Unlike single-threaded machines, however, SMT doesn't have to perform a full context switch to allow another program to run. A full context switch includes saving the registers of the old process to memory, and bringing the registers of the new process into the processor. This is a very expensive task, on the order of hundreds of cycles.

In their introduction of the SMT design, Tullsen et al.[27] made the assertion that unlocking the SMT advantages would not require massive amounts of redesign from the traditional superscalar, out-of-order processors that are in the market today. A large portion of the work in creating a SMT architecture is the reproduction of (what are now considered) standard parts: program counters, instruction dispatch queues, reorder buffers, and exception handling units. One of the concerns in the design, implementation, and validation of the CPU is the design complexity of the control unit.

When the theory of a context switch is discussed, the operating system is the entity that controls which processes will be run. Many of the commercial and educational operating systems only deal with a single thread. The primary goal of this thesis is to explore the effectiveness of the SMT architecture in a real-world situation, where one has a large number of processes active at any time.

The goal of this project was to design, simulate, and analyze a model of a simultaneous multithreaded architecture. Chapters 2, 3, and 4 are theoretical backgrounds on computer architecture, operating systems, and the simultaneous multithreading architecture, respectively. Chapter 5 provides more detail about the architecture that is proposed in this paper. Chapter 6 explains the details of the simulator that was developed for this project. The results that were extracted from simulations are illustrated and explained in Chapter 7. Finally, Chapter 8 summarizes the project's findings, and suggests possible future work that could be performed.
Chapter 2

Theoretical Background of CPU Architecture

First and foremost, when describing the background of computer architecture, one must define the word architecture. The term was defined and brought to light by IBM in the 1960s, as they set out to define a family of computers that would run the same software. Up until that point, a new computer would mean that new software would have to be written to run on that machine. IBM came up with the following definition of architecture [11]:

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The purpose of this chapter is briefly discuss the history of the microprocessor, as it lends itself to why the direction of the processor is heading the way it is. Also, this chapter will discuss the basic parts of a modern microprocessor, as it is necessary to understand how all of those parts are used to understand their use in a simultaneous multithreaded machine.

2.1 Early Microprocessors

Early microprocessors would have a special microoperations to perform to complete a single instruction. There was a counter that pointed to the current address, so the processor would know where to fetch instructions from. This counter is referred to as a program counter, or PC. At first, the control unit was a large finite state machine. As more instructions were added, the state ma-
machine grew very large. The various instructions would take different numbers of clock cycles. These machines had specialized instructions to perform each operation, and were referred to as complex instruction set computers (CISC).

Instruction set architects analyzed the usage of instructions in a series of programs, and found that a relatively few instructions were used frequently. The highly specialized instructions, and generally longer latency, were not used as often. The latency of an instruction is the number of clock cycles it takes that instruction to finish. Additionally, in early microprocessors real estate on a chip was at a premium, and the specialized instructions were expensive in that regard. A new direction was taken to decrease the size of the instruction set, and optimize the machine to handle the reduced number of instructions. Microprocessors with the smaller number of simpler instructions are referred to as reduced instruction set computers (RISC). In addition, the number of addressing modes which a particular instruction set has is also simplified. Instead of a single instruction, such as

\[
\text{add} \ R1,(R2) + \quad // \text{add the value located at the address}
\]
\[
\quad // \text{in register 2 to the value in register}
\]
\[
\quad // 1, \text{and increment register 2 by one.}
\]

It would be replaced with the following series of instructions:

\[
\text{lw} \quad R3,(R2) \quad // \text{load the value located at the address}
\]
\[
\quad // \text{in register 2 to register 3}
\]
\[
\text{addi} \quad R2,#1 \quad // \text{increment R2 by } 1
\]
\[
\text{add} \quad R1,R3 \quad // \text{add the values in register 1 and}
\]
\[
\quad // \text{register 3, and store it in register 1}
\]

The tradeoff of compact code space was that the smaller number of instructions would be optimized to be faster, and therefore, the overall system performance in terms of execution time could be reduced.

Many metrics are examined to measure performance. One particular metric that is measure to show the relative speed of a particular machine is the cycles per instruction, or CPI for short. A single instruction takes multiple cycles. In order to perform a single operation, such as an add, many steps must be taken:
First, the processor must fetch the instruction from memory. That memory would either be a general purpose memory, or a more specialized instruction memory, as in the Harvard architecture [11].

The processor must find out what type of instruction it must perform, which is referred to as decoding the instruction.

The operation must then be performed.

Finally, the result must be written back to some kind of memory (register, DRAM, etc.).

If each of these step required a single cycle, then the CPI would 4. In addition, if this was a memory access instruction (load or store), the memory would either have to be read from or written to. If each step took 10 nanoseconds, this instruction would take 40 nanoseconds to complete. Instead of having a CPI of 4 or 5, the other option would be to make the clock cycle long enough, such that all operations could be performed in one clock cycle. However, with the clock cycle being increased, it may have a period of 50 nanoseconds to allow the operations to complete. Every instruction would take 50 nanoseconds. Hence, the CPI has been reduced to 1 for the longer clock cycle, but the total execution time has been increased by 10 nanoseconds per instruction (assuming the CPI of 4).

2.1.1 Pipelining

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>

Figure 2.1: Standard Five–Stage Pipeline

Computer designers came up with a new technique called pipelining, to help increase the total system throughput. Pipelining takes advantage of the common steps that are required to complete an instruction (fetch instruction from memory, decode instruction, etc.), and the fact that the logic for any particular step is only used for a fraction of the total time it takes to complete an
instruction. For example, if the longer clock cycle was used, and the logic for any particular step took 10 nanoseconds, that logic is only being used 20 percent of the time.

In pipelining, each of the different steps become pipeline stages. Registers are needed to hold the results of each stage, so that on the next clock those results can be passed on to the next stage. A five-stage pipeline is shown in Figure 2.1, and consists of the following stages:

- **Instruction Fetch (IF):** The memory containing the instruction is read from either main memory, or some secondary memory source, such as a cache (to be discussed later in this section).

- **Instruction Decode (ID):** The memory that was fetched is examined to determine what type of instruction it is, what its operands are, and help determine whether or not there will be any hazards with regards to this instruction and past/future instructions. It is also the stage during which the register file is read in order to get values for the operands. The operands are either register numbers, or memory addresses (for load/store instructions).

- **Execution (EX):** The actual operation is completed, unless it is a memory access instruction.

- **Memory Access (MEM):** If it is a memory access instruction, the memory (DRAM or cache) is read from or written to.

- **Writeback (WB):** The result of the operation is written back to the register file.

The key to a pipeline is that once instruction \( i \) finishes the fetch stage, on the next clock cycle, instruction \( i + 1 \) can be sent to the fetch stage, as instruction \( i \) is sent on to the instruction decode stage.

After \( n \) cycles, where \( n \) is the number of pipeline stages, the first instruction is finished. Each cycle following the \( n \)th cycle, an additional instruction completes. This, theoretically, brings the CPI down to one. The clock period can only be as small as the largest latency for any particular stage. Therefore, the latency for any individual instruction is not reduced with pipelining, and in fact, it may be slightly increased.
Unfortunately, a CPI of one is only a theoretical possibility. With the introduction of pipelining, additional problems arise. These problems are called hazards. There are three types of hazards: data, control, and structural hazards.

Data Hazards

Data hazards are caused by data dependencies. A name dependence occurs when two instructions use the same register or memory address, but there is no data being shared between the two instructions [11]. There are two types of name dependencies: antidependency and output dependency, or write after write (WAW). For example, if instruction $i$ writes its result to a register, and instruction $i + n$ reads from that register. Therefore, instruction $i + n$ is dependent upon instruction $i$. There are three types of data hazards:

- Read After Write (RAW) hazard: Occurs when an instruction attempts to read a register before a previous instruction has written to it. This is also known as a true data hazard.

- Write After Read (WAR) hazard: This type of hazard occurs when an instruction attempts to write to a register before a previous instruction has read from it.

- Write After Write (WAW) hazard: This type of hazard occurs when an instruction attempts to write to a register before a previous instruction has written to the register. This hazard cannot occur in a single-pipelined computer, but can occur in the superscalar architecture discussed in Section 2.3.

The easiest way to eliminate the hazards is to stall the pipeline until the hazard resolves itself (first read or write completes). Unfortunately, this may cause a penalty of up to $n^2$, where $n$ is the number of stages (as the decode stage and before must be delayed). Another method that is effective for RAW hazards, is data forwarding. While the result is not written back to the register file until the Writeback stage, it is available after the Execution stage. Therefore, if that result is also routed back to the input of the Execution stage, there is no need to stall the pipeline.
Control Hazards

Control hazards are due to the branch and jump instructions, the PC is not always incremented sequentially. The penalties caused by control hazards are reduced via branch prediction and speculative execution.

Structural Hazards

Structural hazards occur when a processor can't handle all of the possible instruction combinations. The primary cause of this would be if a functional unit with a longer latency (floating-point multiply/divide) that is not fully pipelined [11]. If the functional unit is not pipelined, and it takes more than one clock cycle, another instruction cannot be issued to it until it finishes the previous instruction. The way to prevent this type of structural hazard, is to design the functional units in a fully-pipelined manner.

Another way that structural hazards can exist is if hardware isn’t replicated in the proper fashion. Two examples of this are: memory interface and register file. If the register file only allows one access (read or write) per cycle, then there may be a conflict, as the writeback stage will be attempting to write to the register file, almost every cycle, and the decode stage reads from the register file, almost every cycle. Memory can cause the same problem. Memory, in most cases, does not run at the same speed, or with the same throughput as the processor. Therefore, if the memory interface is not pipelined, which would mean that any additional memory accesses must wait for the previous access to finish, there will be structural hazards, and the pipeline will have to be stalled. To solve the register file problem, a designer could implement multiple ports (both read and write) to the register file. The tradeoff of adding the multiple ports is the ability to run the clock at a high frequency. In order to compensate for the slower memory, a technique called cache is used in most, if not all, major commercial and educational processors.

2.1.2 Cache

Ever since microprocessors were invented, they have been speeding up, and executing instructions at faster and faster speeds. Unfortunately, memory (DRAM) has not been speeding up at the same
rate for either throughput or access latency. A key building block of instruction execution is the instruction fetch. Without the memory of the instruction, the processor is an idle piece of silicon. If main memory (DRAM) is accessed every cycle, and it has a latency of 20 processor cycles, then an instruction (or two depending on how many instructions are fetched) can only start every 20 cycles. Computer designers analyzed the code that was being run, and realized that a program by its nature is not completely sequential. For example, look at the following piece of C code.

```c
for ( i = 0; i < 100; i++ )
{
    a = (b + c);
    d = (b - c);
}
```

That would translate approximately (no machine specified) to the following assembly code:

```
.text
loop_i move R1,#0
lw R2,&A
lw R3,&D
lw R4,&B
lw R5,&C
loop
lw R6,(R2)
lw R7,(R3)
add R8,R6,R7
sw (R2), R8
sub R9,R6,R7
sw (R3), R9
addi R1,R1,#1
cmpi R1,#100
bne loop
loop_end
...
.data
int A
int B
int C
int D
```

The code, between `loop` and `loop_end`, will be executed 100 times. Ignoring all of the other memory accesses, and assuming a memory access latency of 20 cycles for a single access, that program time would be dominated by 220 x 100 cycles of memory access time for the instructions alone. What computer designers began to notice is that often the memory that is fetched for memory (and data) is accessed repeatedly. What if, instead of accessing DRAM every cycle, we
accessed a smaller piece of memory that would store these frequently used addresses? The principle of locality was introduced. There are two types of locality that shaped computer design, spatial and temporal.

Spatial locality states that if an address of memory is accessed, then the addresses near that address will also be accessed relatively soon. Spatial locality will occur after the integer $A$ is accessed, as $D$, $B$, and $C$ are also accessed in the instructions following $A$’s access. The addresses $A$, $B$, $C$, and $D$ could be held in a smaller set of memory, called the data cache. If the cache only has an access latency of two cycles for a read, then every time $A$, $B$, $C$, or $D$ is accessed 18 cycles are saved.

Temporal Locality states that if an address of memory is accessed, then that address will be accessed again relatively soon. Temporal locality is realized with the repeated access (100 times in this case) of the code between loop and loop_end. The code between loop and loop_end could be held in a small subset of memory called instruction cache. If that cache only has an access time of two cycles, then 18 cycles are saved for every instruction between loop and loop_end.

Cache Misses

A cache cannot hold all of the contents of main memory, so misses occur. There are three types of cache misses: compulsory, conflict, and capacity cache misses. In the example above, the first time that the first address (move #0,R1) was requested, the cache missed. That address hadn’t been in the locality of the cache. That type of cache miss is called a compulsory cache miss. There is no true way to fight those types of cache misses. When two addresses are mapped to the same block, and one is replaced by the other, and later has to be retrieved again (because of its replacement), a conflict cache miss occurs. One way to reduce the number of these is to increase the associativity of the cache. The associativity refers to how many different blocks are available to one mapping. In a set–associative, or fully–associative cache, once all of the entries in a particular set (or in the entire cache, as the case is for a fully–associative cache) are used, the following cache accesses that miss are going to be considered capacity cache misses because there is not enough space in the cache to handle the data (or instruction) of a program [11]. To decrease the number of capacity misses, the
size of the cache should be increased.

A lot of research has gone into cache development. With it, the processor has a set of memory that is running at, or near the processor's clock speed. However, in order to get that speed, the cache must tradeoff the larger size that main memory is afforded. A diagram relating the sizes of the various memory types and their access latencies and sizes is shown in Figure 2.2.

Computer designers experimented further, and came up with the idea that if a very small cache set could run at nearly processor speed, could the memory penalty be reduced even more if a larger, slower second level of cache was added. This is the L2 cache. The first level of cache, L1, was the smallest set of cache, and was closest to the CPU. When the processor makes a memory request, whether it is for an instruction fetch, or for reading or writing data, its memory interface is routed first to the L1 cache. The exception to this is during write requests in the write-through cache arrangement to be discussed in Section 2.1.2. If the L1 cache doesn't hold the address, the request is then sent on to the L2 cache. If the L2 cache doesn't have the address, then main memory receives a request from the processor.

If a cache has both instruction and data references in it, it is referred to as a unified cache. In many cases, the L1 cache will have separate instruction and data sections, and the L2 cache will be unified. Keeping the instruction memory (.text section in a C program) and the data memory in separate caches provides the following benefits:

- Two accesses can occur simultaneously without having to worry about adding additional ports to the cache.
- The accesses and replacements will not interfere with each other, and inadvertently remove often used instructions/data.
Cache has become an integral part of microprocessors. It also has been receiving greater, and greater portions of the total silicon area of the processor. The realization is that the performance of the processor is directly related to the manner in which the difference in access latency between the processor and memory is reduced.

**Cache Arrangements**

How does the CPU know whether or not an address is located in the cache? Cache is a type of *content addressable memory* (CAM). Part of the address is used to decide whether or not a particular block is going to be used. Part of that address is also written when the cache block is brought in from memory. Therefore, the data is actually checked to help decode whether or not this is the block we want, and this is the definition of a CAM. There are mapping functions in HW to convert the address to a cache block. The simplest conversion is called the *direct mapped cache*. The conversion is a straight modulo operation:

\[
\text{Cache Set} = \text{Address} \mod \text{Number of Cache Blocks}
\]
That method is the easiest to implement in hardware, but what happens if two addresses that are frequently used map to the same block. At that point, the cache’s benefits will be negated. Another method for mapping addresses to a cache block is the fully-associative cache arrangement. In the fully-associative arrangement, any address can be placed in any cache block. While this method, conceptually, would be advantageous, implementing the algorithm in hardware is costly. The intermediate result of these two extremes is the set-associative cache arrangement.

Cache Set = Address mod (Number of Cache Sets × Number of Blocks per Set)

If the set-associative cache arrangement, an address is mapped to one of \( n \) cache blocks, where \( n \) is the number of cache blocks in the cache set. The block which the address is selected for is dependent on the block replacement algorithm, which will be discussed later in Section 2.1.2.

The components of a cache block are the tag, index, valid bit, dirty bit, data field, and the block offset field. The tag is the part of the address used to determine whether there is a cache hit or not. The index is used to select the correct cache block from all of the cache blocks in the cache \([11]\). The valid bit is an indicator of whether or not the block is being used. If this bit is set, then the address and data in the cache block are valid (hence the name). The dirty bit reflects on whether or not the address has been written to since it has been in the cache, and the lower levels of memory (L2 cache, DRAM, etc.) have not been updated. This bit is not necessary in a write through cache (to be discussed in Section 2.1.2), as all levels of memory are updated during any write to memory. The data field is the value at the address stored in the cache block. The block offset is used in a set-associative cache to select the correct block in a particular set.

Cache Writes

There are two highly used memory write handling cache techniques called writeback caches or write through caches. The tradeoffs that the two techniques weigh are memory bandwidth usage, and memory consistency.

The writeback cache will change the first level of cache, and then only write the result back to lower levels of memory (L2 cache, main memory, etc.) when that particular block is replaced. That way, if the block is changed multiple times before it is replaced, it will only use the memory
bandwidth on one occasion. This is not a major problem in a single processor system, as the CPU will be the only user of the memory. However, in a multiple processor system, where more than one CPU is accessing the same memory, there could be a problem. Consider a two-CPU, one memory system, with CPU one holding a piece of data. CPU one writes to that address and the cache is updated, but the main memory isn't updated. Now, CPU two attempts to read that address from memory, expecting the changed data, but instead gets the old value. That is a cache consistency problem. The design of the solutions is beyond the scope of this thesis, with the exception of the write through cache.

The write through cache takes the opposite approach of the writeback cache. When a cache block is written, it writes the value through to each level of the memory hierarchy. This consumes a larger percentage of the memory bandwidth, but the cache consistency is guaranteed by design. There are also techniques, such as write buffers which allows the memory bandwidth problem to be reduced slightly. The conservation of memory bandwidth is a recurring theme in microprocessor design, and is increasingly important in SMT design.

Cache Block Replacement Algorithms

If a particular cache block that has been selected as a target for writing is full, then the block replacement algorithm determines how to handle the situation. The simplest cache block replacement algorithm is the direct-mapped replacement. It is the simplest because if an address is mapped to a particular block, whether or not the block is being used, it must be replaced. During replacement in a writeback scheme, if the block is valid (valid bit is set) and it has been modified (dirty bit is set), then the old block must be written back to the next level of cache (L2 or main memory).

Another fairly easy technique to determine which block (in a set-associative cache) should be replaced is the round robin approach. In the round-robin approach, a simple counter keeps track of which block in a set should be replaced. Consider a 4-way set-associative cache. If set $n$ is selected, then one of the 4 cache blocks in set $n$ must be replaced (assuming they are all in use). For a 4-way set-associative cache, the counter must be two bits ($2^2 = 4$ different blocks). The counter will be incremented (or decremented) by one every time a replacement occurs. While this
is easy to implement, it doesn't take into consideration whether or not the replaced block was used recently.

A more complicated technique is the least recently used method. This will take into consideration which blocks have been used recently. Unfortunately, the hardware implementation is more complex, as each of the blocks must have their own counter which will be reset if the block is used, and incremented if the block isn't used on an access. The block with the highest count will be replaced.

There is a technique, which is simple, but also takes into consideration the use of the blocks, and it is called the not recently used replacement algorithm. In this replacement algorithm, each block only needs a single bit, where if it is set, the block was not the last used. This way, the cache avoids replacing the last used block. A more complete reference summarizing cache arrangements and replacement algorithms is [1].

### 2.2 Instruction Set Architecture

The instruction set of a processor will define how a programmer is going to interface with the CPU. In modern processors, software is usually written in a high level language, such as C. The compiler hides the entire assembly language from the programmer. The programmer does not even see registers, so the architecture of the computer is not the concern of the programmer. To the computer designer, however, the instruction set is as important, if not more important, than the pipeline of the processor. There are two schools of thought for the instructions sets, small and general, or large and specialized.

The small and general instruction set refers to the RISC train of thought, where the smaller number of instructions will be optimized for speed. One of the early RISC instruction sets was designed by Patterson and Hennessey [11], and was called the DLX instruction set. It was primarily used as an educational tool, though there have been MIPS microprocessors which use the DLX instruction set as its native language. The DLX instruction set not only had a limited instruction operation repertoire, but also had only a few addressing modes.

There are six types of instructions that are of importance: arithmetic, logical, floating-point,
memory, and control instructions. Arithmetic instructions are those which will perform functions like add, subtract, multiply, or divide on fixed point (integer) registers/data. Floating-point instructions perform operations similar operations on floating-point registers/data. Logical instructions perform bitwise operations (AND, OR, etc.) on fixed point (integer) registers/data. Memory instructions are instructions which will interact with memory, either copying the contents of memory locations into registers, or copying registers into memory locations. Control instructions are instructions which will alter the sequential nature of a program, either conditionally or unconditionally.

Control instructions also include some special instructions (TRAP, RTE, JAL, etc.). These instructions allow the programmer to use subroutine calls (JAL) and return from interrupts (RTE). The TRAP instruction, or SYSCALL instruction in some architectures, allows control of a program to be transferred to operating system (kernel) code. The instructions TRAP (SYSCALL) and RTE are the hooks that allow the operating system to work with the processor, though the instructions are not the limits of what the OS designer needs.

2.3 Superscalar Microprocessors

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Dispatch</th>
<th>Issue</th>
<th>Exec</th>
<th>Writeback</th>
<th>Commit</th>
</tr>
</thead>
</table>

Branch Misprediction Penalty (5 cycles)

Figure 2.3: Typical Superscalar Pipeline [4]

By examining the instruction dependencies, it was apparent that performance could be improved with more unrelated instructions being sent through the processor at the same time. This is referred to as instruction level parallelism (ILP). Additional resources, such as pipeline stages and latches are required to receive the benefit of this improvement. A diagram showing a generic pipeline for a superscalar, out-of-order microprocessor is shown in Figure 2.3. Instead of running a single instruction through a pipeline stage during any particular cycle, two to as many as eight instructions in some processors could be worked on simultaneously. This allows a processor's CPI to drop below
one. The implications of this approach are higher performance, at the expense of more complex control circuitry in the processor. Also, the performance of a processor is limited to the degree of ILP which occurs in a particular program.

2.4 Speculative Execution

Another important step in computer architecture, was the development of the out-of-order processor. Control instructions, described in Section 2.2, cause control hazards, which were described in Section 2.1.1. A program which has been proceeding sequentially in nature is having its fetching disrupted because the program counter may or may not be changing after a branch instruction (conditional change of the PC). Therefore, if the processor were to wait until it knew whether or not to alter the PC, two or three cycles (depending on where the condition was determined) would be wasted each time a conditional branch instruction occurred.

Computer designers decided that the wasted time could be reduced, or eliminated altogether, if the computer decided to "guess" which path to take. This is referred to as branch prediction. In branch prediction, the processor decides whether to predict that a branch is "taken" (the PC is altered in a non-sequential manner), or the processor predicts that a branch is "not taken" (the PC is changed sequentially). A branch misprediction occurs when the processor makes the wrong guess. The hardware cache used to help predict branches is referred to as a branch target buffer, or BTB. An entry in the BTB holds both the current PC and the predicted PC values.

Branch prediction methods were devised to help improve the success of a "guess". If a processor incorrectly guesses the path to take, a heavy penalty is paid, as the pipeline is flushed of incorrect instructions. With the introduction of both superscalar processor techniques (described in Section 2.3) and the speculative processor techniques, the traditional pipeline was altered to include new stages, such as Instruction Dispatch (DIS), Instruction Issue (IS), and the Commit (COM) stages [11].

The dispatch stage is used to select instructions from the decoded instruction buffer and send them to reservation stations. The Tomasulo approach, invented by Robert Tomasulo, and used in the IBM 360/91 [11] uses reservation stations to hold instructions which are waiting to be sent.
to a functional unit. During the issue stage, instructions waiting in the reservation stations are sent to available functional units. Functional units are hardware that perform the actual functions, such as add, subtract, NOT, or floating-point operations. In the out-of-order processors, the writeback stages is no longer the final pipeline stage. The writeback stage is used to broadcast the functional unit results back over the common data bus (CDB) to the reservation stations and the inputs of the functional units. This is the manner in which data forwarding occurs in the more complicated out-of-order, speculative processors. Figure 5.2 shows how the SMT processor handles the common data bus. This also shows a good picture of how the general Tomasulo approach works with reservation stations and the CDB.

The commit stage is the final stage in the pipeline. If no exceptions occurred, it is at this point that instruction results are written back to the register file. Instructions which have their results written back to the register file, are said to be retired, or committed. Instruction completion in the out-of-order processors occurs for arithmetic and logical functions after the functional unit has completed.

The pipeline is now an eight, or nine stage pipeline. In some processors, such as Intel’s Pentium Pro, there are thirteen stages. Intel’s x86 family is a CISC architecture, but the Pentium Pro internally transforms the CISC instructions into RISC micro-instructions. Longer pipelines imply heavier penalties for branch misprediction.

2.5 Multithreaded Microprocessors

In a single processor system, the operating system decides which process (program for simplicity) is going to run every interval. The length of an interval is operating system dependent, but is a predetermined amount of time (operating systems will be discussed later in Chapter 3). At that point, the scheduling function of the operating system decides whether to allow this process to continue running, or swap it out and bring in another process. This is referred to as a context switch. The context switch is a very expensive operation, as there are many memory access (read/write) operations involved. In order to reduce the time spent on context switches and avoid the penalties of longer latency instructions and branch mispredictions, computer designers developed the
multithreaded processor.

In traditional, or fine-grained, multithreaded microprocessors, multiple sets of registers are utilized. These registers include the generic integer and floating-point registers, and also all required specialized registers, such as the PC and stack pointer (SP). Other registers that could be included in the context would be the text base register, text size register, floating-point, and integer condition code registers.

The text base register indicates where the processes code (text) begins in memory. The text size register indicates the size of the process' code and data. Each memory access must be checked by the processor to protect memory of other processes [11]. The protection is enforced by making sure that the address confirms to the following inequality:

\[
\text{TextBase} \leq \text{Address} \leq \text{TextBase} + \text{TextSize}
\]

If the inequality isn't conformed to, then an exception will occur, and the operating system will handle the errant memory access.

Fine-grained multithreaded processors hold a number of program's contexts in memory, and select one to run for \( x \) cycles. Such a multithreaded processor has been designed and implemented in [9]. A context switch that is internal to the processor only requires the processor to begin executing instructions from the new context, as opposed to writing register values out to memory and reading new register values in from memory. This allows the processor to better tolerate longer-latency instructions and latencies created by slow memory interfaces.

A significant point of operating deficiency in both the traditional multithreaded and SMT processors, is the performance of the caches [21, 4]. Instead of attempting to map a single set of data into a smaller set of memory, two, four, or eight memory address spaces are being mapped to this cache. The sizing and arrangement of caches in multiple-context processors is a very important, and a non-trivial, step in the architecture. It will be further discussed for the SMT processor in Chapter 4.
Chapter 3

Theoretical Background of Operating System Issues

A microprocessor is nothing more than a fancy piece of molded sand if it isn't supplied with any useful instructions to run. At first, each time a new processor was designed, all of the software that was run on it had to be developed from scratch. IBM came up with the idea of creating a family of computers that would share the same lowest-level commands, the IBM 360 series. Even a single program stored on a hard disk by itself is pretty useless. How is the processor supposed to get the information from the hard drive and into memory for it to run? Suppose there is a piece of software that can abstract away the fact that the program is not in memory. Then all the processor has to be told is when that memory is valid, and it can begin to execute the program from memory. Part of the operating system's job is to take care of details like that. An OS also provides an interface, similar to the instruction set of the processor. This set of functions and procedures are called the operating system's system calls.

The operating system is a program, albeit a complex one, that performs the various "housekeeping" tasks like directing the processor to run a particular program at a particular time (this is called scheduling). The kernel is the main code, the heart, of the operating system. It usually deals with process and memory management, and interacts with device drivers. While device drivers are a crucial part of the entire system, they are beyond the scope of this paper. The OS concepts in this paper will deal with the kernel services of process and memory management, and system calls. In many operating systems, a layer of abstraction between the hardware and the processor is set
up. Therefore, in the example above, when a processor needs to read from an external device such as a disk drive, it doesn’t need to know where on the drive the data is located, or what kind of drive it is. All it has to do is provide an address, and how much data is needed, and then call a function that the operating system, in most cases, provides to read from the device.

In order for an operating system to run a program it should have some universal way of dealing with, and managing different programs. An OS is considered multitasking when it provides the illusion of running more than one program “at the same time.” In actuality, the OS fakes the “at the same time” by sharing the processor’s time amongst the programs.

“Computer systems usually do have some concurrent capabilities, but the most visible form of concurrency, multiple independent programs executing simultaneously, is a grand illusion.” [2]

The following items, according to Patterson and Hennessey [11], are necessary for the CPU to provide to the OS developer:

- At least two modes of operation: kernel (or supervisor) and user.

- Portions of the CPU state that the user can’t write to. Examples would be user/supervisor bit, exception enable/disable bit, and other memory protection bits.

- Mechanisms that allow a CPU to enter and exit supervisor mode. In the MIPS architecture, the TRAP instruction allows a process to go into kernel mode. SYSCALL is another name for the TRAP instruction on other machines. The RFE instruction will return from the exception, and thus go back into user mode.

The OS portion that deals with scheduling and memory management can also be considered as a process management subsystem. Process management can be thought of as another layer on top of the operating system, with hooks into the lower layers. In order to understand the process management, one must understand what constitutes a process.
3.1 Process Introduction

In Linux, a process is defined as “a scheduling entity and the only thing unique to a process is its current execution context” [19]. A thread is “a scheduling entity that shares its resources with other threads” [19]. A process can be considered to be a running program, or as described by [7] a task or thread of execution.” A context is a concept that refers to the register set, PC, and SP of a running process. A process is created by the fork [7] syscall, or the create syscall, depending on the OS. A system call is a function which performs a specialized task, such as reading from an external device, or creating a new process. System calls will be discussed later. In Xinu [2], a process, or job, is described as an isolated computation. A task refers to one of a set of cooperating computations. The system call create must instantiate a new process that looks like it had just gone to sleep [2]. Creating a process includes allocating a space in the process table, and creating the process’ stack space. The process table is a data structure in memory that contains the information about all of the processes that are active in the computer.

3.1.1 Process Memory Management

A process's memory space may consist of multiple sections, depending on the OS implementation. In general, there is a text and data segments. The data segment may be broken up into a number of different sub–segments. For example, there is a binary file format, known as the COFF format, which is broken into six sections: .text, .rdata, .data, .sdata, .bss, and .sbss segments. The .rdata, .data and .sdata segments are initialized data segments, while the .bss and .sbss segments are uninitialized data segments. In BSD UNIX [7], the uninitialized data segments are created using zero–filled memory (memory blocks that are all 0’s). In Xinu [2], a C program is laid out as in Figure 3.1.

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>FREE SPACE</th>
<th>---stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_cdata</td>
<td>_end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: C Program Memory Layout in Xinu OS [2]
In addition to the memory that a program has been allocated at compilation time, in some systems it may request more while running the program. In BSD Unix, there is an additional section of memory that may or may not be used by a process. This section is called the *heap*. The heap is used to dynamically allocate memory to a program at runtime. A diagram of a program that includes the heap segment is shown in Figure 3.2. A second task of the operating system is memory management. In general, memory management refers to handling processes' memory allocation and deallocation. Memory allocation involves both the allocation of a processes' code, data, and stack.

The memory manager in Xinu keeps track of all of the free space using a singly-linked list. It allocates memory by searching through the list from the beginning until it reaches a block of memory that is large enough to fulfill the memory request.

```
0  _etext  _cdata
 text data bss heap--> FREE SPACE <--stack
```

Figure 3.2: C Program Memory Layout in BSD Unix OS [7]

Each process has its own private address space. That space is divided up into a number of sections, depending on the OS and execution file format. In BSD UNIX, an executable (program) is broken up into three segments: *text*, *data*, and *stack* [7]. The *text* is the code that is going to be executed, it is a read-only area. The *data* segment is broken up into two sections: initialized and uninitialized data. The *.data* segment holds the initialized data, and the *.bss* segment holds the uninitialized data. The *stack* is a section of memory used to save return addresses and pass parameters to and from subroutines. Both the *data* and *stack* segments are readable and writeable.

**Memory Model**

The way in which the OS handles interactions between processes is an important part of any modern OS. A shared memory model occurs when multiple copies of a particular program share the same code, but have their own distinct data space. By breaking a program up into separate sections, it
allows an OS to run multiple copies of the executable using the same text segment. The extreme case is the multiprogrammed model, where none of the programs are associated with each other. In the multiprogrammed model, more than one execution of a program would result in two sets of addresses being generated.

While the memory model to be used by an OS is important, it is beyond the scope of this thesis, as the programs/processes that are examined here will have their own code and data space.

3.2 Process Creation

BSD UNIX [7] creates new processes through the *fork* system call. Xinu [2] uses the *create* system call. A fork (process creation) call has three functions:

1. Find an empty slot in the process table.

2. Duplicating the parent context (with the exception of the process ID). This includes setting up the user structure [7] and the virtual memory resources.

3. Scheduling the child process to run.

The difference between normal procedure calls and process creation is the following: a procedure call does not return until the called procedure completes. The system calls that create a process, such as fork or create return to the code where they were called after starting the new process. This then allows the parent process to continue execution, with the byproduct of a new process that will execute concurrently [2].

3.3 Process Termination

A process can be terminated in one of two fashions:

- Explicitly calling a function such as *kill* [2] or *exit* [7], or

- Through the use of a *signal*. 
The termination of a process includes reporting back the exit status to its parent process. In UNIX, if the parent process has been terminated before the child process, the child becomes a child of process number 1. In a multiuser environment, process 1 is in charge of spawning off the initial processes for a user login [6].

3.4 Process States

![Process State Diagram](image)

Figure 3.3: Process State Diagram

The process state diagram is shown in Figure 3.3. It shows the various transitions that can be made from the creation to the end of the processes life cycle. The state transitions are described in Table 3.4.

The different states a process can be in at any particular time are:

**IDLE** This is the transition state that the process goes into as it is being created. At this point, its various segments (code, data, and stack) are being initialized.
WAIT  This state occurs when a process is ready to run (usually it involves the process being put into a run queue), but it has not been scheduled to run. In order to be scheduled to run a context switch must occur, and it must be in the top $n$ threads in terms of priority (where $n$ is the number of threads that the CPU supports simultaneously). For a single-context machine, $n$ is one.

SLEEP  This state occurs when the process is waiting on a data value, a signal, or some event to occur. The process will be swapped out of the CPU (if there are other threads waiting to be run), and will be put into a sleep queue.

WAIT_HW  Now that there is more than one thread available in the CPU itself, an additional state will be possible. This state may actually be transparent to the OS depending on how it is defined. If a thread incurs an I-cache miss, then there is a possibility that it could be marked as such in the operating system's process table. This may not be necessary if the I-cache miss delay is insignificant.

ZOMBIE  This state marks the beginning of the end of a process. Either the process has called a system call like exit on itself, which would suggest that it is finished, or it has been killed by another process by means of a signal. At this point, the kernel must mark its entry in the process table as invalid, and reclaim the memory given to it. Also, if the process had been holding any system resources, it should release those at this point.
<table>
<thead>
<tr>
<th>Transition</th>
<th>Handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Creation</td>
<td>OS</td>
<td>Process is created via a <em>fork</em> or <em>create</em> system call (OS dependent)</td>
</tr>
<tr>
<td>2. Initialization</td>
<td>OS</td>
<td>Processes’ memory space is initialized and is the process has had its priority calculated. It is now waiting to be run.</td>
</tr>
<tr>
<td>3. Woken Up</td>
<td>OS</td>
<td>The event the process has been waiting for has occurred. It is waiting to be run.</td>
</tr>
<tr>
<td>4. Killed</td>
<td>OS</td>
<td>Someone has killed a waiting process.</td>
</tr>
<tr>
<td>5. Selected to run</td>
<td>OS</td>
<td>The process has been scheduled into one of the top $n$ priority slots (where $n$ is the number of hardware threads the CPU supports).</td>
</tr>
<tr>
<td>6. Context switch</td>
<td>CPU/OS</td>
<td>The process has moved out of the top $n$ priority slots, and its state has been saved to main memory (or cache).</td>
</tr>
<tr>
<td>7. On-chip</td>
<td>OS/CPU</td>
<td>This state transition can’t happen, as if there are less than $n$ threads in the CPU, then the next thread will be selected to run, not wait in the CPU.</td>
</tr>
<tr>
<td>8. Context switch</td>
<td>CPU/OS</td>
<td>The waiting process has moved out of the top $n$ priority slots, and its state has been saved to main memory (or cache).</td>
</tr>
<tr>
<td>9. Signaled</td>
<td>CPU/OS</td>
<td>The signal, or memory from an I-cache miss that the process was waiting for has arrived.</td>
</tr>
<tr>
<td>10. Killed</td>
<td>OS</td>
<td>Someone has killed a sleeping process.</td>
</tr>
<tr>
<td>11. Died/Killed</td>
<td>OS</td>
<td>Either the process has been killed by another process, or it has <em>died</em>.</td>
</tr>
<tr>
<td>12. Killed</td>
<td>OS</td>
<td>Someone has killed a waiting, CPU-cached process.</td>
</tr>
<tr>
<td>13. Sleep</td>
<td>OS</td>
<td>The process has put itself to sleep, via a system call (such as <em>sleep</em>) or it is waiting on a signal, or for data.</td>
</tr>
<tr>
<td>14. Hardware wait</td>
<td>CPU/OS</td>
<td>The process is waiting for an I-cache miss, or a signal, and there are less than $n$ threads ready to run.</td>
</tr>
</tbody>
</table>

Table 3.1: Process State Transition Description
3.5 System Calls

The operating system, like the processor, provides a set of “instructions” that the programmer can use to make their life easier. The functions that the OS provides are called system calls. They are accessed via a software TRAP, or SYSCALL instruction provided to the OS by the CPU. With system calls, the OS creates a “virtual machine” similar to the physical machine created by the CPU. The programmer can use these functions to handle common tasks, such as reading from, or writing to, a file.

3.5.1 Entering the Kernel

Once the kernel begins to create and run user processes, how does a program get access to the kernel, and its resources? How is the kernel used and accessed to perform its various functions? There are various methods for entering the kernel:

**Timer Interrupt** In order to share the processor between the processes that are active, there must be some way to indicate it’s time to switch the running processes. A synchronous event occurs at the same time in a program, whereas an asynchronous event does not occur at the same time. Usually, asynchronous events are caused by an external source, as in this case, where it is the timer that causes the event.

**Hardware Exceptions** When dealing with arithmetic calculations, there are various errors that can arise, as there is a limit to the size of the numbers that can be represented. When these limits are exceeded, the OS must step in and determine what steps to take. The OS organizes jumps to predetermined exception handling routines are resident in memory. The CPU has exception-handling hardware to change the PC to the appropriate address when an exception occurs. At the end of the exception-handling subroutine, an instruction such as RFE (return from exception), will cause the CPU to return to the program which caused the exception (usually one address after the violating instruction).

**Software Trap** At some point in an user-program, the program might need to gain explicit access to the operating system’s functions, e.g. when requesting additional memory at runtime. In
order to do this, the program will use a special instruction, such as SYSCALL, or TRAP. The argument of this instruction would be the number of the system call when it encounters this instruction, the processor will translate this number into an address to jump to in order to handle the trap. As with exception handlers, the RFE instruction will return the processor to the address after the software trap.

**Process Scheduling**

Interrupts, as discussed in Section 3.5, are vital to the operating system, especially timer interrupts. The process scheduler needs to be awakened periodically in order to determine if a different process should be run. The timer interrupt is used for this purpose. It causes an interrupt every $n$ microseconds at which point, the CPU processes it with software handling routines that the OS provides. Many times these software-handling routines, or at least part of them, must be written in assembly language as they involve saving and restoring registers.

When the timer goes off, the scheduling process becomes the active process in the CPU. At this point processes are scheduled using an equation which takes into account many processes there are, how much time each process has been allotted up to this point, and the priority of the processes. In Xinu, the scheduler follows the following algorithm:

At any time, the highest priority process eligible for CPU service is executing. Among processes with equal priority scheduling is round–robin [2].

Round–robin means that the scheduler selects tasks in the same order, with equal priority. Information about the system’s processes is kept in a structure called a process table. It keeps information on the process like the state of the process, which will be discussed in Section 3.4. Also, it keeps track of the priority, the initial PC for the code, the stack base and limit, the text base and limit. In addition, it may track: file handles that the process had open, which user and group owned the process, current working directory, and which memory space the program had access to and how it was laid out [19].

The next question would be, how is the process’ priority determined? The priority determination mechanism will indicate which type of processes are favored. In UNIX, processes are categorized
into two types of processes: interactive and non–interactive. Interactive processes are those which
require a lot of input and output, while the non–interactive typically involve heavy computation.
Interactive processes are given highest priority as once the input or output is ready, the process
doesn’t typically require a lot of CPU time. The non–interactive processes usually use up all of
their *time slice*, which is a specified portion of CPU time. The amount of time slice used up is
also used to determine priority, as if a process has used up a lot of its time slice, it has had the
processor for a longer period of time than one that hasn’t used much of its time slice. When the
priority of a process which is running becomes lower than another process, a context switch must
occur. Context switches are discussed in Section 3.6.

### 3.6 Context Switching

This is a machine–dependent operation. The core of the context switch is the saving and restoring
of the context’s registers. This is one part of the kernel that must be written in assembly language,
as higher level languages, such as C, cannot operate directly on the machine’s registers. A sample
set of code is included below. It is from Xinu [2], and is written for the PDP 11/2. In an SMT
system, the CPU takes on the added responsibility of creating a "soft" context switch during every
clock cycle when it selects the different threads to execute. However, since the CPU "caches" some
of the contexts in hardware, there is no reason to swap a context out to memory every time a
new process has the highest priority. This uses the principle of locality (similar to that used in
memory hierarchy), and applies it to hardware contexts (registers, program counters, and stack
return addresses).

```c
/* csv.s - csv, cret */

/* C register save: upon entry here, procedure A has called B,
and B has called csv to save registers. r5 contains return
address in B. The stack has old r5, return addr. in A, and
arguments on it. C return: cret (below) is used to restore
regs when the called proc. finally exits. */

.globl csv, cret
/*-------------------------------------------------------------*/
/* csv -- C register save routine */
```

30
There are two types of context switches, voluntary and involuntary [7]. A voluntary context switch will occur when a process blocks as it is waiting on a signal, data, or some other resource. This can occur through a couple of different systems calls such as the `sleep` or `wait` signals.

### 3.7 Single-Threaded OS versus SMT OS

In single-threaded OSes much of the actual system execution is done as a result of a software TRAP. For example, if the process wants to make a system call to `sleep`, then there is (in the MIPS instruction set) an instruction called `syscall`.

In an SMT system, the CPU takes on the added responsibility of context switching during every clock cycle. However, since the CPU "caches" some of the contexts in hardware, there is no reason to swap a context out to memory every time a new process has the highest priority. This uses the principle of locality (similar to the memory hierarchy), and applies it to hardware contexts (registers, program counters, and stack return addresses).
Single-Threaded OS

OS chooses highest priority process. OS swaps out the old program and swaps in the new context every time the priority changes amongst the processes.

The kernel will do its scheduling on a process-by-process basis.

In a single-threaded system, the kernel only has to know if a particular process (context) is active.

---

SMT OS

OS chooses top $x$ priority process (where $x$ is the number of threads). OS only swaps out a process when a new process gets into the top $x$ priority.

Once the kernel places the top $x$ processes/threads into the CPU, the actual instruction-by-instruction scheduling is done by the CPU's hardware instruction scheduler.

In a multi-threaded system, the kernel will keep track of which hardware context a particular process is associated with this can be done via an extra field in the process table.

<table>
<thead>
<tr>
<th><strong>Single-Threaded OS</strong></th>
<th><strong>SMT OS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS chooses highest priority process. OS swaps out the old program and swaps in the new context every time the priority changes amongst the processes.</td>
<td>OS chooses top $x$ priority process (where $x$ is the number of threads). OS only swaps out a process when a new process gets into the top $x$ priority.</td>
</tr>
<tr>
<td>The kernel will do its scheduling on a process-by-process basis.</td>
<td>Once the kernel places the top $x$ processes/threads into the CPU, the actual instruction-by-instruction scheduling is done by the CPU's hardware instruction scheduler.</td>
</tr>
<tr>
<td>In a single-threaded system, the kernel only has to know if a particular process (context) is active.</td>
<td>In a multi-threaded system, the kernel will keep track of which hardware context a particular process is associated with this can be done via an extra field in the process table.</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison Between Single-threaded OS and SMT OS
Chapter 4

Theoretical Background of Simultaneous Multithreading

The simultaneous multithreading (SMT) architecture is a modification of the traditional interleaved (or fine-grained) multithreaded processor. The interleaved multithreaded microprocessor will run all threads at the same time, but will only issue to the functional units from one thread during a cycle. The architecture was originally introduced by Tullsen et. al. [3] from the Department of Computer Science at the University of Washington. In one paper, the comparison was made between different types of simultaneous issuing, which ranged from a single thread issuing per cycle to all threads issuing in every cycle. This paper will deal with the full simultaneous issue, where every thread can potentially issue an instruction on every cycle. The architecture attempts to surpass the limits of performance placed on the speculative, out-of-order, superscalar, and the traditional multithreaded processors that exist today, by converting "TLP into IIIP" [14]. TLP is thread-level parallelism, a measurement of how well different threads can be run in parallel. In addition to the benefits that the processor gives, Tullsen et. al. [4] claim that major changes are not required to convert an existing superscalar processor into a SMT processor. Some of the resources, such as the PC, SP, and register file must be replicated. The entire idea of SMT is for the processor to take advantage of the resources that are at its disposal in the most efficient manner. This occurs by allowing a combination of threads to issue instructions to the functional units.
One of the goals of the original SMT design [4], was to avoid imposing a heavy penalty upon a single thread running through the system. The added complexity created when SMT is enabled may cause penalties, such as additional pipeline stages to handle the increased number of registers. A realistic SMT pipeline proposed by the Tullsen et. al. [4] is shown in Figure 4.1. The diagram also indicates the branch misprediction penalty, which is also increased due to the increase in the number of pipeline stages. Consider the benefits of pipelining, as a single instruction may have its latency increased by a small amount, but instead of having one instruction done at 40 nanoseconds, the process completes its first instruction at 50 nanoseconds (assuming a five-stage pipeline and a clock cycle of 10 nanoseconds), and then an additional instruction completes every clock cycle. A parallel that can be drawn to an SMT machine, with the latency of entire programs instead of the latency of instructions. On a single-context machine with a traditional five-stage pipeline, a program could run for 1000 cycles. Now, with the SMT machine running four separate copies of that program, it may be finished in 1050 cycles (heavier penalties for branch prediction, cache misses, etc.). For the penalty of 50 extra cycles, perhaps two or three more programs have completed running.

4.1 Comparison Between SMT and Single–Chip Multiprocessor

Two of the most promising advanced architectures are the SMT and the single–chip multiprocessor. The single–chip multiprocessor is two–way issue superscalar core replicated two to sixteen times on the same chip to provide thread level parallelism (TLP). A diagram of the general architecture of both the SMT (Figure 4.2), and single–chip multiprocessor (Figure 4.3) are included. Currently, the technology is not available to do this on a large scale (eight, or sixteen processor cores). The chip
density may be available to pack two or four processor cores on the same chip. The comparison is valid, however, as by the time when the chip densities will allow for four or eight, threads on the same chip for SMT, eight or sixteen processor cores should be possible for a single–chip multiprocessor.

![Figure 4.2: General Architecture of SMT](image)

The design of a single–chip multiprocessor will be relatively simple in comparison to the design of the SMT because once one processor core is designed and verified, it is primarily a cut and paste to reproduce the pipelines. The SMT’s processor core is more complex, as all of the threads are present in the same pipeline. The pipeline for a single–chip multiprocessor is homogeneous to one thread. Tullsen et. al. [3] show that SMT processors will outperform single–chip multiprocessors. They also believe that the resource utilization will be higher in the SMT’s case. Hammond et. al. [18] performed a comparison between the single–chip multiprocessor and the SMT microprocessors. They indicate in their results that the single–chip multiprocessor will hold the slight advantage in performance (between 10 and 12 percent). Their experimental setup, however, compared a single–chip multiprocessor with 16 issue slots to the 12 issue slots used for SMT. In addition, they concur with Tullsen et. al. [3] in the fact that the SMT machine uses the functional units more efficiently.
For the same number of functional units, Tullsen et. al. [3] concluded that the SMT is superior in performance.

![Figure 4.3: General Architecture of Single-Chip Multiprocessor](image)

Both architectures will be underutilized if there isn't sufficient TLP. For a single thread, the single-chip multiprocessor will not suffer a performance penalty, in relation to the superscalar processor, as its core is the superscalar core. The SMT will share the cache subsystem amongst the threads. Cache design for SMT will be discussed in Section 4.4.3. The single-chip multiprocessor will have a distinct cache for each of the cores. Additionally, both architectures will only have a single memory interface for off-chip main memory, as the pinout of the chips (number of external pins) should be comparable. Adding another memory interface would not only require additional transistors in the chip, but also additional pins on the outside.

Other additional physical design concerns are the interconnect delays in the chip, or wire delays. The transistor count will continue to increase in the coming years, as transistor sizes are decreased. Therefore, the individual transistors will be able to switch faster and faster. The wires will become the problem, as the time to transmit a signal over a wire will dominate in comparison to the time that it takes the transistor to switch on or off. To offset this increasing wire delay, the design of the
microprocessors will have to be partitioned to achieve the highest clock rate. Hammond et. al. [18] give the single-chip multiprocessor the edge when it comes to being able to handle the increasing wire delays, as the design is already partitioned into the separate cores. Overall, the performances of the two microprocessors will be comparable, but which processor has the better performance may depend on the benchmark that is run. Both designs look to be superior to the current designs, and viable options for running multiple programs on single chip.

4.2 Design Limitations of Advanced Microprocessors

The two designs that are the roots of the SMT microprocessor are superscalar and multithreaded processors. The superscalar microprocessor was introduced in Section 2.3, and the multithreaded processor was introduced in Section 2.5. While both processors allow for CPIs under one, both also have their limitations to performance.

The inherent limitation in the superscalar microprocessor is the instruction-level parallelism that can be extracted from a program. Assuming an eight-way issue superscalar microprocessor, it would be difficult to find eight instructions that don't have any dependencies between them cycle after cycle. Even with advanced compiler technology, which will rearrange the instructions in an order to maximize the ILP a program has, the number of instructions that can be issued in a particular cycle will be limited by the dependencies amongst the instructions. Tullsen et. al. [3] referred to two types of instruction-issuing bandwidth waste caused by the superscalar microprocessor, vertical and horizontal. Waste occurs when the processor doesn’t have any instructions to issue. Vertical waste occurs when entire cycles are wasted because of an instruction cache miss, so no new instructions can be fetched, and therefore issued to a functional unit later in the pipeline. Horizontal waste occurs when the maximum bandwidth of instruction issue isn't reached. This waste occurs because of the lack of ILP in a program.

The fine-grained multithreaded processor will reduce the amount of vertical waste [4], but can’t overcome horizontal waste. When a cache miss occurs, a long latency will occur while the processor waits for the data to be fetched from memory. During this period, single-context processors have completely wasted cycles. Multithreaded processors, however, can switch to another con-
text, and run the other context with a small penalty (0–2 cycles). A limitation of the traditional multithreaded processors is that issue bandwidth will go unused when data hazards occur. The traditional multithreaded processor does have standard speculative, out-of-order methods implemented, but its use of functional units is limited to the ILP of the process. The processor has only the single instruction queue to issue from, and therefore some of the instruction issue bandwidth will go unused.

Simultaneous multithreading (SMT) offers a solution to both vertical and horizontal waste. It combines the features of both the superscalar processor and the multiple contexts of the fine-grained (traditional) multithreaded processor. It interleaves the instructions from multiple threads by selecting the instructions from multiple queues to issue on every cycle. In addition, the design may be such that the processor also fetches from multiple PCs on every cycle. In using a comparable number of functional units, the SMT processor can extract both higher resource and issue bandwidth utilization.

Consider a microprocessor with the following functional units (assuming all are fully pipelined, and therefore have an instruction issue latency of one):

4 **Integer ALUs** completion latency: 1 cycle

1 **Integer mult/div units** completion latency: 3 cycles

3 **memory ports** completion latency: 2 cycles (assuming data cache hit)

2 **Floating-point units** completion latency: 5 cycles

The following code, and Table 4.1 compare how the superscalar and SMT (running two copies of the program on different threads) microprocessors would handle issuing this segment to its functional units. The time that it takes for the instruction to complete is also given in the list above. While, these latencies are not very accurate, the idea of the exercise is just to compare the SMT issuing with the superscalar microprocessor. Both the superscalar and SMT issue assumes no out-of-order issuing, for simplicity, although that would be an integral part of both microprocessors. For the SMT results, a $T#$ indicates from which thread the processor is issuing the instruction.

38
The SMT machine's issue algorithm will select all of the possible instructions from the first thread, and then give the remaining issue slots to the second thread. Other possible issuing algorithms alternatives will be discussed later in the chapter.

<table>
<thead>
<tr>
<th>inst</th>
<th>code</th>
<th>description</th>
<th>FU required</th>
</tr>
</thead>
<tbody>
<tr>
<td>A)</td>
<td>LUI</td>
<td>R5,100</td>
<td>integer ALU</td>
</tr>
<tr>
<td>B)</td>
<td>FMUL</td>
<td>F1,F2,F3</td>
<td>FP ALU</td>
</tr>
<tr>
<td>C)</td>
<td>ADD</td>
<td>R4,R4,8</td>
<td>integer ALU</td>
</tr>
<tr>
<td>D)</td>
<td>MUL</td>
<td>R3,R4,R5</td>
<td>integer mul/div</td>
</tr>
<tr>
<td>E)</td>
<td>LW</td>
<td>R6,R4</td>
<td>memory port</td>
</tr>
<tr>
<td>F)</td>
<td>ADD</td>
<td>R1,R2,R3</td>
<td>integer ALU</td>
</tr>
<tr>
<td>G)</td>
<td>NOT</td>
<td>R7,R7</td>
<td>integer ALU</td>
</tr>
<tr>
<td>H)</td>
<td>FADD</td>
<td>F4,F1,F2</td>
<td>FP ALU</td>
</tr>
<tr>
<td>I)</td>
<td>XOR</td>
<td>R8,R1,R7</td>
<td>integer ALU</td>
</tr>
<tr>
<td>J)</td>
<td>SUBI</td>
<td>R2,R1,4</td>
<td>integer ALU</td>
</tr>
<tr>
<td>K)</td>
<td>SW</td>
<td>ADDR,R2</td>
<td>memory port</td>
</tr>
</tbody>
</table>

Table 4.1 indicates that thread number two actually took two additional cycles to finish as opposed to the superscalar processor's program. However, when the overall system throughput is looked at, the superscalar's and SMT's results are 1.57 and 2.44 instructions per cycle (IPC), respectively. That is a 55 percent increase in system throughput. A minimal penalty paid for a slower individual performance (22 percent decrease in thread two's IPC), although thread one of the SMT did finish on the same cycle. It should be noted that the performance metric instructions per cycle, or IPC is going to be used more often in this paper than its reciprocal CPI. The reason for this is that as the CPI becomes a smaller and smaller fraction, it is easier to understand the IPC growing larger and larger, as higher performance is reached.

In addition to the higher throughput realized by the SMT machine, the resources were better used. The vertical waste created by the superscalar microprocessor was reduced to one cycle by the SMT microprocessor. In addition, the number of cycles where the issuing slots were filled was increased from zero cycles for the superscalar to three cycles for the SMT. Laudon et. al. [16] indicate that increased number of processes increases the gain from interleaving. Interleaving is similar to SMT, as you are running two or more threads at the same time, but in separate cycles. The point still holds true for SMT. At a certain point, as the shared resources, such as the functional
units, cache, branch prediction, become a limiting factor as the number of threads increase. The design tradeoffs will be discussed in Section 4.4.

The fine-grained multithreaded microprocessor would have run the two threads, like the SMT, except that it would have switched between the two threads on every cycle, or every other cycle. Therefore, the instructions really don’t mix together, and the issuing of instructions to reservation stations remains relatively simple. This would eliminate some of the vertical waste, but the processor’s performance would still be limited by the ILP.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Superscalar Issue Slots</th>
<th>SMT Issue Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>LUI (A)</td>
<td>FMUL (B)</td>
</tr>
<tr>
<td>2</td>
<td>MUL (D)</td>
<td>LW (E)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADD (F)</td>
<td>NOT (G)</td>
</tr>
<tr>
<td>6</td>
<td>FADD (H)</td>
<td>XOR (I)</td>
</tr>
<tr>
<td>7</td>
<td>SW (K)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>T2.XOR (I)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>T2.SUBI (J)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Comparison Between the Superscalar and SMT Issuing Patterns

### 4.3 Additional Benefits

In addition to the ability to reduce wasted issuing slots, the simultaneous multithreaded architecture provides a benefit when a context switch occurs. There are three factors that will determine the percentage of time that the processor will spend on context-switches [16]:

1. The cache miss rate will help determine the total number of context-switches
2. As the processor spends less time idle, the relative percentage of time spent executing instructions increases.

3. As more contexts are added, the cost of a context switch decreases.

The reasoning behind the third case can be explained in the following formula:

\[
Time(ContextSwitching) = Time(SoftContextSwitch) + Time(HardContextSwitch)
\]

A hard context switch would be one that requires a context’s registers to be swapped out to memory, and the new processes’ registers to be brought in from memory. A soft context switch would be one that just requires the processor to select another one of the active hardware-supported threads to be run. The SMT architecture is continually context-switching at the lowest level, the instruction level, as it gives priority to particular threads in both the instruction fetch and issue stages. Therefore, the soft context switch does not cost the SMT processor any cycles, as it is already occurring continuously. The time for HW supported (soft) context switches on a blocked multithreaded processor is going to be in the 3–10 cycle range while the pipeline is flushed. Whereas, the time for external context switches is going to be somewhere between the 200–1000 cycle range because of the number of memory accesses.

4.4 Design Tradeoffs

Laudon et al. [16] describes the interleaved multithreaded processor as "an individual context is allowed to issue an instruction every \(x\) cycles (where \(x\) is the number of contexts in the processor).” That is not the same as the SMT, as the SMT can issue an instruction from one context during every cycle, but the similarity is there. [16] discusses the fact that an "interleaving scheme suffers when there aren’t enough processes to fill up the instruction issue slots." If the TLP is non-existent, then the SMT’s overall system performance is determined by the program’s ILP.

In the example above, Table 4.1, even the second thread didn’t completely eliminate the vertical waste. If a third thread was added to run on the SMT machine, then the vertical waste would be eliminated, and six (up from three with two threads) of the cycles would have the full issuing
bandwidth used. The third thread would finish in one additional cycle, resulting in a throughput of 3.0 IPC, which is an increase in throughput of 22 percent. Adding a fourth thread would push the throughput to 3.38 IPC, for an increase in throughput of 12 percent. This slowing of the throughput increase shows that eventually, when adding threads with no additional resources, the throughput will level off.

The majority of modern advanced microprocessors have some form of out-of-order execution. In out-of-order execution, the processor will issue instructions that are ready (no data dependencies pending) to their respective function units, regardless of the original instruction order. Once the instruction has completed its execution, the processor may also retire instructions out-of-order. According to Hily and Seznec [24], the out-of-order execution that occurs in many microprocessors, may not be worth the increase in design complexity for the SMT processor. In their findings, they point out that while the single- and multiple-thread execution will benefit from out-of-order execution, the difference between the two using four threads was as low as 9 percent. In addition, the systems today should rely on overall system performance (performance of all programs and their interactions), as opposed to the performance of a single program. This is becoming increasingly important as operating systems are spawning off tens of threads before the user even runs a program. This inorder execution will also reduce the design complexity, and allow for additional resources in other areas (cache, branch prediction, etc.). Although this topic should be considered for future applications, the out-of-order design will be the focus of this experiment.

In the SMT processor, there is the additional design complexity that occurs when attempting to issue from multiple contexts simultaneously. In particular, the control logic takes on the added responsibility of comparing the thread number tag that must be carried through the pipeline with the instruction. With a four-thread, eight-way issuing SMT microprocessor, during every cycle, four different instruction queues must be searched on every cycle. From those queues, eight instructions, must be selected in some priority. The priority-determination mechanism is important to the overall performance. In addition, when there are two, four, or eight sets of data attempting to occupy a single cache, there is a greater chance that conflicts are going to exist. The cache design is especially important to the multiple-context processor [21], as the multiple data sets create
additional stress to a memory hierarchy that is already lagging behind. Other tradeoffs that must be taken into consideration are in the decode and commitment stages when the register files are accessed.

4.4.1 Thread–Priority Determination

Instructions from different threads are kept separate for the majority of their lifetime in a SMT processor. For instance, it begins with the fetch stage, where the instruction is fetched into a thread’s fetch queue, and then passed on to the decode stage, and eventually the dispatch stage, where the threads are mixed with the other thread’s instructions as they are placed reservation stations (using a pseudo–Tomasulo approach) and on to the functional units. Once finished with the functional unit, the CDB will be used to "publish" the results to both the pending instructions, and the commitment buffer, which will write the results back to the register file.

While working in the separate thread’s resources (buffers and register files), the priority of an instruction is simple, first instruction in, first instruction out. The exception of this is if out-of-order commitment is allowed. If that is the case, the first instruction that is finished has its results written to the register file. The complexity, and also design choice, occurs when attempting to mix all of the threads’ instructions together to be issued to the limited resources. This occurs in two places: the fetch and the issue stages.

In the example above, the first thread had the highest priority, and would continue to issue until it could no longer issue. Once it was done, the next thread would be given whatever room was left over until either there were no more instructions to be issued, or the issue bandwidth was consumed. This is not a fair strategy, but it is a relatively simple algorithm.

There are alternative algorithms for issuing instructions to functional units. The simplest algorithm is round–robin issuing, where the processor uses a simple counter to select a thread. If that thread has an instruction to issue, then it will issue, and the processor will move on to the next. Another fairly simple algorithm is the oldest instruction first. In that scheme, the instruction that has been in the dispatch queue the longest is selected. Tullsen et. al. [4] point out that deciding whether or not a particular branch path is correct or incorrect is crucial to the processor's
They examine the SPEC_LAST, OPT_LAST, and BRANCH_FIRST algorithms. SPEC_LAST will give lowest priority to speculative instructions. OPT_LAST will issue instructions following a branch last after all other instructions have been issued. The BRANCH_FIRST technique is used to issue branches first, in order to know whether or not the prediction was correct as soon as possible.

These issuing priority–selection methods provide the first level of determination. In the ICOUNT feedback, the thread with the smallest number of instructions in the static part of its pipeline (fetch and dispatch queues), is given the highest priority. This technique pushes the highest bandwidth threads to the top priority, and therefore promotes higher overall system throughput. A thread with a small number of instructions in its pipeline is pushing instructions through at a higher rate, and should be rewarded with higher priority.

When starting in the fetch stage, there are many options to chose from in order to determine which thread fetches on any particular cycle. Similar to the issue solution, the ICOUNT technique will be focused on, as it is relatively simple, and promotes high throughput. Tullsen et. al. imply that the ICOUNT technique fulfills two objectives (in addition to promoting high throughput) in the fetch cycle: ICOUNT prevents fetch queues from being filled, and "it provides an even mix of instructions from the available threads, maximizing the parallelism in the queues" [4]. This solution avoids profiling the instructions that are in the pipeline already. In addition the ICOUNT method also helps avoid filling up the instruction queues.

4.4.2 Register File Issues

In the SMT architecture, the register file becomes an area for heavy redesign. There are now two, four, or eight sets of registers, and the processor determines from which register file to read. This is going to cost extra time during each register read and write. The transistor cost won’t be as important as the time it will take to access the registers. One technique by which to access the correct register file for a particular instruction is to translate a virtual register number, which is the register number that the program indicates, to a physical register, which is the number of the register in hardware. The register file is a type of content addressable memory (CAM). CAM is a
memory in which it is accessed by the value which it holds. Registers hold two pieces of information:

1. Register number

2. Data

Now, the thread number could be prepended to the virtual register number to indicate the physical register number. This could require an additional cycle penalty (for an additional pipeline stage) to decode this information, such as Tullsen et. al. [4] use in their pipeline (see Figure 4.1). In addition to the standard registers that are visible to the programmer, there are registers used for renaming which will have to be duplicated. Register renaming is a technique used to reduce the number of name dependencies, where there is no data interaction. They also add an extra pipeline stage to write back to the register file at the end of the pipeline, in the commit stage. The additional cycle in the decode stage increases the branch misprediction penalty, and this is what causes the slight decrease in single-thread performance in [4].

4.4.3 Cache Design

Now there are two, four, or eight sets of data and instructions that are going to be competing for the limited cache. Anaconda [26], a block multithreaded processor, uses a fully associative cache to take full advantage of memory. This is a more complicated structure, but the replacement algorithm used is Not Last Used. This has better performance than random replacement, but is simpler to implement than Least Recently Used, as it only requires a single bit. In addition, the penalty for a cache miss is going to increase, as instead of having a single cache attempting to fetch the data from the next level (L2 for an L1 miss, main memory for an L2 miss, etc.). There may be multiple requests for the L2 cache simultaneously. It is necessary for the memory hierarchy to be a lock–up free cache, which means that cache can handle multiple outstanding memory requests. Thekkath and Eggers [21] indicate that the cache must be able to handle a minimum of one outstanding memory request per hardware context.

In a single–context processor, there are many choices when it comes to cache design. There may be one, two, or three levels of cache. A designer could add another level, but the cost in hardware
may outweigh the gain. Any level of cache may be unified, where both the instruction and data memory is combined into one structure. The caches could also be separated into the instruction and data caches, which allow more accesses with one or two ports per structure.

The threads may share the cache, whether it is instruction or data, or the threads may have private caches. A private cache will cancel interthread cache conflicts, but will have to be smaller to fit in the same space, and therefore single-context cache conflicts will increase. A private cache, on the other hand, will only have to have a standard single or dual port for the thread to gain access. A public cache will allow all of the threads' working sets to take better advantage of a larger space. It would also require a field in the cache to indicate to which thread the cache entry belonged. According to Hily and Seznec [23], the shared first level of cache for multiple threads provides better performance until a larger number of threads are used.

The next choice to be made would be the size of the cache. Tulls in et. al. [4] indicate the increased workload on their design did not create thrashing with the same sized cache. Tulls in et. al. [3] performed cache performance experiments on the different configurations, both private and shared L1 caches. The L2 cache configuration is shared amongst the different threads. Their design separated the cache connections into banks. Their results showed that the shared data cache will allow a single thread to allow "multiple memory instructions to different banks" [3]. For multiple processor environments, the shared data arrangement also allows for cache coherence (within the chip) without any modification to existing cache designs.

The added contexts in hardware put a greater strain on the memory hierarchy, as the number of memory requests in the same amount of time has increased by the number of contexts in hardware. Hily and Seznec [23] contests the cache performance claim made in [3] saying that the benchmarks that were run are known to not produce a representative amount of stress on the memory hierarchy. They cite that the benchmark does not create a large number of L1 cache misses, therefore the performance/contention on the second level of cache is not measured accurately. Even with a good first level of cache performance, when the L2 is presented with a request, it must be able to contend with the multiple requests for L1 cache misses coming from the different threads simultaneously.
4.4.4 Additional Resources

In addition to the strain on the memory hierarchy, increased design complexity for the fetch and issue stages, and handling the increase in register file size, other resources must either be duplicated, or modified to handle the shared workload of the processor. For instance, branch prediction now has one cache of predicted addresses for two, four, or eight times the number of branches that it faces. The Translation Lookaside Buffer (TLB) will be facing the same increase in the number of working sets. There must also be an exception handler per hardware context.

For branch prediction two requirements must be met, the predicted values of branches, and the return addresses for subroutines. The hardware should also support one stack per hardware context for the subroutine return addresses. The stack, or RAS (return address stack) will help the fetch unit predict where to go on return from subroutine instructions. Hily and Seznec [22] indicates that a 12-entry RAS should allow for adequate performance. The RAS is a circular queue that holds subroutine return addresses, and there shouldn’t be a large hardware cost (transistors) for one per hardware context.

Research into the benefits/disadvantages of sharing the branch prediction has been studied in [4, 22]. Branch prediction is heavily researched because as Hily and Seznec [22] point out, “most applications exhibit a ratio of 15% to 30% of branches to other instructions.” The conclusion in [22] that as long as the branch prediction mechanism is proportional in size to the number of contexts that the hardware supports, there should be very little thrashing amongst the different threads. Tullsen et. al. [4] state that the increased workload exhibited no "thrashing" in the branch prediction. However, if a shared branch prediction structure is going to be used, a tag to indicate which thread the prediction entry is associated with will be required. This is necessary because in a virtual memory system, different threads may use the same virtual address, as it is the virtual address that is stored in the branch prediction table. The indication of a thread number will avoid branches occurring where there is no branch, and when there are branches, to avoid incorrect targets (thread one getting a target for thread two).

Another option for the added branch prediction would be a branch prediction unit for each of the threads. This would have two benefits. First, existing branch prediction designs could be used
without any modification. Each thread will be directly connected to a single branch prediction unit, so it wouldn’t collide with the other threads. The second benefit is that there will be no need to increase the branch prediction unit’s access bandwidth. With multiple threads accessing the branch prediction unit, it will have to have multiple ports to keep up the requests being presented to it. With multiple branch prediction tables, only the bandwidth for a single thread will have to be attained.
Chapter 5

Proposed Architecture

This chapter will provide a more in-depth overview of the changes to the standard architecture that will be necessary in order to implement a simultaneous multithreaded approach.

5.1 CPU Pipeline Modifications

While Tullsen et. al. [4] claim that major changes to the current superscalar processor will not be required, some changes will have to be made. This section will identify those changes, especially the changes to the core pipeline stages, and their components.

5.1.1 Register File

By definition, each of the threads in the machine must have its own hardware context dedicated to it. Therefore, there must be $N$ sets of registers in the CPU. In the implementation of the architecture accessing these registers is going to be a major bottleneck. In an attempt to reduce that bottleneck, when attempting to access a register, the thread number will be compared to the register label. This will also keep the design hierarchical in nature, as opposed to a large pool of registers labelled $R0$ to $R127$. By prepending or appending the thread number to the register number of the register to access, a set of register “banks” will be created. The effective compare operation that will occur for a register access will then be a 7-bit compare for both integer (fixed point) or floating point registers, for four threads with 32 registers each.
<table>
<thead>
<tr>
<th><strong>Register</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter (PC)</td>
<td>This points to the address to fetch the next instruction from for a particular context.</td>
</tr>
<tr>
<td>Stack Pointer (SP)</td>
<td>This is used for the per-process stack. The stack is used in order to pass parameters between subroutines and other functions.</td>
</tr>
<tr>
<td>General Purpose Integer Registers</td>
<td>These are manipulated by the user program. Register 0 (R0) hardwired to zero, to be used as a reference. Modifications made by the user program will always be neglected.</td>
</tr>
<tr>
<td>Floating Point Registers</td>
<td>The floating point instructions will manipulate these registers. One option is to make two consecutive 32-bit registers (such as F0 and F1) act as one 64-bit register for a double precision number.</td>
</tr>
<tr>
<td>Text Base Address Register</td>
<td>Memory protection register indicating where the beginning of the code is located in memory. Memory checks will be performed for every memory access using this register and the Text Limit Address Register.</td>
</tr>
<tr>
<td>Text Limit Address Register</td>
<td>Memory protection register indicating where the end of the code for a particular thread is located in memory. Memory checks will be performed for every memory access using this register and the Text Base Address Register.</td>
</tr>
<tr>
<td>Data Base Address Register</td>
<td>Memory protection register indicating where the beginning of the data for a particular thread is located in memory. Memory checks will be performed for every memory access using this register and the Data Limit Address Register.</td>
</tr>
<tr>
<td>Data Limit Address Register</td>
<td>Memory protection register indicating where the end of the data for a particular thread is located in memory. Memory checks will be performed for every memory access using this register and the Data Base Address Register.</td>
</tr>
<tr>
<td>Hi</td>
<td>This register becomes the upper four bytes of the results from an integer multiply or divide.</td>
</tr>
<tr>
<td>Lo</td>
<td>This register becomes the lower four bytes of the results from an integer multiply or divide.</td>
</tr>
<tr>
<td>FCC</td>
<td>This is the floating point condition code register. It will hold the conditions for floating-point operations (such as negative, overflow, zero, etc.)</td>
</tr>
</tbody>
</table>

Table 5.1: Register File Description

50
The reorder buffer and the reservation stations will help keep track of the status of all available registers. The register update unit, see Section 6.1.1, is a combination of both the reorder buffer and a reservation station in the simulations that were performed.

The entire set of registers (which includes the general purpose integer registers and the floating point registers and the special purpose registers) make up the machine state for any particular context. The special purpose registers include the program counter, stack pointer, and condition code register, as the registers are described in Table 5.1.

The collection of all registers, called the register pool, will contain all of the registers to which the CPU has access. The reorder buffer and the reservation stations will help keep track of the dependencies, and will determine when a particular register can be committed or retired. In simulation the additional registers used by the register pool for register renaming are not tracked. The chains of instruction dependencies are tracked by create_vectors, tell the dependent instructions when their operands are ready.

5.1.2 Instruction Fetch Stage

The first step for any CPU is fetching instructions from memory. This memory may be in the form of DRAM, or I-cache. SMT will place a great deal of stress on the fetching stage, so the memory hierarchy’s design is a very important step in defining the architecture. In this design, the L1 cache was separated into instruction cache (I-cache) and data cache (D-cache).

Priority is determined (based on the number of instructions in the queues, and whether or not the thread is currently waiting on an I-cache miss), and the two highest priority threads give their respective program counters (PC) to the two ports on the I-cache. The I-cache, provided that the instructions are cached, will then fetch a block of (up to four) instructions. The instructions will be passed onto the decode stage for further processing. A diagram of the fetch unit is shown in Figure 5.1.
The TCU is the thread control unit, and will be used to determine the priority of the threads. There will be multiple PC units working concurrently on every cycle. The PC units will have the task of updating the program counter for a particular thread. The new PC value can come from a number of places:

- *Instr. Addr. + 4*: if the process continues in a sequential fashion.
- *BTB Target*: this would indicate that it has an entry in the BTB, and it is predicted taken.
- *Exception Vector*: if an exception occurs, the PC may need a completely different address.

Each exception's address is predetermined by the CPU and OS.

The fetch unit will only fetch from one or two of these addresses during any cycle. The Thread Control Unit (TCU) will indicate which context should be fetched during any particular cycle.
These two encoded values will be latched and passed on to the decode stage, so the instruction blocks (128 bits wide) can be demultiplexed into the appropriate instruction queues, and then continue to be processed by the decode stage.

The I-cache will be dual-ported. This choice will complicate the design of the cache. This is a necessary step, as the SMT design in itself requires a higher bandwidth from memory. Dual-ported cache designs have been implemented [29], so their use is not a great leap in terms of what is available today for design. In the simulation, the cache’s bandwidth will not be tracked, so this feature’s impact in the overall system performance will not be measured.

5.1.3 Instruction Decode Stage

The decode unit must take care of the following things:

1. Classify a bit pattern as to the type of instruction (data transfer, arithmetic/logical, control, special, or floating-point) it represents.

2. Be able to fetch the register values; the value to be fetched will be determined by the encoded value within the instruction. Which bits will indicate a particular register will be determined during instruction set definition.

3. Help the control unit determine which types of dependencies exist among instructions. This is done by writing each instruction into the thread’s reorder buffer, which lists the input and output dependencies (registers).

When the instructions (4 instructions wide maximum) are returned from the I-cache, they are demultiplexed into one of the four individual decoding queues, as shown in Figure 5.1. In the Simple Scalar toolset, there isn’t a true decode stage. All instructions are pre-decoded in order to speed up the simulation. The dependencies are recorded, however, in the dispatch stage.

At this point, the instructions are decoded, but the register values must also be accessed from the register file. This may take more than one clock cycle. If each of the threads has its own individual decode queue, then the problem with accessing multiple contexts may be avoided, as
they will all have direct access to a particular set of registers. Register renaming logic will also need to be implemented at this point.

If only two threads are fetched during any one cycle, the decode will not be fully utilized. However, if there is a stall from the dispatch stage, the decode stage may not be able to forward its results. To avoid the decode stage stalling any thread that may have all required resources available to it, the thread control unit (TCU) must also account for stalls in determining whether or not a thread can have instructions fetched during a particular cycle. If the instruction queue is full, then the fetching for that thread should be disabled. The ICOUNT feedback (Section 4.4.1 method helps avoid attempting fetches to full instruction queues, as this method will give the highest priority to the emptiest queues.

The instruction decode stage in the standard in-order, five-stage pipeline need only decode the instructions and fetch its operands from the register file. In an out-of-order processor with dynamic pipeline scheduling, the traditional decode unit’s tasks are divided into two different stages: instruction decode and instruction dispatch (also referred to as the issue stage). The instruction decode stage determines what type of operation the instruction is attempting, and fetches its operands from the register file. In many cases, the most recent operands may not be located in the register file, as it may still be completing in the pipeline. In this case, the operand field is replaced with the register number.

During decoding, in order to gain the ability to perform out-of-order execution and still handle precise exceptions (see Section 5.2 for exception models), a reorder buffer must exist. In order to preserve a hierarchical design within the SMT machine, a separate reorder buffer will exist for each of the hardware contexts. In terms of design, the reorder buffers will require the addition of the context ID.

In the reorder buffer, there must be a way in which to keep track of the logical order. For this, a number will be assigned to each instruction. Therefore, there will have to be an instruction number generator, which will create a unique number for each instruction as it placed into the reorder buffer. This number will in putting the instructions back in order, once they finish the execution stage.
The instruction number is limited by the size of an `int` in the simulator, but in a silicon-based system, there would have to be a limit on the range of the numbers generated. Given ten bits, there is only 1024 different numbers to be generated. However, the total number of instructions that the CPU executes will be greater than that number. The architecture can get away with limited instruction numbers as long as the limit of possible instruction numbers is larger than the number of instructions that a thread could have in the CPU at any time. When the count wraps around to zero, a certain “window” of numbers between 1000 and 1023 that may still be in the pipeline. These instructions are determined to be older than instruction numbers zero even though their instruction numbers wouldn’t show that to be true.

There are two approaches that can be made for allocating instruction numbers in the reorder buffer:

1. Allocate all instruction numbers globally

2. Allocate instruction numbers locally (to a particular thread)

The second approach leads to a more hierarchical design, as an instruction is written back to a thread’s specific reorder buffer, which can be a smaller unit, and therefore allow faster access. If a single global instruction reorder buffer is used, then its access will be much slower as it will have to deal with a “queue” with thousands of entries.

The instruction decode stage must also tag on the thread number onto the information that is extracted during decode. This will allow the instruction to be placed into the correct reorder buffer and instruction buffer. The instruction buffers will be replicated in order to handle four different threads for our example. The separate buffers allow each of the different thread’s instructions to be handled independently of the other threads.

5.1.4 Instruction Dispatch/Issue Stage

This stage will read instructions from the instruction queues into which they have been placed by the `decode` stage (see Section 5.1.3). This stage must have a hardware scheduling mechanism that will allow it to select instructions from different threads to be issued to the functional units. The
priority determination is the same as the fetch stage (see Section 5.1.2), where highest priority goes to the thread with the smallest number of instructions in the instruction queues. A simpler scheduling algorithm, such as round robin determination, may also be used.

The instructions that are selected will be sent to the reservation stations if their operands are available, and there is a reservation station available. The instruction will wait in a reservation station until the appropriate functional unit is ready. The dispatching, or issuing, of instructions to the reservation stations may not be in the original instruction order, as the oldest instruction doesn't have to be first to the reservation station (although it does have priority over other newer instructions).

The interaction of the dispatch execution, and writeback stages is shown in Figure 5.2. It is not a complete description as it only shows a single integer ALU and the floating point adder. The functional units that are used in this architecture are shown in Table 5.2. In the simulator, the dispatch and issue stages are two separate stages. The difference between the two will be discussed in Section 6.1.

5.1.5 Instruction Execution Stage

In general, the SMT architecture with its modified issuing approach should use the resources given to it more efficiently. However, since there is an eight instruction issue bandwidth, more instructions are being placed into functional units. Therefore, between 12 and 14 functional units are used in this architecture, as opposed to the 12 functional units that were used in the sim-outorder simulator, which is a standard out-of-order simulator. The functional units that are included in the SMT design are included in Table 5.2.

5.1.6 Writeback Stage

This stage is responsible for writing the results of the execution units back across the CPU’s internal data bus to those units that need the results. The results will be fed back to the reorder buffer, the dispatch queues, and to the functional units’ reservation stations that are awaiting a result.
### Table 5.2: Functional Units

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Quantity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Unit</td>
<td>4</td>
<td>This is the basic integer ALU that will perform logical and arithmetic operations on integer instructions.</td>
</tr>
<tr>
<td>Floating Point ALU</td>
<td>4</td>
<td>This performs many of the basic floating-point functions such as add, subtract, compare, and convert. This execution unit will have to access operands from the floating point register set.</td>
</tr>
<tr>
<td>Integer Mult/Div</td>
<td>1-2</td>
<td>This unit will perform the integer operations of multiplication and divide. Since 32-bit numbers produce 64-bit results, the extra bits will be stored in the HI and LO registers.</td>
</tr>
<tr>
<td>Integer Unit</td>
<td>2</td>
<td>This is a memory port for reading and writing results to data memory. It is fully pipelined, so a new value may be (theoretically) written/read from it every cycle.</td>
</tr>
<tr>
<td>Floating Point Mult/Div</td>
<td>1</td>
<td>This performs floating-point multiplication and division, which are both very lengthy operations.</td>
</tr>
</tbody>
</table>

In this way, the instructions that are waiting in the reservation stations (see the dispatch and execution stages) will not have to wait for the results to be completely written back to the register file. This is the manner in which an out-of-order processor does data forwarding.

When the instructions' results are written back, the reservation station that held the instruction is marked as empty, and the functional unit is already processing another instruction. In the next clock cycle, the dispatch stage can issue another instruction to that reservation station.

During the writeback stage, the reorder buffers will also determine which instructions can be written back to change the system state. In this system, only in-order retirement of instructions is allowed. This allows for the preservation of precise interrupts and exceptions. The actual program state won't be changed until the next stage, the commit stage.

The *Common Data Bus (CDB)* is a data bus that will transmit not only the results, but additional information like target register, hardware thread, and instruction address. The CDB and the reservation stations perform the data forwarding necessary for out-of-order execution. The hardware thread number will have to be appended to the information passed throughout the processor. When comparisons are performed, the thread number will have to be compared first. Also, there is one reorder buffer per hardware context, so there will have to be logic to determine
into which buffer to write back the instruction.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>2 bits</td>
<td>This is the thread number associated with the result.</td>
</tr>
<tr>
<td>Register</td>
<td>7 bits</td>
<td>This is the register that the result is supposed to be written back to. It will also be checked by the dispatch unit and reservation stations to see if they need to this data.</td>
</tr>
<tr>
<td>Data</td>
<td>32 bits</td>
<td>This is the result of an operation.</td>
</tr>
<tr>
<td>Address</td>
<td>32 bits</td>
<td>This is the address of the instruction that the result is taken from. This will be used to handle exceptions in the Commit Stage.</td>
</tr>
<tr>
<td>Instruction Number</td>
<td>10 bits</td>
<td>This is the instruction number used in the reorder buffer. This number will be used in the Commit Stage to determine which instruction is ready to be committed.</td>
</tr>
<tr>
<td>Exception</td>
<td>1 bit</td>
<td>This is used by the commit stage to determine whether or not an exception occurred during the execution of this instruction. Taking care of exceptions at commit time allows for precise exceptions in the CPU.</td>
</tr>
<tr>
<td>Valid</td>
<td>1 bit</td>
<td>This will be used to remove instructions that were executed and then were invalidated.</td>
</tr>
</tbody>
</table>

Table 5.3: Information Forwarded Via the Writeback Stage

Logic will be required to determine which instructions in the reorder buffer can be committed. The commitment bandwidth will be determined by the register file bandwidth. In order for the two instructions to be committed simultaneously, they must not have any hazards between them. Also, an instruction may not be committed if it generates an exception. Therefore, if instruction $i$ generates an exception, and instruction $i+1$ follows it, neither instruction may be committed (regardless of whether instruction $i+1$ generates an exception). However, if instruction $i+1$ generates an exception and $i$ does not, instruction $i$ can and should be committed.
Figure 5.2: Modified Tomasulo Approach for SMT
5.1.7 Commit Stage

This stage will be the final committing of the results back to physical (instruction set) architectures and will handle any exceptions. The Exception bit in the forwarded data (from the Writeback stage) will determine whether or not the exception handler (set up by the operating system) should be invoked.

In order to understand this stage, several terms must be defined. Once an instruction reaches the commit stage of the pipeline, it will have been completed. Completion of an instruction (also called graduation [11]) involves the instructions whose executions are being completed. For an add, when the result is ready from the functional unit, it is considered completed. For a branch instruction the completion occurs when it gets to the head of the reorder buffer and it has been determined that the branch prediction was correct. If the branch instruction gets to the head of the reorder buffer, and it was determined that a branch misprediction has occurred, all instructions after the branch instruction must be flushed. Commitment of an instruction occurs when the state of the processor has been updated, with the instruction's result being written back to the register file.

An advantage that the SMT architecture holds over conventional single-threaded processors is that in a single-threaded machine, if a branch is mispredicted, the entire pipeline has to be flushed and it suffers a penalty of the number of cycles up to the commit stage (depending on pipeline depth). In a SMT machine, however, even though that one particular thread may have to suffer a penalty for flushing its pipeline, the other threads' activities should be able to "mask" the penalty. SMT isn't going to improve any particular thread's individual performance, but it will allow overall system performance to increase. A disadvantage of the SMT processor's pipelines, is that the control logic for flushing the entire pipeline will be more complicated.

Pipeline flushing requires that a VALID bit be present as the instructions are sent through the pipeline. If a branch instruction is mispredicted, when it reaches the commit stage, all instructions that immediately follow it must be flushed from the pipeline. The control unit will take the instruction number that caused the incorrect prediction and invalidate all instructions that occur after it in the reorder buffer. At this point, the program counter will have its value updated with
the correct value. Until the first instruction with the new program counter reaches the commit stage, no instruction will be allowed to commit its value to the register file.

The commit stage is where a thread will handle the exceptions that occur. Exceptions are discussed in Section 5.2. The exception handling hardware must perform a number of tasks in order for the software exception handling routines to perform their job. If an exception occurs, the fetch stage (Section 5.1.2) must also be made aware, as it will have its PC changed to the appropriate exception vector. It must push the current program counter and stack pointer onto the kernel stack, and it must also push the exception status onto a stack, so the OS knows what type of exception occurred. These operations may either be performed in hardware or in software. The software exception handler will be responsible for saving the remainder of the context's state.

There is in-order commitment of instructions. However, since there will be multiple queues of instructions (for the different threads), commitment is more complicated. At this point, the register file must be multi-ported, as more than one instruction from a thread may write back its results, if they are completely independent. In any particular thread, if the oldest instruction (in order) can't be committed (retired), then no instructions can be committed for that thread on this cycle. This will create the required precise exception handling.

## 5.2 Exceptions

Exceptions are “errors” that occur during the execution of a program. These errors may occur at the same time during each program (a synchronous exception), or at different times (an asynchronous exception). Exceptions may also be used to force the use of the operating system. For instance, when a process is waiting on an external event, it will use a system call such as `sleep`, and the exception handler that the operating system provides will be executed when the system call instruction is committed. In general, all exceptions will be handled by the processor when the instruction reaches the commit stage. Some examples of exceptions are:

**Page Fault** The memory address that was requested by a user program is not located in user memory. At this point, the operating system is going to have to fetch the memory from a secondary storage device (hard drive).
Invoke OS  The user program must wait on some event to occur, so it will put itself to sleep via a system call such as \texttt{sleep} or \texttt{wait}. At this point the operating system must swap this process out and bring in another process that has been waiting for some CPU time.

Timer Interrupt  Operating systems require a timer in order to keep multiple processes running with some regularity. When the timer interrupt goes off, the OS can then use its process management routines to determine which process will be activated. The timer is also used to collect system management information from the CPU.

Floating Point Error  A floating point operation may have created an undefined result such as overflow or underflow. The handling of this operation will be determined by the operating system.

Divide By Zero  A division operation was attempted where the divisor was zero, thereby producing an undefined result.

There are two different models when dealing with exceptions, precise and imprecise. Each of these models has tradeoffs involving the ease of design, time at which a register may be released, and ease of keeping the correct execution state after an exception. The models differ in when a physical register, which is mapped from a virtual register, may be released for use. The precise exception model holds more rigid requirements concerning the time when a register may be released for use back into the register pool. The gain associated with this strictness is the ease of returning the CPU to its pre-exception state. This is important especially when considering that an operating system has a significant portion of its code run through exceptions.

SMT System Call Handling

When the TRAP, or SYSCALL instruction is executed, the CPU will put that particular thread into kernel mode. In a single-threaded CPU, the PC, user SP, and the processor status word are pushed onto the per-process kernel stack [7]. From this point until the exception-handler returns control of the processor to the user program, all references to the SP use the kernel's SP. The exception status must also be pushed onto the stack, so the exact nature of the exception can
be determined (and in turn, the exception can be handled). An assembly language routine saves the registers, after which the system call handler is executed. Upon completion, the system call handler executes a RTE (return from exception) instruction, and returns one or zero through a register (register seven in this architecture). The SP, PC, & processor status word are restored in the opposite order from in which they were saved. At this point, all SP references again use the user SP.
Chapter 6

Simulator

A set of results is only as good as the model that created them. In this case, the model was the sim-SMT simulator. The main reason for using a simulator is to speed up the development cycle. A simulator uses the faster software development cycle to verify that a microprocessor, or some other hardware device, is performing as expected. The other choice would be to continually produce the hardware in silicon, which is a very slow and expensive process.

There are a variety of classifications of simulators. There are functional simulators, which will produce the results that the programmer will see, but ignore all of the internal details. This is testing the architecture of a microprocessor, more specifically the instruction set. A performance simulator concerns itself with the internal details of a device, which are also referred to as the microarchitecture of the device. The performance simulator usually deals with time [28]. There is another classification of simulators, either trace-driven or execution-driven. A trace-driven simulator reads a previously-generated list of instructions, and executes them. The execution-driven simulator generates its own trace while it is running. Sim-SMT is an execution-driven, performance simulator.

Sim-SMT was derived from the architectural SimpleScalar [28] toolset, which was developed at the University of Wisconsin. The original toolset provided all of the necessary supporting code used to simulate the operation of an out-of-order speculative microprocessor. In addition to the out-of-order simulation, it provided a profiling simulator, an in-order option to the out-of-order simulator, a fast simulator, a debugger, and a manner by which to graphically display the pipeline
for portions of the code. It also provided routines to simulate cache, functional units, registers, memory, and branch prediction.

In order to simulate a microprocessor, there are many supporting functions that must be in place. The most important of which is a compiler, so that a program can be written in a higher-level language program, and it would be translated to a native language (machine code) that the microprocessor, or simulator in this case, could understand. The toolset was augmented with a group of binary utilities that had been modified for use with the Simple Scalar instruction set. In addition, the libc standard C library has been ported for use with Simple Scalar.

Running a program will generate a group of statistics. The statistics range from the number of instructions that are dispatched, to the number of simulated machine cycles that it took to complete the program. The simulator also allows for statistical formulas to be entered. The statistics are entered into the program code, and updated as the program runs. When the simulator exits, the statistics that have been gathered are printed out to the screen, through standard error.

6.1 Simulated Pipeline

The simulator, sim-SMT, simulates a processor pipeline of six stages. Five of these stages are controlled by a separate functions [10]:

ruu_fetch Instructions are fetched from memory in this subroutine. The address is checked to make sure that it falls within the text boundaries of the thread. If it doesn’t, which is possible with mis-speculation, then a NOP instruction is sent into the pipeline. If an instruction cache miss occurs, then a penalty is assigned to the violating thread. If the instruction is a control instruction, then the branch prediction unit will produce the predicted PC value. If a branch is fetched, then the fetch unit will stop fetching instructions for that thread during that cycle, unless the branch prediction routine produces a value. In sim-SMT, the fetch unit will fetch up to four instructions from two different threads. The two threads to be fetched are those that have the highest priority, according to the ICOUNT method. Penalties for cache misses, branches, and branch mis-predictions are enforced in this subroutine, as a thread cannot fetch if its ruu_fetch_issue_delay value is non-zero.
ruu_dispatch  Instructions are entered into the RUU and LSQ (load store queue) in this subroutine. The function will continue to dispatch instructions for a thread, until either the RUU or LSQ fill up, the decode bandwidth is used up, or the fetch queue is emptied. In addition to being placed into the RUU and/or LSQ, the actual instruction is executed in this subroutine. If the instruction is a memory reference instruction, it will be placed in the RUU and the LSQ. Many of the statistics are added in this routine to keep track of the total number of certain types of instructions.

ruu_issue  This stage is the first stage where instructions from different threads are mixed together. This function will take the highest priority thread and issue from it until it either meets its issue bandwidth, or there are no more instructions to issue from that thread. Then, the second highest priority thread does the same, and this continues until all possible instructions, from all threads, are issued, or the issue bandwidth (ruu_issue.width variable) is met. Instructions are only issued if there is an available functional unit. If there isn’t an available functional unit, then the instruction is placed back in the ready_queue. After all possible instructions have been issued, the ready_queue is reclaimed and sorted for those instructions which couldn’t be issued. There is an option to run a second issue subroutine that performs a round robin issue technique (ruu_rr.issue).

ruu_writeback  This stage scans the event_queue for all instructions which are finished with their respective functional units. The instruction is then marked completed. At this point, mis-predicted paths are also found, and handled. If a path is found to be mis-predicted, the ruu_recover subroutine is called to put the simulator back on the correct path, and a branch penalty is recorded for the thread. If the instruction has output dependencies, the instructions that are relying on the output are updated. If one of those instructions is then ready to be issued, it is enqueued into the ready_queue.

ruu_commit  This subroutine emulates the commit stage of the pipeline. In this stage, instructions are committed, and made invalid by increasing the tag value. Loads and stores are completed in this stage. Once instructions are committed, the RUU entry that they were tracked in, is
released for use. The LSQ entry is also released if this was a memory instruction.

The sixth stage is the execution stage which is controlled by the functional units, the event queues, and the function ruu_release_fu. A diagram showing the pipeline of the simulator is shown in Figure 6.1. In the diagram, there is an indication of an instruction decode stage. However, that stage does not truly exist. The decode occurs during the simulator’s initialization. A pipeline stage could be inserted to do the decoding, which would make the pipeline a more accurate representation of the out-of-order processor core that it represents. The simulator uses many #defines to create the instruction set.

![Diagram of Simulator Pipeline](image)

**Figure 6.1: Pipeline of Simulator**

In addition to the explicit stages, that are listed above, there is an execution stage that is controlled by the functional units. A functional unit resource is a struct has the following fields:
**name** The name of the resource.

**quantity** The number of the resources that exist in the simulation.

**busy** This field will be 0 until an instruction is issued to it. At that point, the oplat will be assigned to this to indicate how much longer until this resource is free.

**class** This is the matching resource class. In order to be issued to this resource, instructions must have this resource class.

**oplat** The operation latency, which is the number of cycles until result is ready for use.

**issuelat** The issue latency, which is the number of cycles before another operation can be issued to this resource.

As can be seen, the resource itself does not handle any of the calculations. The resources are used to keep track of which resources are filled, and when they can be issued to again. The resources will also indicate when an instruction has finished its execution, and is ready for the writeback and commit stages.

### 6.1.1 Register Update Unit (RUU)

The main entity that tracks the progress of a small window of instructions is the RUU, or register update unit. It is the center of the pipeline, as all of the stages, except for ruu_fetch, interact with the RUU. The RUU keeps track of the following items:

**thread_number** The thread from which this instruction is executing.

**IR** The actual instruction bits fetched from memory.

**op** The decoded instruction opcode.

**PC** The program counter value where this instruction was fetched.

**next_PC** The correct value for the next PC.

**pred_PC** Predicted PC value for this instruction. This may only used for branch instructions.
in.LSQ  This value non-zero if the instruction is in LSQ, and therefore a memory instruction.

ea_comp  This is non-zero if operation requires an address computation.

recover_inst  Non-zero if this instruction is at the start of a mis-speculated path.

stack_recover_idx  This is the non-speculative top-of-stack (TOS) for return stack buffer (RSB) prediction.

dir_update  This value reports branch prediction direction update info.

spec_mode  This value is non-zero if this is a speculative instruction.

addr  The address to use for load and store instructions.

tag  RUU slot tag, if this is incremented the instruction is not valid, and therefore squashed.

seq  Instruction sequence number, used to sort the ready list and tag instruction.

ptrace_seq  Used for the pipeline viewing interface.

queued  Is set TRUE when the instruction is put into the ready_queue, and therefore is ready to be issued to a functional unit.

issued  If TRUE, then the operation was issued to a functional unit.

completed  If TRUE, the operation has completed execution.

onames[2]  Logical register names of output operands. These lists are used to limit the number of associative searches into the RUU when instructions complete and need to wake up dependent instructions.

odep_list[2]  This structure links the outputs to the dependent instructions.

idep_ready[3]  These values indicate whether or not the input operands are ready. When these three values are TRUE, the instruction may be issued to a functional unit.
6.1.2 Ready Queue

The `ready_queue` is a structure that keeps track of those instructions which are ready to be issued to a functional unit. Each of its parts is a structure called a `RS_link`. The `RS_link` has a pointer to the next entry in the queue, a reservation station for the instruction it is linking, a tag to determine whether the link is valid or not, and a union of three entries. The entries in the union are `when`, `seq`, and `opnum`. The `when` variable is a time variable that is used in the event queue, as `RS_links` are the backbone of both the `ready_queue` and the `event_queue`. The `seq` variable is the sequence number of the instruction, and the `opnum` is the input or output operand number. It is scanned in the `ruu_issue` function to find instructions to issue.

The `readyq_enqueue` function is used to place instructions into the queue. It keeps instructions in order based upon the sequence numbers. The exception to this is memory, control, and long latency instructions, which are issued to functional units first. There is also an option to make all instructions have equal priority in the `readyq_enqueue` function.

6.1.3 Event Queue

The `event_queue` is a structure that is also made up of `RS_links`, or links to the reservation stations (RUU stations). Unlike the `ready_queue` structure, it is not polled to just see if there are any entries. It is polled to see if any of the entries in the queue are ready to be removed. Entries are placed into the `event_queue` with a specific time at which they expire, or complete. When the `sim_cycle` variable is equal to or greater than an `event_queue` entry's `when` variable, the `ruu_writeback` stage will remove the instruction from the `event_queue`, and update the output-dependent instructions for the entry. The `when` entry in the `event_queue` is determined by the current `sim_cycle` value and the operation latency of the functional unit to which the instruction is linked.

6.2 Simulator Memory Space

The main challenge with using the simulator is the memory space usage. The memory used for each process was described in Section 3.1. A diagram of both the original memory space for sim–outorder (left) and the new memory space for sim–SMT (right) is shown in Figure 6.2. The diagram for
sim-outorder was taken from [28]. Addresses 0x80000000 and above were reserved for future use, but any access to an address 0x80000000 and above causes a segmentation violation.

```
+-----------------------------------+......................+---------------------+
| Unused                            | Unused             |
| 0x00400000                         | 0x00400000         |
| Text (code)                        | Code 1             |
| 0x10000000                         | 0x00500000         |
| Data (.data) (.bss)                | Code 2             |
| 0x7FFFC000                         | 0x00600000         |
| Stack                             | Code 3             |
| 0x7FFFFFFF                          | 0x00700000         |
| Arg & Env                          | Code 4             |
| 0x80000000                         | 0x10000000         |
| Actual Simulator Code & Data       | Data 1             |
|                                   | 0x20000000         |
|                                   | Data 2             |
|                                   | 0x30000000         |
|                                   | Data 3             |
|                                   | 0x40000000         |
|                                   | Data 4             |
|                                   | 0x4FFFFFFF         |
|                                   | Stack 1            |
|                                   | 0x5FFFFFFF         |
|                                   | Stack 2            |
|                                   | 0x6FFFFFFF         |
|                                   | Stack 3            |
|                                   | 0x7FFFFFFF         |
|                                   | Stack 4            |
|                                   | 0x80000000         |
|                                   | Actual Simulator Code & Data |
+-----------------------------------+---------------------+
```

Figure 6.2: Simple Scalar sim-outorder and sim-SMT Memory Space

### 6.2.1 Simulator Loader

When the simulator is started there is no program loaded to run. The procedure `ld_load_prog` reads a program from the binary file, and loads it into the simulator’s memory space. It also sets up the memory protection boundaries:

- **ld_text.base** This is the starting address of the code of the simulated program.

- **ld_text.size** This is size of the code of the simulated program. The upper boundary of the code is given by the formula: `ld_text.base + ld_text.size`.

- **ld_data.base** This is the starting address of the data of the simulated program.
ld_data_size  This is size of the data of the simulated program. The upper boundary of the code is given by the formula: \( ld\_data\_base + ld\_data\_size \).

ld_environ_base  This is a pointer to the environmental variables, as all current environmental variables are pushed onto the program's stack.

ld_stack_base  This is maximum value, and initialization value, for the SP.

ld_prog_entry  This is a pointer to the first address which should be executed. This may or may not be the same as the ld_text_base value.

Before each fetch, the address is checked to make sure that it is between the lower and upper text boundaries. In addition, all memory accesses are checked to make sure that they are instruction reads between the text boundaries, or a data access between the boundaries of the data section.

The compiler generates a binary file in an a.out format. The first section in a binary file is a header which tells the loader whether the file is binary file is a big endian or little endian file. In addition, there is information like the text and data start and sizes. Each of the sections (.text, .data, etc.) gives an address to which it is compiled. \( \text{Ld}\_\text{Load\_prog} \) then moves the section's contents to that location in the simulator's memory space.

6.2.2 Memory Space Notes

The memory space has been the biggest obstacle to having a multithreaded simulator. The problem is that the simulator has one memory space, from 0x00000000 to 0xFFFFFFFF. Part of this space must be used for the simulator's code and data sections. As it turns out, only addresses below 0x7FFFFFFF would be accessible, and would not cause segmentation violations. The sizes of the programs that have been run are not very large. In fact, the programs were designed to fit inside of the limited simulator memory space. The one concern is that the stack pointer of the first thread, and the data of the fourth thread may interact, as the start of the data section for the fourth thread is located at 0x40000000, and the stack pointer for the first thread is initialized to 0x4FFFFFFF. This should be enough room, but the programs had to be profiled with sim-outorder to ensure that they fit in that space.
Another stumbling block was the compiler's limitations. The compiler supplied with the Simple Scalar toolset can use the linker flags, -Ttext \textit{addr}, -Tdata \textit{addr}, and -Tbss \textit{addr} to relocate the different sections of code. If the linker flags were not used, then all of the programs would be compiled to the same starting address. In a single-context simulator that is not a problem. In sim-SMT, however, the loader routine must relocate the sections by hand. The reason for this is that the compiler will not relocate the .rdata section. The linker flags can be used to move all of the other sections of the program to different locations. The .rdata section would not relocate. It was not determined whether that was a bug or a "feature" of the gcc compiler used in the Simple Scalar toolset. The amount that the sections have to be moved are predetermined, and recorded for use in the simulation.

The movement of the code and data to new addresses is not enough alone. For instance, when a program goes to load a piece of data from memory, it will be looking for the address that the compiler generated. By keeping track of the amount by which the sections were relocated, the memory and branch instructions can have an offset added to them to give the correct addresses.

6.2.3 Compiling A Program

Both C programs and assembler programs, written the Simple Scalar ISA can be run by the simulator. Both have to be run through the compiler to generate the binary file to be executed. To compile a C or assembly language program, the following command line should be executed:

```
/home/marc/SS/bin/sslittle-na-sstrix-gcc -Xlinker -Ttext
-Xlinker <addr> -o <program_file> <file>
```

\textit{Program} is the name of the binary file to be created. \textit{Addr} is the location to where the text section should be relocated by the linker. For thread 0, the two linker flags (those arguments proceeded by "-Xlinker") can be omitted. For thread 1, \textit{addr} should be 0x00500000. For thread 2, \textit{addr} should be 0x00600000. For thread 3, \textit{addr} should be 0x00700000. \textit{File} should be the name of the file that is to be compiled.
6.3 Exiting the Simulator

When a program is running on the simulator, unless an infinite loop is created, it will eventually finish. At this point, the compiler inserts an `exit` system call into the program text. In sim-outorder, the exit system call would execute a `longjmp` back to the main-SMT.c main function, which would then proceed to uninitialize the simulator, and print out the statistics. Sim-SMT proceeds differently, as the simulator should only exit after the last thread has completed running. To do this, the boolean array `thread_in_use` was used. If `thread_in_use` is true for a particular thread, then the thread is active, and may have instructions fetched. When the system call `exit` is encountered, then it will set `thread_in_use` to FALSE, and if there are no other threads active it will exit in the same manner as sim-outorder. If there are other threads active, then this thread exits, and the others continue as normal.

6.4 Simple Scalar Instruction Set

The Simple Scalar instruction set is a MIPS/DLX instruction set with additional addressing modes. It doesn’t have the delay slots used for reducing branch penalties like the MIPS/DLX ISA. It is a RISC-like instruction set, but the added number of memory load/store addressing modes makes it more like a CISC ISA. The instructions are listed in Appendix A.

![Simple Scalar Instruction Format](image)

The instructions are 64 bits wide. This allows for instruction set experiments without having to worry about fitting it into 32 bits. A diagram of the Simple Scalar instruction format is shown in Figure 6.3. The upper 16 bits are used to annotate, which will effectively allow for new instructions without having to change the compiler or assembler. In addition, the registers are specified in an
8-bit wide field, and the instruction supports four registers. This allows for easier decoding, and expanding the existing register file to a larger size.

### 6.4.1 Simple Scalar System Calls

SimpleScalar also provided a proxy used to simulate the syscalls of an Ultrix operating system. There are approximately 75 system calls used in the SimpleScalar toolset. In order to use the system calls, an instruction SYSCALL is inserted into the program by the compiler/assembler. When the simulator encounters this instruction, it will call the subroutine ss_syscall. This subroutine is a large case statement which handles all of the possible system calls. The system calls are executed by the host. System calls return a value when they are called to indicate success or failure. These values are returned to the simulator through registers. In general, register 2 is used to get the value from the system call executed on the host, and register 7 indicates to the simulator whether or not the system call succeeded. A sample of the simulator code that shows the error condition reporting is shown below.

```c
/* check for an error condition */
if (regs_R[thread_counter][2] != -1)
    regs_R[thread_counter][7] = 0;
else
    /* got an error, return details */
    regs_R[thread_counter][2] = errno;
    regs_R[thread_counter][7] = 1;
```

One current limitation of the simulator is that the `fork`, or `vfork` system call is not supported. Implementing that system call is important if kernel code is to be examined. That system call would allow for a more dynamic environment in the simulator. If that system call is implemented, then how the new thread would be handled, must be investigated.

### 6.5 Instruction Flow in Simulator

In order to get a better understanding of how the simulator works, an instruction is followed as it goes through the pipeline. An instruction's life in the simulator begins when it is fetched via the PC to the `fetch_data` queue. At that point, the actual and predicted PCs are recorded for future
use. The fetching continues until the instruction fetch queue is filled, or a branch is found and there is no predicted PC value for it. The top two threads will be fetched from, if possible (because of ruu_fetch_issue_delay variable for a threads).

In the next cycle, the instruction will be in the dispatch stage. The decode stage is not performed, as the instructions are pre–decoded. If the thread’s RUU is not full, and the instruction is selected for dispatch, then the instruction will be placed into a RUU station, which is similar to a reservation station. As Kawak et. al. [10] state, the RUU is like a reorder buffer and reservation stations combined into one structure. If the instruction is a memory instruction, it is also placed in the LSQ, which is the load and store queue. It holds and tracks a small window of memory instructions in the pipeline. In the simulator, the actual instruction’s execution takes place here. The results are not available, however, until the writeback stage (ruu_writeback), which occurs in at least three more cycles (issue, execution, writeback).

In the third cycle, the instruction will be in the issue stage. If the instruction’s input operands are not dependent on a non–completed instruction, and the instruction has reached the front of the RUU dispatch queue (tracked by the RUU_head variable), the instruction may be issued to a functional unit if two more conditions are met. First, there must be a functional unit available that matches this instruction’s requirements (integer ALU, Floating–Point ALU, etc.), and second, the instruction’s thread’s ready_queue must be selected to issue an instruction. Once the instruction is issued, the instruction’s issue flag will be set to true. In the simulator, the instruction is now executing, although the result was already created in the dispatch stage. In this stage, the event_queue will have the event of this instruction completing added to it, so the simulator knows when the results of this instruction are valid, so it can be retired.

In the fourth cycle, the instruction will be in the execution stage. The instruction remains in the execution stage, and therefore the functional unit is busy until the instruction completes. Functional units may be made fully pipelined by making the issue latency one for the unit. That states that every cycle, a new instruction may be issued to the functional unit.

In the cycle after the functional unit finishes, the instruction will be in the writeback stage. The results of the instruction are now valid, and all instructions that were dependent upon this
result are informed that the instruction has completed. If this instruction was a conditional control instruction, then at this point, whether or not this was a mis-predicted path is also indicated to the simulator. If this was a mis-predicted path, the tracer.recover subroutine begins to correct the execution. Additionally, the branch prediction unit will be updated to reflect the incorrect decision. The fetch unit will be stalled to indicate a branch penalty.

In the next cycle, the instruction will be in the commit stage. If the instruction was a memory instruction then its execution completes at this point. If the instruction is completed it will be removed from the RUU, and LSQ if it is a memory instruction. At this point the instruction’s lifetime is over in the simulator, and the space that it took in the RUU/LSQ is freed for a future instruction to use.

6.6 Approximations, Limitations, and Simulator Tricks

The sim-SMT simulator takes on an added time complexity from having to search through not one queue during each cycle, but NTHREADS queues. NTHREADS is the #define variable that determines the number of threads that the simulator supports. For our purposes, its maximum value is 4, as the memory space is too limited to go above this value. In order to fight the time complexity that is added, the thread.in_use variable is used as a shortcut. If the thread.in_use variable is FALSE, then many of the operations in the pipeline are skipped. The ruu_fetch will not fetch for threads without the thread.in_use variable set to TRUE.

The branch prediction is divided into different structures for all of the threads. The size on the individual branch prediction structures will be decreased. Another solution, which is more involved, would be to add a field in the branch prediction structure to keep track to which thread the entry belonged. These are two design tradeoffs that were discussed in Chapter 4. However, the individual branch prediction structures was chosen, as it was easier to create, and the bandwidth on the branch prediction unit was enforced.

There was a problem with the cache, which is shared amongst the different threads. At first, neither of the programs was executing, until it was noticed that the starting addresses were mapped to the same cache block. Since the cache was direct mapped, the two continued to thrash, and
the first instructions from each of the threads were never executed. The solution was to make the cache set-associative. The two alternatives are the same as the alternatives for the branch prediction structure. That is either replicate the caches in an array-based fashion, or to add a tag which indicates the thread to which the cache entry belongs.

Another approximation that occurs is the use of a unified cache. The cache handles the instruction and data memory, for all of the threads. In addition, the cache access bandwidth is not tracked, so the memory statistics may be skewed. To create a more realistic simulator, there should at least be two caches for instructions and data.

The simulator does not support the precise exception model. This is a limitation that will have to be fixed if kernel code is to be run on this simulator. The actual register is changed at the time of execution (ruu_dispatch), and some instructions cannot be undone. When exceptions occur, or a timer interrupt for starting a rescheduling process is triggered, the precise state of the processor must be intact to recover from the exception at a later time. Kawak et. al. [10] investigated a way to make this simulator run a precise exception model. They also introduced a timer interrupt setup, so that the kernel could be run. Their work involved creating a pthreads-safe simulator.

The manner in which the system calls are handled are not very accurate. The system call is handled by the host machine. Therefore, the number of instructions that are executed by a particular system call are not taken into account in the final statistics. This could be modified if the system calls were written for this simulator. The operating system Xinu [2] is an educational operating system which provides all of its source code. If those system calls could be integrated into the simulator, that would allow for a closer count on total instructions. Also, it would allow for the simulator to identify the total percentage of time that a thread uses to run kernel code. That would help identify which system calls are vital for the highest system performance.

6.7 Running The Tools

The simulator uses a configuration file in order to set up the architecture. This configuration file is called SMT.cfg and is included in Appendix B. The following command line is used to run the cross-compiled executable “prog”:
/home/marc/SS/simplesim-2.0/sim-SMT -tc:num_progs 1 \
-config config/SMT.cfg <prog>

The simulator also uses the -tc:num_progs 1 flag to indicate that there will be one program
to run. Up to four programs could be run at any one time. To run four programs, the following
command line would be used:

/home/marc/SS/simplesim-2.0/sim-SMT -tc:num_progs 4 \
-config config/SMT.cfg <prog1> <prog2> <prog3> <prog4>

The simulator will print out simulator information (options), run the program, print out the
program's output to STDOUT, and then print out the simulation's statistics. It was from these
statistics that the results in Chapter 7 were derived.

In order to set up the simulations, gather the results, and compare the different runs in an
efficient manner, a set of perl scripts were developed. These scripts were not necessary, but did
automate the results retrieval process, and therefore would be useful. The test_gen.pl script will
take in a list of test names and generate a file that can be parsed by the run_reg.pl script. The
run_reg.pl script takes in the following parameters:

-mt “number” Maximum number of programs to run at one time.

-res_dir “dir” Directory to put the results into.

-of_pre “string” Prefix put on front of result filenames.

-cfg “file” Configuration file to use.

-tl “file” Test list file to use (pre-generated test cases from the test_gen.pl script).

-rr_issue Use the alternate run_issue function.

It will then print out (to standard out) a shell script which can then be run. The test list file
should be generated initially by test_gen.pl, but edited down to the cases that are desired. The
reason for this is that test_gen.pl creates all permutations of the test names from one to four programs.

The shell script can be run, and once those test results are generated, the strip_stats.pl script produce a side–by–side comparison of the different runs’ statistics. To run the strip_stats.pl script on four sim–SMT–generated files, the following command line should be used (where “outfile” is the file to save the comparison to):

```
print "'/home/marc/bin/strip_stats.pl <stat sum. file> \ 
-no_res -s <file1> -s <file2> -s <file3> -s <file4>
```

The results that were found with the different configurations are discussed in Chapter 7.
Chapter 7

Simulation Results

The goal of the simulator is to provide insight to the performance of this architecture. More specifically, measurements of the overall system performance, through the CPI and IPC metrics, the issue efficiency, and the functional unit utilization efficiency results were focused upon. In addition, the round-robin, I-count feedback, and in-order issuing techniques were investigated.

While the cache and branch prediction results are important to the overall performance of the architecture, the main focus of the analysis lies in the overall system performance (through the IPC metric), the efficiency of the issue function and the functional unit utilization. The sizes of both the cache and the branch prediction table were the same for the single thread, but the cache was made four-way set associative for sim-SMT. That was the only difference between the sim-outorder configuration and the sim-SMT configuration. All of the single thread performance measurements were made using only one thread of the sim-SMT simulator. Two threads represented running two copies of a program, and so on. The overall system simulations represented one copy of each program on the different threads of the simulator.

The test programs that were run did not put great stress on the memory hierarchy, as the main concern was making the programs fit into the limited simulator memory space. Their limitations are described in Section 7.1.1. Figure 7.1 shows the manner in which the IPC metric increased as the number of threads running a copy of the individual test programs (discussed in Section 7.1) increased.
Since the turnaround in getting the results is important in a simulator, a time complexity comparison is shown in Section 7.2. Looking at the individual program's performance versus a system simulation will be investigated in Section 7.2.1. Section 7.5 describes the functional unit utilization rate, and the performance of the system with alternate functional unit configurations. Comparing different issue techniques and their respective issue rates/performance is presented in Section 7.3. The wasted issue slots will be discussed in Section 7.4.

7.1 Test Programs

A set of four test programs were developed in order to gather results. The programs were chosen/developed for two reasons. One, they fit in the limited memory space available because of the four threads, and two, to make sure that a mix of functional units were used in the simulation. Additionally, they represent approximately equal number of instructions. Each of the programs have around 200000 instructions in them, except the matsolve.c, which has about 300000 instructions. These programs used in the simulation analysis are:

my-test.c This program is a loop that continues until the product of two increasing numbers
reaches a certain value. The values are output at each step via the printf function. This
program performs a large number of integer operations.

**matsolve.c** This program represents a matrix solver via LU decomposition and backwards sub-
stitution. It performs this function four times. This is the largest program in terms of total
instructions and time complexity. This program, while containing a large number of floating-
point operations, is generally well rounded with both integer and branch instructions.

**newton.c** This program runs a newton interpolation on a number of different functions. This
program is made up of a large number of floating-point operations, in addition to branch
instructions.

**fp-test.c** This program is similar to the my-test.c program, except that the values are float
variables. The values are output, as integers, at each step via the printf function. The reason
that they are not outputted as floating point numbers, is the stack space limitation. This
program is made up of mostly floating-point and branch instructions.

**7.1.1 Applicability of Results**

While the results of the test programs show a proof of concept, a system with the SMT architec-
ture would not expect to meet these speed ups. There reason for this is that the test programs are
small, in order to fit into a particular size. They are meant portray a diversified group of instruc-
tions. Many system benchmarks are saturated with a particular type of instructions, whether it be floating-point, or branch instructions. These test programs have not been profiled to see what
percentages of the instructions belong to the different classes. Also, there are many combinations
to run four programs on four threads. Each of these will produce different results, as thread 0
has the highest priority at the beginning of the simulation. Therefore, the first thread will begin
executing first, followed by the second thread, and so on. The results that are used for the system
results (four different programs on the four simulator threads) are usually the best and worst cases.
While not completely accurate, it does give the general picture of the performance.

In addition, the cache and branch prediction sizes are vital to the performance of the processor.
In each simulation, the performance of the parts is modeled, but not necessarily over-utilized to see how the system performed. There were cache and branch prediction misses, and they were modeled accurately. The frequency at which they occur is not particularly representative of an overall system.

7.2 Time Complexity of Simulator

The performance of a simulator as a program is very important. The idea behind a simulator is to leverage the quicker development cycle of software in order to reduce the development time of hardware. In order to do this, the simulator itself must produce its results in a timely manner. The nature of a multithreaded architecture produces a longer simulation, as there is more than one program running. The sim-SMT simulator has a maximum of four threads running at any time. If a particular thread isn’t in use, some of the execution time is eliminated by using the thread Jn_use variable.

The matrix solver program is the most complex, and time-consuming program in the suite of programs. The sim-outorder simulator adds between two and four seconds of execution time to
one any of the programs. Figure 7.2 shows the time that it takes to run a particular program for one, two, and four threads. The figure shows that the matrix solving program takes the longest amount of time. Also, while the other programs increase at a linear rate with an increasing number of threads, the matrix solving program's simulation time increases more rapidly.

7.2.1 Individual Thread Performance versus System Performance

The overall system performance was a key issue for the SMT architecture. In Figure 7.3, the IPC metrics of all of the individual programs are compared against the IPC for a "system" simulation that runs a copy of all of the programs in the test set. All of the IPC values for the individual programs are nearly identical, and are between 1.42 and 1.45. To get a worst case speedup, the maximum individual program IPC (1.4451 for the matsolve test program), and the worst case IPC from the system simulation are used. The worst case IPC is 2.0786, which is generated from the my-test, fp-test, newton, and matsolve combination. The system combinations are generated by ordering the different programs on different threads in the simulator. For example, the arrangement above puts the my-test program on thread 0, fp-test on thread 1, newton on thread 2, and matsolve on thread 3. This worst case with SMT still produces a speedup of 43.6 percent. The best case
occurs with a comparison between the my-test IPC (1.4205) and the fp-test, newton, matsolve, my-test combination (2.2494). This case results in a 58.4 percent increase in performance.

7.3 Issue Methods and Rates

![Graph showing issue rate versus number of threads](image)

Figure 7.4: Issue Rate versus Number of Threads

The instruction issue method has a major impact on the overall system performance. The issue rate (which is the percentage of total issue slots that are used) for the different tests, are examined in Figure 7.4. As the figure shows, the greatest decrease in waste occurs when jumping from a single copy of the program to two copies. The my-test program has the greatest increase in the issue rate, from one to four threads, with 49 percent. All of the other programs respond with increases in the issue rate of 43 to 48 percent.

The line at the top represents the maximum issue rate for a system simulation (74 percent wasted cycles). The minimum system issue rate was approximately 68 percent. The difference between the individual thread's results at four copies, and the system issue rate indicates that the more heterogeneous nature of the instructions in the system simulation lead to less resource conflicts. This can also be shown in Figure 7.9, which shows the number of times that no functional unit was available for the individual programs and the system simulation.
7.3.1 Issue Bandwidth

A second comparison was made to determine how related the IPC was to the issue bandwidth. Figure 7.5 shows the results of this experiment. The issue rate, as expected, decreased as the issue bandwidth increased. The fetch rate remained the same. The processor was unable to find additional instructions to fill in the added issue slots. With 8 issue slots, the issue rate remained at about 50 percent, and the IPC reached 2.563. When 16 slots were used, the IPC only increased by 0.03 percent, and the issue rate dropped off even further.

7.3.2 Alternate Issuing Schemes

<table>
<thead>
<tr>
<th>Configuration</th>
<th>IPC</th>
<th>Δ</th>
<th>Issue_Rate</th>
<th>Δ</th>
<th>FU_Util_Rate</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-Count Feedback</td>
<td>2.2494</td>
<td>-</td>
<td>0.7392</td>
<td>-</td>
<td>0.2588</td>
<td>-</td>
</tr>
<tr>
<td>Round-Robin Issue</td>
<td>2.1778</td>
<td>-3.2%</td>
<td>0.7053</td>
<td>-4.6%</td>
<td>0.2473</td>
<td>-4.4%</td>
</tr>
<tr>
<td>In-order Execution</td>
<td>0.8906</td>
<td>-60.4%</td>
<td>0.2814</td>
<td>-62%</td>
<td>0.099</td>
<td>-61.7%</td>
</tr>
</tbody>
</table>

Table 7.1: Alternate Issuing Technique Results
The experiment discussed in Section 7.3 was augmented with an investigation of two other issuing methods, round-robin issuing, and in-order execution. The benefit of the in-order execution is the simplification of the processor's control unit. The results of the in-order execution were not very good. The in-order execution was created with the \texttt{-issue:in-order} flag in sim-SMT. It waits until the previous instruction has completed execution. The in-order simulations don't take an ILP into consideration, only strict program order.

The round-robin issuing mechanism may have similar complexity in hardware as the I-count (which is discussed in Section 4.4.1) feedback (which is used in the base results). It will go to all of the threads, in order of priority, and select the first instruction to issue. It will continue to do this until there are no instructions to issue, or the bandwidth has been filled. It produced results that were slightly lower than the I-count technique (five percent or less). Overall, if the complexity is comparable, the I-count method would have the edge over the round-robin issuing scheme.

\section*{7.4 Reduction of Wasted Issue Slots}

The two types of wasted issue slots discussed in Section 4.2 are horizontal and vertical waste. With this simultaneous multithreaded architecture issuing from up to four threads at any time, this waste should be reduced. For the results below, the issuing bandwidth was kept constant, as the number of threads was increased from one to four. The issue bandwidth of the processor was four slots, as shown in the configuration file variable \texttt{-config} in Appendix B.
7.4.1 Reduction of Horizontal Waste

When the processor fills some of its issue slots, but doesn’t fill all of the slots, that is considered horizontal waste [3]. This waste is created by the inherent ILP in any of the programs. Figure 7.6 shows how each of the program’s issuing patterns will reduce the amount of horizontal waste. The experiment is similar to the example shown in Section 4.2. As the figure shows, the greatest decrease in waste occurs when jumping from a single copy of the program to two copies. The matrix-solving program (matsolve) has the greatest decrease in the horizontal waste rate, from one to four threads, with 40 percent. All of the other programs respond with approximately 35 to 37 percent decreases in cycles with horizontal waste.

The line at the bottom represents the minimum horizontal waste rate for a system simulation (37 percent wasted cycles). The maximum system horizontal waste rate was approximately 46 percent. The fact that the individual programs exhibit almost the same performance as the system simulation shows that, for four threads, the SMT architecture can do a good job of mixing the threads to utilize all of the issue slots.

Figure 7.6: Horizontal Waste Rate versus Number of Threads
7.4.2 Vertical Waste

When the processor is unable to fill any of its issue slots, that is considered vertical waste [3]. This waste is created by branch mispredictions, I-cache misses, and longer latency instructions. Figure 7.7 shows how each of the program's issuing patterns will reduce the amount of vertical waste. The experiment is similar to the example shown in Section 4.2. Similar to Figure 7.6, the greatest decrease in waste occurs when jumping from a single copy of the program to two copies. The matrix–solving program (matsolve) has the greatest decrease in the vertical waste rate, from one to four threads, with 53 percent. All of the other programs respond with approximately 46 to 49 percent decreases in completely wasted issue cycles (vertical waste).

The line at the bottom represents the minimum vertical waste rate for a system simulation (15 percent wasted cycles). The maximum system vertical waste rate was approximately 19 percent. These results show SMT architecture does a good job of decreasing the vertical waste by finding an instruction to issue from alternate threads. When comparing the highest single thread program's vertical waste (38 percent) to the lowest system performance rate (15 percent), the simulation exhibits a 61 percent reduction in wasted cycles.
7.5

Functional Unit Utilization

0.28 – 0.26 = c

cos pe

B

0.24 – 0.22

0.18

0.16 –

my-test

matsolve

--+--

newton -Q-

fp-test

x-

system'

Number of Threads

Figure 7.8: Functional Unit Utilization versus Number of Threads

One of the boasts of Tullsen et. al. [3] was that the SMT architecture will better utilize the same number of functional units as the typical out-of-order microprocessor. In the simulation a statistic was dedicated to tracking the percentage of time that all of the functional units were being used. This translated in the functional unit utilization rate. Figure 7.8 shows both the individual program's functional unit utilization response to increasing the number of copies of the program that is being run, and the maximum overall system functional unit utilization. The minimum system functional unit utilization rate was 23.9 percent, while the maximum was 25.9 percent. The trends of the graph show that all of the programs exhibit the greatest increase in functional unit utilization when increasing from one to two threads. The my-test program exhibited the largest overall (one to four threads) increase of 49 percent. The other programs produced increases in utilization of 43 to 48 percent. This is important as the functional units are wasted transistors if they lie idle.
While the functional unit utilization will increase, as shown in Figure 7.8, there could be a problem if they are saturated. At that point, their availability could become a bottleneck in an SMT processor. Figure 7.9 indicates how saturated the functional units become as the number of threads is increased for all of the test programs. In addition, the maximum and minimum number of times no functional unit was available in the system simulations is shown on the graph.

The number of times that there isn’t a functional unit increases dramatically in all of the test programs. For instance, the my-test program exhibits an increase in unavailable functional units of 2670 percent. A single copy of the program shows a moderate number (1500) of unavailable functional units, whereas four copies shows a very large number (40000 times). The other programs exhibits similar patterns and increases (from one to four threads) between 1000 and 2600 percent. This statistic lead to the following investigation of alternate functional unit configurations.

After investigating the number of times that functional units weren’t available, it was decided that investigating alternate functional unit configurations would be appropriate. This could reveal a higher overall system performance without losing much of the functional unit utilization rate that was achieved with the base results.
Table 7.2: Alternate Functional Unit Configuration Results

Table 7.2 shows the different configurations that were attempted on a system simulation. It also shows how they compared to the base case (12 functional units) with respect to overall system IPC, percentage of issue slots used (issue rate), functional unit utilization, and the number of times that no functional unit was available. Increasing the number of integer ALUs, FP ALUs, and FP multiplier/divider only did not significantly improve performance, but the increase in the number of functional units caused a decrease in the functional unit utilization rate.

An additional memory port gives a small increase in performance (IPC) and issue rate, and decreases the number of times that a functional unit wasn’t available considerably. It also produces a smaller decrease in functional unit utilization. Adding a memory port will, however, cause greater strain to the overall memory hierarchy, and therefore, produce more cache misses. The cache miss is a greater penalty than waiting for any of the functional units to complete.

Adding an integer multiplier/divider produces a moderate increase in performance (2.8%) and issue rate (2.8%), and a much smaller number of times that a functional unit wasn’t available–74 percent decrease. The functional unit utilization suffers the smallest hit, with a decrease of only five percent, which still leaves it much higher than any of the individual test program’s functional unit utilization.

Another interesting aspect is shown, as that the issue rate and the IPC are changed identically. That would tend to show that the issue rate is still a major bottleneck in this system, despite simultaneous issuing. Based on the results of Table 7.2, the designer should increase the number of integer multiplier/dividers, and attempt to increase the issue bandwidth from four slots to eight slots.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>IPC</th>
<th>∆</th>
<th>FU Util</th>
<th>∆</th>
<th>Issue_Rate</th>
<th>∆</th>
<th>No FU</th>
<th>∆</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>2.249</td>
<td>-</td>
<td>0.2588</td>
<td>-</td>
<td>0.7392</td>
<td>-</td>
<td>33027</td>
<td>-</td>
</tr>
<tr>
<td>+1 INT ALU</td>
<td>2.249</td>
<td>-</td>
<td>0.2389</td>
<td>-7.7%</td>
<td>0.7392</td>
<td>-</td>
<td>33027</td>
<td>-</td>
</tr>
<tr>
<td>+1 INT mult/div</td>
<td>2.313</td>
<td>+2.8%</td>
<td>0.2457</td>
<td>-5.1%</td>
<td>0.7601</td>
<td>+2.8%</td>
<td>8578</td>
<td>-74%</td>
</tr>
<tr>
<td>+1 MEM port</td>
<td>2.255</td>
<td>+0.3%</td>
<td>0.2397</td>
<td>-7.4%</td>
<td>0.7415</td>
<td>+0.3%</td>
<td>25914</td>
<td>-53%</td>
</tr>
<tr>
<td>+1 FP ALU</td>
<td>2.249</td>
<td>-</td>
<td>0.2389</td>
<td>-7.7%</td>
<td>0.7392</td>
<td>-</td>
<td>33027</td>
<td>-</td>
</tr>
<tr>
<td>+1 FP mult/div</td>
<td>2.249</td>
<td>-</td>
<td>0.2389</td>
<td>-7.7%</td>
<td>0.7392</td>
<td>-</td>
<td>33018</td>
<td>-0.2%</td>
</tr>
</tbody>
</table>

93
Chapter 8

Conclusions and Recommendations

Several enhancements have been made to the traditional general purpose load–store computer architecture. Among the enhancements are memory hierarchy improvements, branch prediction, and multiple issue processors. The simultaneous multithreaded architecture is an extension of the single-threaded architecture that helps hide the performance penalty created by long-latency instructions, branch mispredictions, and memory accesses. The goal of this project was to design, implement, and analyze a model of a simultaneous multithreaded architecture, by modifying a version of the Simple Scalar toolset.

This study focused on the overall system performance improvement possible with a simultaneous multithreaded architecture. One of the main advantages of the simultaneous multithreading architecture is a higher resource utilization. In early simulation results performed with the same number of functional units, an improvement in the number of instructions per cycle (IPC) of between 43% and 58% was found using four threads versus a single thread. Additionally, the issue rate, which is a measurement of the use of instruction issue slots, was found to also increase by between 43 and 58 percent. These results are derived from a set of four sample programs. The horizontal waste rate, which measures the number of unused issue slots, was reduced between 35 and 46 percent. The vertical waste rate, which measures the percentage of unused issue cycles (no issue slots used in a cycle), was reduced between 46 and 61 percent. Different functional unit configurations were simulated, with an additional integer multiplier/divider providing a system performance increase of 2.8 percent and an issue rate increase of 2.8 percent. That result led to
the conclusion that the issue rate was still a major bottleneck in the model. Other metrics such as functional unit utilization and the time complexity of the simulator were addressed.

8.1 Future Work

The simultaneous multithreaded architecture is a very new idea, developed in the last three to five years. It has not been studied as widely as the traditional out-of-order and multithreaded architectures; there are many areas remaining to be investigated.

The results showed that most of the performance increase occurred moving from one to two thread jump. It may therefore be beneficial to create a complete model (potentially in VHDL) of a SMT microprocessor with two threads, so that the other issues that have to be dealt with can be investigated with simpler control logic.

Other topics might include:

- The conversion of the simulator from C to C++ might be useful in making a more robust and flexible simulator with each stage as a separate object. Of course, the design of an object-oriented simulator is not a primary concern of this thesis.

- The design of the out-of-order SMT architecture is very complex. How it can be simplified using in-order execution, and what kind of performance decreases will accompany that design change for a multi-threaded OS kernel environment. Examining an in-order execution architectural model could be beneficial. It is possible that this simpler design can achieve nearly the same performance as an out-of-order execution model.

- In multithreaded real-time applications, the context switch is a crucial element to meeting a deadline. Applying the SMT architecture to such applications and studying its effects would be beneficial.

- Redesigning the simulator for a more flexible memory system. For example, one could investigate separate or unified IL1/DL1 caches for each of the threads, for instance.

- Redesigning the simulator to handle system calls more realistically. In this system, such calls
were handled using a proxy, so the assembly code in the system call is not tracked. Developing a method to track this code would also allow for more OS–kernel related investigations.

- Investigation of how interrupts will be delivered to a processes in an SMT architecture. Also, what implications will SMT have on how OS signals will be delivered.

- Looking into the ALPHA instruction set for the Simple Scalar Toolset. There is a beta version available at the University of Wisconsin. This would allow the investigation of a more RISC–like system, using existing compilers.

- Examining the SMTSIM program (from University of Washington/Tullsen), and how it handles the different issues compared to sim–SMT developed in this thesis.

- Looking into the benefits of different memory models (shared, private, etc.) in conjunction with the SMT architecture.

- Investigating the handling of an increased number of registers. One method for accessing registers is to break them up into banks inside one register file. For instance, if there are four banks, all register numbers with a mod four result of zero, will be placed in bank zero. This would allow four accesses to each of the register files during a single cycle, and reduce the number of registers that have to be searched.
This is the instruction set that is defined in the ss.def file. The instruction set is found in the SimpleScalar toolset hacker's guide [28]. All of the instructions are created through `#defines`.

<table>
<thead>
<tr>
<th>Load/Store Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.d</td>
</tr>
<tr>
<td>s.d</td>
</tr>
<tr>
<td>lb</td>
</tr>
<tr>
<td>sb</td>
</tr>
<tr>
<td>lbu</td>
</tr>
<tr>
<td>sbu</td>
</tr>
<tr>
<td>lh</td>
</tr>
<tr>
<td>sh</td>
</tr>
<tr>
<td>lhu</td>
</tr>
<tr>
<td>shu</td>
</tr>
<tr>
<td>lw</td>
</tr>
<tr>
<td>sw</td>
</tr>
<tr>
<td>dlw</td>
</tr>
<tr>
<td>dsw</td>
</tr>
<tr>
<td>l.s</td>
</tr>
<tr>
<td>s.s</td>
</tr>
</tbody>
</table>

In addition to the load instructions listed here, there are additional addressing modes: `(C) (reg + C) (with pre-increment) (reg + C) (with pre-decrement) (reg + reg) (with pre-increment) (reg + reg) (with pre-decrement)`.
• (reg + reg) (with post-increment)
• (reg + reg) (with post-decrement)

A.2 Integer Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Integer add signed and Logical AND</td>
</tr>
<tr>
<td>addu</td>
<td>Integer add unsigned or Logical OR</td>
</tr>
<tr>
<td>sub</td>
<td>Integer subtract signed nor Logical NOR</td>
</tr>
<tr>
<td>subu</td>
<td>Integer subtract unsigned xor Logical XOR</td>
</tr>
<tr>
<td>mul</td>
<td>Integer multiply signed sll Logical shift left</td>
</tr>
<tr>
<td>mulu</td>
<td>Integer multiply unsigned srl Logical shift right</td>
</tr>
<tr>
<td>div</td>
<td>Integer division signed sra Arithmetic shift right</td>
</tr>
<tr>
<td>divu</td>
<td>Integer division unsigned slt Set less than</td>
</tr>
<tr>
<td></td>
<td>situ Set less than unsigned</td>
</tr>
</tbody>
</table>

A.3 Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>Jump beq Branch if equal to zero</td>
</tr>
<tr>
<td>jal</td>
<td>Jump and link bne Branch if not equal to zero</td>
</tr>
<tr>
<td>jr</td>
<td>Jump register blez Branch if less than or equal to zero</td>
</tr>
<tr>
<td>jalr</td>
<td>Jump and link register bgtz Branch if greater than zero</td>
</tr>
<tr>
<td></td>
<td>bltz Branch if less than zero</td>
</tr>
<tr>
<td></td>
<td>bgez Branch if greater than or equal to zero</td>
</tr>
<tr>
<td></td>
<td>bct Branch FCC register TRUE</td>
</tr>
<tr>
<td></td>
<td>bcf Branch FCC register FALSE</td>
</tr>
</tbody>
</table>

A.4 Floating-Point Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.s</td>
<td>Floating-point single-precision add abs.s Single-precision absolute value</td>
</tr>
<tr>
<td>add.d</td>
<td>Floating-point double-precision add abs.d Double-precision absolute value</td>
</tr>
<tr>
<td>sub.s</td>
<td>Floating-point single-precision subtract neg.s Single-precision negation</td>
</tr>
<tr>
<td>sub.d</td>
<td>Floating-point double-precision subtract neg.d Double-precision negation</td>
</tr>
<tr>
<td>mult.s</td>
<td>Floating-point single-precision multiply sqrt.s Single-precision square root</td>
</tr>
<tr>
<td>mult.d</td>
<td>Floating-point double-precision multiply sqrt.d Double-precision square root</td>
</tr>
<tr>
<td>div.s</td>
<td>Floating-point single-precision division cvt Integer, single, double conversion</td>
</tr>
<tr>
<td>div.d</td>
<td>Floating-point double-precision division c.s Single-precision comparison</td>
</tr>
<tr>
<td></td>
<td>c.d Double-precision comparison</td>
</tr>
</tbody>
</table>
A.5 Miscellaneous Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>No operation</td>
</tr>
<tr>
<td>syscall</td>
<td>System call</td>
</tr>
<tr>
<td>break</td>
<td>Declare program error</td>
</tr>
</tbody>
</table>
Appendix B

Sim–SMT Configuration File

This is the configuration used with the \texttt{-config} flag.

\begin{verbatim}
# # default sim-outorder configuration #

# random number generator seed (0 for timer seed)
-seed 1

# instruction fetch queue size (in insts)
-fetch:ifqsize 4

# extra branch mis-prediction latency
-fetch:mplat 3

# branch predictor type \{nottaken|taken|perfect|bimod|2lev\}
-bpred bimod

# bimodal predictor BTB size
-bpred:bimod 2048

# 2-level predictor config \langle\ell_1\text{size}\rangle \langle\ell_2\text{size}\rangle \langle\text{hist}\_\text{size}\rangle\rangle
-bpred:2lev 1 1024 8

# instruction decode B/W (insts/cycle)
-decode:width 8

# instruction issue B/W (insts/cycle)
-issue:width 4

# run pipeline with in-order issue
-\text{issue:inorder} false

# issue instructions down wrong execution paths
-\text{issue:wrongpath} true

# register update unit (RUU) size
-ruu:size 16
\end{verbatim}
# load/store queue (LSQ) size 
-lsq:size 8

# 11 data cache config, i.e., {<config>|none} 
-cache:d11 d11:128:32:4:1

# 11 data cache hit latency (in cycles) 
-cache:d11lat 1

# 12 data cache config, i.e., {<config>|none} 
-cache:d12 ul2:1024:64:4:1

# 12 data cache hit latency (in cycles) 
-cache:d12lat 6

# 11 inst cache config, i.e., {<config>|dl1|dl2|none} 
# -cache:il1 il1:2048:32:1:1
# -cache:il1 il1:512:32:4:1

# 11 instruction cache hit latency (in cycles) 
-cache:il1lat 1

# 12 instruction cache config, i.e., {<config>|dl2|none} 
-cache:il2 d12

# 12 instruction cache hit latency (in cycles) 
-cache:il2lat 6

# flush caches on system calls 
-cache:flush false

# convert 64-bit inst addresses to 32-bit inst equivalents 
-cache:icompress false

# memory access latency (<first_chunk> <inter_chunk>) 
-mem:lat 18 2

# memory access bus width (in bytes) 
-mem:width 8

# instruction TLB config, i.e., {<config>|none} 
-tlb:itlb itlb:16:4096:4:1

# data TLB config, i.e., {<config>|none} 
-tlb:dtlb dtlb:32:4096:4:1

# inst/data TLB miss latency (in cycles) 
-tlb:lat 30

# total number of integer ALU’s available 
-res:ialu 4

# total number of integer multiplier/dividers available 
-res:imult 1
# total number of memory system ports available (to CPU)
-res:memport 2

# total number of floating point ALU's available
-res:fpalu 4

# total number of floating point multiplier/dividers available
-res:fpmult 1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat false
Appendix C

Test Programs

These are the test programs that were used to gather data.

C.1 MY-TEST.C

/* Test program with minimum stack sizes */
#include <stdio.h>
#define THREAD 3

int main(void)
{
    int i = 0;
    int j = 0;
    int k = 0;

    while(k < 1000)
    {
        printf("Thread %d: i(%d) * j(%d) = %d\n", THREAD, i, j, i*j);
        i++;
        printf("Thread %d: i(%d) * j(%d) = %d\n", THREAD, i, j, i*j);
        j++;
        k = i*j;
    }
    return 0;
}

C.2 MATSOLVE.C

/* Filename: main.C
 * Author: Marc Torrant
 * Description: This is the main function for the
 * equation solving program. It takes in
 * arguments which may or may not include:
method by which to solve it, whether or not to print out the iterations, and whether or not you want to print out the entered matrices.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#define THREAD 3

#if 0
#define ROWS 12
#define COLS 12
#else
#define ROWS 4
#define COLS 4
#endif

#define ARRAY_TYPE double

/* type of individual elements */
ARRAY_TYPE array[ROWS][COLS];
ARRAY_TYPE ansarray[ROWS];
ARRAY_TYPE rightside[ROWS];

/* number of rows (horizontal) in the matrix */
int rows = COLS;

/* number of columns (vertical) in the matrix */
int cols = ROWS;

/* prints the current object in the matrix format of rows and columns */
void print(void)
{
    int i, j;
    for(i = 0; i < rows; ++i)
    {
        printf("Thread %d: matrix: ", THREAD);
        for(j = 0; j < cols; ++j)
            printf("%12.4e ", array[i][j]);
        printf("\n");
    }
}

/* performs lu decomposition on current object. Elements below diagonal * are U matrix; elements on and above diagonal are L matrix */
void lu(void)
{
    int i, j, k;
    int n = rows;
}
for(k = 0; k < n; ++k)
    for(i = k+1; i < n; ++i)
    {
        array[i][k] = array[i][k] / array[k][k];
        for(j = k+1; j < n; ++j)
            array[i][j] -= array[i][k] * array[k][j];
    }

/* performs forward and backward substitution on current object */
/* x: answer matrix */
/* b: solution matrix */
void fbsub(void)
{
    int i, j, k;
    int n = rows;

    /* forward substitution */
    for(k = 0; k < n; ++k){
        for(i = k+1; i < n; ++i){
            rightside[i] -= array[i][k] * rightside[k];
        }
    }

    /* back substitution */
    #if 1
        ansarray[n-1] = rightside[n-1] / array[n-1][n-1];
    #else
        ansarray[ROWS-1] = rightside[ROWS-1] / array[ROWS-1][ROWS-1];
    #endif
    for(k = n-2; k>=0; k--)
    {
        ARRAY_TYPE z = rightside[k];
        for(j = k+1; j < n; j++)
            z -= ansarray[j] * array[k][j];
        ansarray[k] = z / array[k][k];
    }
}

void matgen( ARRAY_TYPE k )
{
    int i, j;
    for( i = 0; i < ROWS; ++i )
        for( j = 0; j < COLS; ++j )
        {
            if ( ( j == i - 1 ) || ( j == i + 1 ) )
                array[i][j] = 1.0;
            else if ( j == i )
                array[i][j] = 3.0;
            else
                array[i][j] = 0.0;
        }

    /* Rightside vector */
for( i = 0; i < ROWS; ++i )
    {rightside[i] = k * (ARRAY_TYPE)i; }
#define RUNS 4

int main(void)
{
    int i,j,k,l;
    int size = 0;
    /* Initialize the answer array */
    for( i = 0; i < size; ++i )
        {ansarray[i] = 0.0; }
    /* Find the solution to the matrix using the appropriate method */
    for( j = 0; j < RUNS; j++ )
    {
        matgen(j);
        print();
        lu();
        fbsubO;
        printf("Thread %d: solution: ", THREAD);
        for( i = 0; i < ROWS; ++i )
            printf("%12.4e ", ansarray[i]);
        printf("\n");
    }
    return 0;
}

C.3 NEWTON.C

/* Filename: newton.c
 * Author: Marc Torrant
 * Description: This program performs a newton interpolation */
#include <stdio.h>
#include <math.h>
#include <stdio.h>
define THREAD 3
/* This structure type represents an x-y pair */
typedef struct{
double x;
double y;
} pair;

double func(double x)
{
    return (1/(1+(25*x*x))));

void findpoints(pair *xy, int n)
{
    int i = 0;
    long double h = 2.0/n;
    for( i = 0; i < n; ++i)
    {
        xy[i].x = -1 + (i*h);
        xy[i].y = func(xy[i].x);
    }
}

double interpolate_ddnewton( double x, const pair *xy, int n, const double *a )
{
    double y = a[0];
    double p = 1.0;
    int k=0;
    for( k = 1; k < n; ++k )
    {
        p *= ( x - xy[k-1].x );
        y += a[k] * p;
    }
    return y;
}

/* Description: This is the implementation of the divided differences function. This function takes
 * in an array of known xy pairs (numbered 0 .. n-1), and n: the number of pairs.
 * It outputs the coefficients of an interpolating polynomial (divided differences) based on
 * the array xy.
 * 
 * Throws an exception if there is a duplicate xy.x
 */
void divided_differences( double *a, const pair *xy, int n )
{
    int k, j;
    /* This is the local array */
    double f[n+1];
    f[0] = 0.0;
    for( k = 1; k <= n; ++k )
    {
        f[k] = xy[k-1].y;
        a[0] = f[1];
    }
    for( j = 1; j < n; ++j )
    {
        for( k = 1; k <= ( n-j ); ++k )
        {
            f[k] = ( f[k+1] - f[k] ) / ( xy[k+j-1].x - xy[k-1].x );
            a[j] = f[1];
        }
    }
```c
#define RUNS 13

int main(void)
{
    int m = 1;
    double x = -0.9;
    double ye;
    double new_error = 0.0;
    double ynew = 0.0;

    int i = 0;
    #if 0
        int n = 0;
    #else
        int n = 5;
    #endif

    double a[n];
    pair xy[n];

    for( m = 0; m < RUNS; m++ )
    {
        ye = func( x );
        new_error = 0.0;
        ynew = 0.0;

        for( i = 0; i < n; ++i )
        {
            xy[n].x = 0.0;
            xy[n].y = 0.0;
        }

        for( i = 0; i < n; ++i )
            a[i] = 0.0;

        /* This is the interpolation of newton */
        findpoints( xy, n );
        divided_differences( a, xy, n );
        ynew = interpolate_ddnewton( x, xy, n, a );
        new_error = (fabs(ye - ynew));

        printf("Thread %d: This is the newton interpolation answer: %f\n", 
               THREAD, ynew);
        printf("Thread %d: The error due to newton is: %f\n", 
               THREAD, new_error);
    }

    return 0;
}
```
C.4 FP–TEST.C

#include <stdio.h>

#define THREAD 3

int main(void)
{
    float a = 1.0;
    float y = 1.0;
    float z = 1.0;

    while( a < 1000.0 )
    {
        printf("Thread %d: y(%d) * z(%d) = a(%d)\n", THREAD, (int)y, (int)z, (int)(y*z));
        y++;
        printf("Thread %d: y(%d) * z(%d) = a(%d)\n", THREAD, (int)y, (int)z, (int)(y*z));
        z++;
        a = y*z;
    }
    return 0;
}
Bibliography


