NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A257 662

DTIC
ELECTE
DEC 02 1992
S
A
D

THESIS

A STUDY ON THE EFFECTIVENESS OF
LOCKUP-FREE CACHES FOR A REDUCED
INSTRUCTION SET COMPUTER (RISC) PROCESSOR

by

Leonard Tharpe

September 1992

Thesis Advisor: Dr. Amr Zaky

Approved for public release; distribution is unlimited.

92-30654

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Study on the Effectiveness of Lockup-Free Caches for a Reduced Instruction Set Computer (RISC) Processor

12. PERSONAL AUTHOR(S)
Leonard Tharpe

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 09/91 TO 09/92	14. DATE OF REPORT (Year, Month, Day) September 1992	15. PAGE COUNT 138

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Reduced Instruction Set Computer (RISC); Lockup-Free Cache Interface

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis presents a simulation and analysis of the Reduced Instruction Set Computer (RISC) architecture, and the effects on RISC performance of a lockup-free cache interface. RISC architectures achieve high performance by having a small, but sufficient, instruction set with most instructions executing in one clock cycle. Current RISC performance range from 1.5 to 2.0 CPI. The goal of RISC is to attain a CPI of 1.0. The major hinderance in attaining that goal is attributed to instructions that require main memory access. In this thesis, we attempt to reduce the effects of high penalties for non-cache accesses by using a non-blocking cache memory subsystem called a lockup-free cache. This interface between the cache and main memory prevents the processor from "locking-up" when a request from main memory occurs. This is accomplished by entering all non-cache requests into a memory queue, while the processor continues to issue and execute other instructions. The evaluation of the effects of the lockup-free cache interface is done using different variations of the interface design. The results show that using the lockup-free cache improves RISC performance

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Amr Zaky	22b. TELEPHONE (Include Area Code) (408) 646-2693

22c. OFFICE SYMBOL CS/Za

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete

A STUDY ON THE EFFECTIVENESS
OF LOCKUP-FREE CACHES FOR A REDUCED
INSTRUCTION SET COMPUTER (RISC) PROCESSOR

by
Leonard Tharpe
Captain, United States Army
B.S., Austin Peay State University

Submitted in partial fulfillment of the
requirements for the degree of
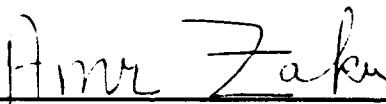
MASTER OF COMPUTER SCIENCE

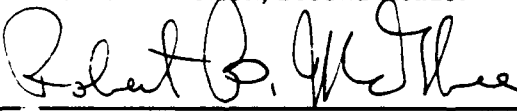from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Author: _____
Leonard Tharpe

Approved By: _____
Dr. Amr Zaky , Thesis Advisor

_____
Dr. Michael L. Nelson, Second Reader

_____
Robert B. McGhee, Chairman,
Department of Computer Science

ii

ABSTRACT

This thesis presents a simulation and analysis of the Reduced Instruction Set Computer (RISC) architecture, and the effects on RISC performance of a lockup-free cache interface. RISC architectures achieve high performance by having a small, but sufficient, instruction set with most instructions executing in one clock cycle. Current RISC performance range from 1.5 to 2.0 CPI. The goal of RISC is to attain a CPI of 1.0. The major hinderance in attaining that goal is attributed to instructions that require main memory access. In this thesis, we attempt to reduce the effects of the high penalties for non-cache accesses by using a non-blocking cache memory subsystem called a lockup-free cache. This interface between the cache and main memory prevents the processor from "locking up" when a request from main memory occurs. This is accomplished by entering all non-cache requests into a memory request queue, while the processor continues to issue and execute other instructions. The evaluation of the effects of the lockup-free cache interface is done using different variations of the interface design. The results show that using the lockup-free cache improves the RISC performance.

TABLE OF CONTENTS

I. INTRODUCTION

A. COMPUTER TRENDS: The RISC Alternative

The Reduced Instruction Set Computer (RISC) architecture is fast emerging as the architecture processor of choice in the computer industry. Since its arrival only a decade ago, numerous implementations and variations of RISC have emerged, and the trend is continuing to shift toward the RISC concept. Industry experts predict that the RISC architecture could capture a major share of the market in the 1990's [Bur90].

Until recently, the ever increasing demands for faster and more powerful computing machinery have been met by the Complex Instruction Set Computer (CISC) architectures. As the name implies, these computers consist of powerful, complex instructions interpreted by microcode residing on a chip which controls the hardware that executes the program [Met90]. The sizes of the machine language instruction sets of CISC architectures become larger as they increase in complexity. The underlying assumption of CISC is that machines that feature many complicated instructions could provide more computing power for its users. Despite the advantages offered by the more complex instructions, the ideal performance of CISC is not achieved because of the overhead resulting from the complexity of the control circuits.

1

RISC takes a radically different approach to improved performance. RISC architecture emphasizes simplicity and efficiency by having a small instruction set [Dei90]. Most RISC architectures are designed so that all instructions would execute in one cycle. This eliminated the more complex instructions that required more than one cycle to execute [SC91]. RISC architectures also avoid complicated instructions requiring microcode support. Instead, these complex capabilities are implemented in software [TT91].

Major characteristics of RISC architectures include a fewer number of instructions, simple load and store operations for register to memory transfers, large register set, deep pipelines, and many levels of memory hierarchy [GM87]. The most significant advantages of RISC include speed and ease of implementation.

B. RISC PERFORMANCE THROUGH MEMORY HIERARCHY

The performance of most computer architectures is often limited by the design of its memory hierarchy. Typically, memory is managed using a three-level memory hierarchy. The first level is high speed cache, which is expensive and of lowest capacity. The second level is real or main memory which is slower and less expensive than cache memory. The third level is the large capacity storage devices such as disks. This level holds programs and data that cannot fit in levels one and two [FM87].

Main memory access delays are a major factor in performance of a program execution. With a typical miss penalty costing between 8 and 32 clock cycles [HP90], the ability to control and minimize access to main memory will have a direct effect on performance. This is particularly critical to the RISC goal of executing one instruction per cycle.

RISC memory systems are usually complex because of the requirement to keep instructions and data supplied to the processors. The RISC memory hierarchy often includes an on-chip instruction buffer to hold the next few instructions. Some memory systems have both an instruction cache and a data cache which may be on or off-chip. The main memory for RISC systems are off-chip and sometimes off the processor board [GM87]. This maximizes the penalty for cache misses or other main memory accesses, making the requirement for highly efficient memory management systems critical.

Improvements in RISC performance can likely be made through improvements in its memory management systems since memory accesses consume a considerable amount of machine cycles. Regardless of the efficiency or hit rate of a cache memory system, *misses* will occur and main memory must be accessed. Main memory access is also required for write/store instructions. Main memory access stalls or blocks the processor for a specified number of cycles while data is fetched and/or written.

A possible solution to reducing the costs of main memory accesses is the concept of a *lock-up free cache interface* [Kro81][SD91]. The lock-up free cache is a non-blocking cache interface that queues main memory access requests (i.e., loads and stores), allowing processing to continue while the memory access queue is being served.

C. OBJECTIVES

1. Primary Objective

The primary objective of this thesis is to analyze the performance of different variations of the RISC architectural concept. Specifically, we examine the RISC architecture and the effects on performance of a memory subsystem known as a lockup-free cache interface. Experiments are made on models of several design possibilities of the lockup-free cache interface.

In accomplishing the primary objective, an intermediate objective is to acquire or develop effective simulation tools to observe the behavior of a RISC implementation as it executes different types of programs. We choose the SPARC as a model of a RISC architecture because SPARC incorporates many characteristics that are typical of RISC architectures, and a trace simulator for it was available.

2. Simulation Tools Objectives

One objective of the simulation tools is to produce executable SPARC binaries for input to a simulator which produces binary address trace files. These address traces are then used for producing instruction count data as shown in Figure 1.1 and for translating the binary address trace into a more readable SPARC assembler language format as shown in Figure 1.2.

```
┌─────────────────────────────────────────────────────────────┐
│                                                             │
│  ┌──────────┐      ┌──────────┐      ┌──────────────┐       │
│  │ Address  │ ───> │Simulation│ ───> │ Instruction  │       │
│  │ Trace    │      │  Tool    │      │ Count Report │       │
│  └──────────┘      └──────────┘      └──────────────┘       │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```
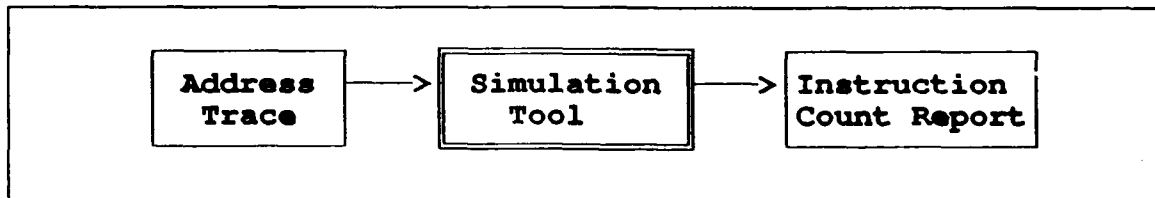
Figure 1.1 Instruction Count Report Generation

```
┌─────────────────────────────────────────────────────────────┐
│                                                             │
│  ┌──────────┐      ┌──────────┐      ┌──────────┐           │
│  │ Address  │ ───> │Simulation│ ───> │ SPARC    │           │
│  │ Trace    │      │  Tool    │      │ Assembler│           │
│  └──────────┘      └──────────┘      │ Language │           │
│                                      └──────────┘           │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```
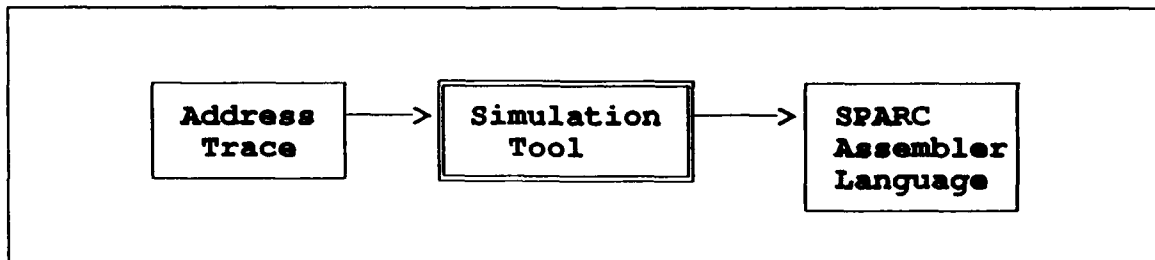
Figure 1.2 Assembly Language Translation

Another objective is to produce specially modified address trace files to use in other simulation tools to observe the RISC architecture under various workloads. They also provide a view for "what-if" analysis as varied architecture configurations are simulated.

5

A final objective of the simulation tool is to provide functions for simulating the performance of a lockup-free cache interface for a RISC processor (Figure 1.3). The functions include simulating a non-blocking cache interface and fetching instructions out of order for execution. These specific techniques will be used to evaluate the effects of a cache interface that minimizes main memory traffic.

```
┌─────────────────────────────────────────────────────────────┐
│  ┌──────────┐      ┌──────────────┐      ┌──────────────┐    │
│  │ Modified │      │ Lockup-Free  │      │ Performance  │    │
│  │ Address  │ ───> │ Cache        │ ───> │ Analysis     │    │
│  │ Trace    │      │ Simulator    │      │              │    │
│  └──────────┘      └──────────────┘      └──────────────┘    │
└─────────────────────────────────────────────────────────────┘
```
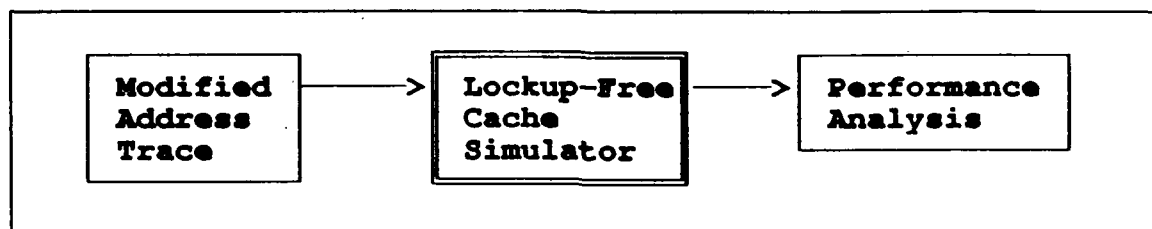
Figure 1.3 Performance Analysis of Alternative Designs

D. ORGANIZATION OF STUDY

The remainder of this thesis is divided into five chapters. In Chapter II background information on RISC, SPARC, and the lockup-free cache is provided. Chapter III presents a model of a lockup-free cache. The simulation tools used to model the lockup-free cache interface and to observe the behavior of the SPARC architecture are discussed in Chapter IV. In Chapter V, we simulate and evaluate alternative design possibilities for the lockup-free cache interface on RISC to improve the system performance. Chapter VI presents our conclusions and further research issues.

II. BACKGROUND

This chapter discusses the origin and characteristics of the RISC architecture and how RISC achieves high levels of performance. We then focus on the SPARC architecture and how it approaches the RISC concept. Finally, the lockup-free cache interface design is introduced as it is modelled in this study to determine its effect on RISC performance.

A. OVERVIEW OF THE REDUCED INSTRUCTION SET COMPUTER

1. General

The Reduced Instruction Set Computer (RISC) was developed as a result of studies in the mid 1970's which suggested that computer architectures consisting of many complex instructions still executed mostly simple instructions. Specifically, an IBM study observed that over two-thirds of the instruction executions on their System 370 architecture accounted for only 10 simple instructions [Dei90]. In 1979 the first RISC machine, the IBM 801, was completed. The IBM 801 also was the first computer to feature single-cycle instruction execution [SPA88].

The RISC architecture is based on the concept that computers with a relatively small number of simple instructions and a large number of registers can operate faster than computers with a large instruction set containing

many complex instructions. Figure 2.1 shows instruction set sizes of several RISC processors [Gro90][GM87]. Although the name Reduced Instruction Set Computer implies reduced instruction sets, there is much more to a RISC architecture than that. The size of the instruction set is merely an end result of the techniques used to improve computer performance. Generally, RISC architectures are designed to exploit the advantages of the latest features of both hardware and software technologies.

RISC PROCESSORS	INSTRUCTIONS
UC Berkeley RISC I	31
Stanford MIPS	32
UC Berkley RISC II	39
Motorola 88100	51
Fujitsu SPARC	67
Cypress SPARC	69
MIPS R3000	78
Intel i860	82
Bipolar Integrated SPARC	88
Pyramid	90

Figure 2.1 RISC Processors Instruction Set Sizes.

2. Characteristics of RISC Architecture

There are several specific characteristics that are typical of RISC architectures that have proven to be the key to enhanced performance. One important characteristic is that all instructions except loads, stores, and floating point instructions can be executed in a single cycle. The single-cycle instruction set design makes it easier for several instructions to be processed at the same time, thus allowing more efficient pipeline operations.

Another characteristic of RISC is its register intensive design. RISC machines have 32 or more general purpose registers, a feature that greatly reduces the number of operand memory references, thus reducing the costs of memory accesses [BEH91]. Generally, all RISC instructions use either two registers or a register and a constant with the result being placed in a destination register. The large number of registers can also be used to reduce the high cost of branch instructions by dedicating registers exclusively for branches [DW90].

RISC is also characterized by its simple fixed-format instructions. All instructions are 32 bits long and the operation codes and addresses are located in the same positions of an instruction. To insure simplicity of the instruction set, RISC uses software designed from simple instructions to execute complex functions. Only those functions that do not degrade performance are implemented in

9

hardware. The simple, fixed-format instruction set is also good for real-time environments because of its speed and ease of execution.

Another characteristic is that RISC designs have a load/store architecture where all operations are performed on operands stored in registers with memory being accessed only by load and store instructions. The load/store architecture also makes it easier for compilers to optimize register allocation [Kan87]. Figure 2.2 summarizes the basic characteristics of RISC and how it differs from the CISC [Met90].

Characteristic	RISC	CISC
Instruction Set	Small (< 100)	Large (> 200)
Instruction Format	All instructions 32 bits long	Variable size instruction
Cycles/Instruction	1	more than one
General Registers	32 or more	16 or less
Memory Addressing	Only load/store instructions	Nearly all instructions

Figure 2.2 How RISC Differs from CISC Architectures.

3. RISC Pipelining

With the goal of achieving an execution rate of one machine cycle per instruction, one technique RISC architectures use is pipelining. The simple fixed instruction

formats make pipelining with RISC architectures very efficient. RISC pipelines are also designed to reduce the cycles lost to conditional branches incorrectly predicted.

One benefit of pipelining is that it provides a way to start a new instruction before a previous one has been completed. Figure 2.3 shows a sequential process being done without the use of pipelining. To process the same task using a five-stage pipeline as shown in Figure 2.4, five different instructions may be processing at a time, and ideally, one instruction is completed every cycle [Ibb90]. Pipelining improves processor speed by reducing the average execution time per instruction throughput.
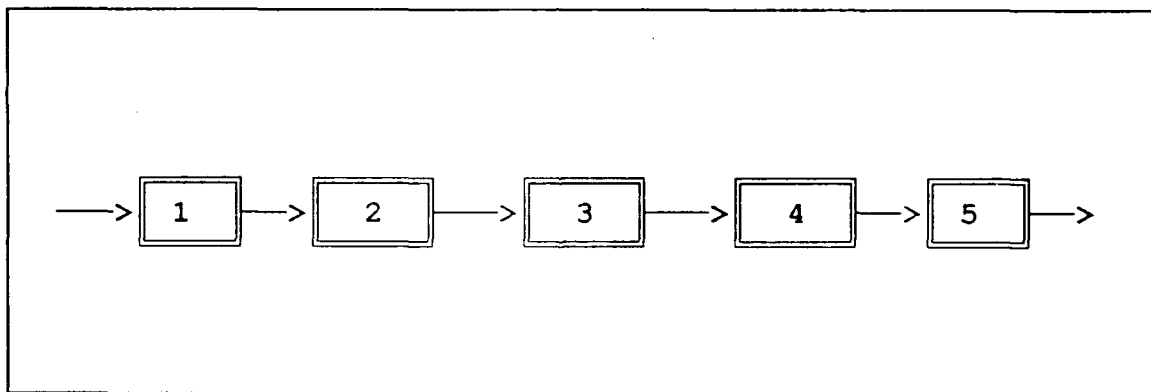
Figure 2.3 A 5-Step Sequential Process.

The RISC I pipeline consisted of only two stages, a fetch and an execute. The fetch stage, which brings the instruction in from memory, took about the same time as the execute stage, which actually performed the calculations and wrote the results back to memory. The RISC II added a third stage, write stage, which wrote the results from a destination

11
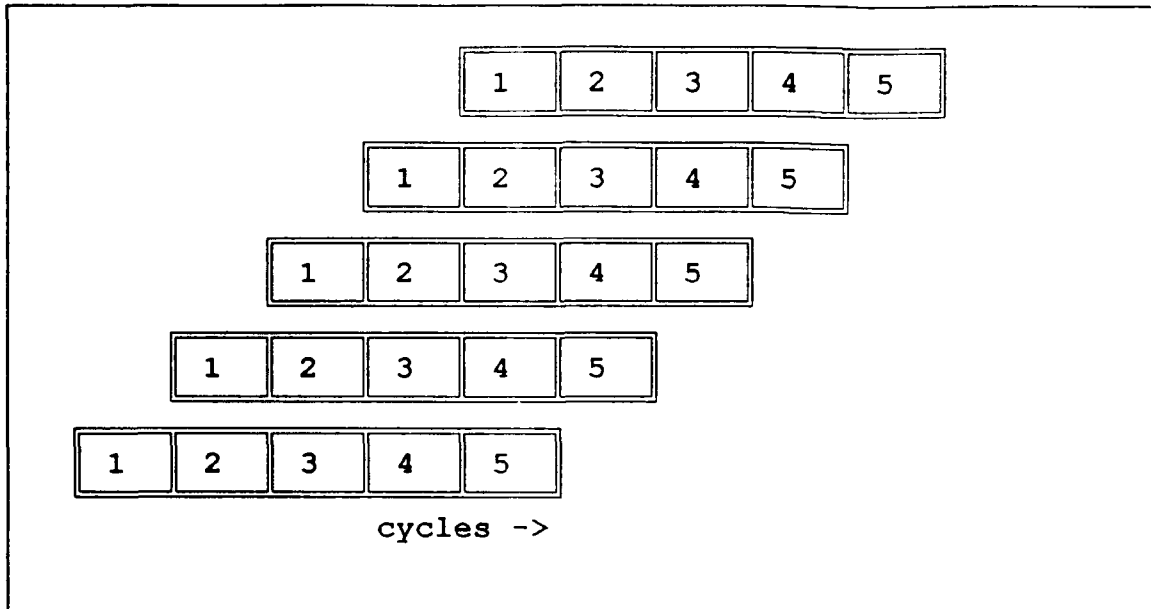
Figure 2.4 Pipelined Execution of a 5-Stage Process.

register to memory at the appropriate time [GM87]. More recent RISC architectures use four or five stage pipelines.

4. Computer Performance and The RISC Approach

a. *Measuring Performance*

Computer performance is measured by the amount of the time required to execute a program. Performance encompasses two types of time, elapsed time and CPU time. Elapsed time is the time required to execute a program from start to finish. It includes latency of input/output activities such as memory and disks accesses, and it includes overhead from the operating system, such as context switching [HP90]. CPU time consists of user CPU time which is the actual time the computer spends in the user program, and system CPU time which is the time the computer spends in the

12

operating system doing some task required by the user program.

The number of clock cycles to execute an instruction (cycles per instruction, CPI) and the number of instructions a computer executes per second (millions of instructions per second, MIPS) are also good indicators of performance. CPI is calculated by knowing the number of clock cycles and the instruction count:

$$CPI = \frac{\text{Clock cycles for a program}}{\text{Instruction count}}$$

From this formula, clock cycles can be defined as CPI * instruction count. MIPS, million instructions per second, can be calculated as such:

$$MIPS = \frac{\text{Instruction count}}{\text{Execution time} * 10^6} = \frac{\text{Clock rate}}{CPI * 10^6}$$

MIPS and CPI values can both be used to calculate program execution time by:

$$\text{program time} = \text{Instruction count} * CPI * \frac{1}{\text{Clock rate}} .$$

Observing the formulas above, improved performance, or reduced program execution time can be achieved by decreasing either the cycle time, the CPI, or the instruction count.

b. The RISC Approach to High Performance

RISC CPI values are typically between 1.5 and 2.0. They achieve this by defining simple instructions and by using

sufficiently large cache memory systems that have low miss rates. Simple instructions imply more efficient pipeline operations. The low-miss-rate caches greatly influence RISC performance, as during a miss the controller must first fetch the instruction or data from main memory. This incurs a significant increase in program execution time because numerous cycles are required to access main memory [TT91].

RISC reduces its instruction count through the use of a large number of registers. Variables, constants, and addresses are placed in registers instead of time-consuming main memory. The use of registers instead of memory for instructions other than loads and stores also reduces the requirement for memory access which could result in a cache miss [AAD90].

The cycle time is dependent mainly on available technology. The design of the cache and pipeline determine whether or not an architecture can achieve the aim of one instruction executed per cycle. RISC's simple, fixed-length instructions allow fast chip-to-cache interfacing. The fixed formats also speeds up decoding and dependency calculations which helps shorten the cycle time [Gar91].

B. SCALABLE PROCESSOR ARCHITECTURE (SPARC)

The Scalable Processor Architecture (SPARC) is a Reduced Instruction Set Computer (RISC) developed by Sun Microsystems in 1987 [SPA88]. The SPARC architecture is based on the design

14

of the Berkeley RISC-II implementation [HP90]. The main features of SPARC, like most other RISCs, include a small, simple instruction set which directly enhances its performance. SPARC is an open architecture with published design specification. This allows standard products to be acquired from a more cost-effective vendor market as integrated circuits can be purchased from chip vendors, and software from software vendors. The primary objective of SPARC was to support the C programming language, numerical applications using FORTRAN, and artificial intelligence and expert system applications using Lisp and Prolog [RT88].

1. The SPARC Architecture

The SPARC architecture consists of an integer unit (IU), a floating-point unit (FPU) configured around a 32-bit virtual address bus, and 32-bit instruction and data busses. The storage system includes a memory management unit and a cache system for both instructions and data. Figure 2.5 shows the arrangement and interaction between components of the architecture. Some implementations of SPARC also include a coprocessor (CP). The IU, FPU, and CP each has its own set of registers.

a. The Instruction Unit (IU)

The IU performs the basic processing for the SPARC architecture. It executes the logical, arithmetic (except floating-point), control transfer, memory reference, and
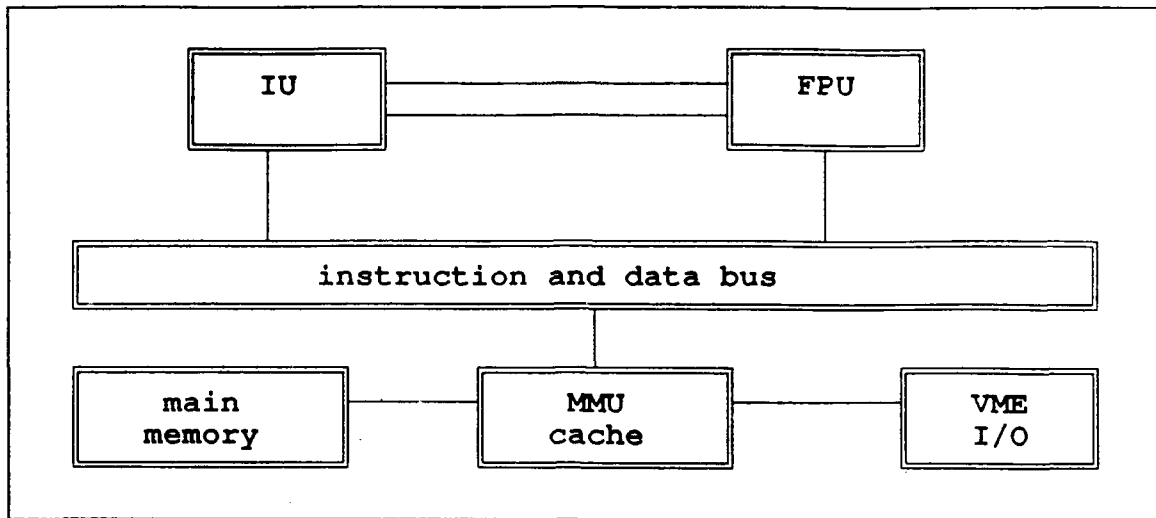
Figure 2.5 SPARC Architecture Components Diagram

multiprocessor instructions (except floating-point operations). It can have between 40 and 520 general-purpose registers, depending on the implementation and register window configuration. In addition to the window registers, the IU includes the processor state register (PSR), the window invalid mask (WIM), the trap base register (TBR), the program counters (PC and NPC), and the multiply stop register.

b. The Floating-point Unit (FPU)

The FPU performs floating point operations concurrently with the IU. It has 32 floating-point registers. Double precision numbers occupy an even-odd pair of register, and extended precision values occupy four consecutive registers. The FPU uses a queue to place floating-point instructions until they are ready to be executed. While floating-point operations are executing, the IU also continues to execute instructions. The FPU registers are accessible

only by special memory load and store instructions. These instructions, called floating-point load/store instructions, are not FPU operations, but IU operations. The IU generates the address and the FPU recognizes and processes the floating-point instructions [Gar91].

c. SPARC Registers and Register Windows

The SPARC is characterized mainly by its register intensive design. The IU, FPU, and CP each have their own set of registers, all of which are 32-bits wide. The use of these registers reduces memory traffic which significantly speeds up program execution. SPARC further exploits the use of registers through a register windowing scheme. The 40 to 520 registers available to the IU are made possible through the partitioning of the register set into 2 to 32 overlapping register windows [HP90]. The actual number of registers is implementation dependent.

The primary purpose of the register windows is to facilitate more efficient parameter passing during the procedure calls of a program execution. During execution, a program may access 32 general-purpose registers: 8 ins, 8 locals, and 8 outs belonging to each window, and 8 global windows. Figure 2.6 shows a design of register windows. The different windows are identified by the Current Window Pointer (CWP) which decrements during a procedure call to activate the

next window and increments at procedure exit to activate the previous window [Gar91].

```
┌─────────────────────────────────────────────────────────────┐
│                                                             │
│   ┌──────────────────┐          ┌──────────────────┐        │
│   │ R[31]            │          │ R[7]             │        │
│   │   :     ins      │          │   : globals      │        │
│   │ R[24]            │          │ R[1]             │        │
│   ├──────────────────┤          ├──────────────────┤        │
│   │ R[23]            │          │ R[0]     0       │        │
│   │   :     locals   │          └──────────────────┘        │
│   │ R[16]            │                                      │
│   ├──────────────────┼──────────────────┐                  │
│   │ R[15]            │ R[31]            │                  │
│   │   :     outs     │   :     ins      │                  │
│   │ R[8]             │ R[24]            │                  │
│   └──────────────────┼──────────────────┤                  │
│                      │ R[23]            │                  │
│                      │   :     locals   │                  │
│                      │ R[16]            │                  │
│                      ├──────────────────┤                  │
│                      │ R[15]            │                  │
│                      │   :     outs     │                  │
│                      │ R[8]             │                  │
│                      └──────────────────┘                  │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```
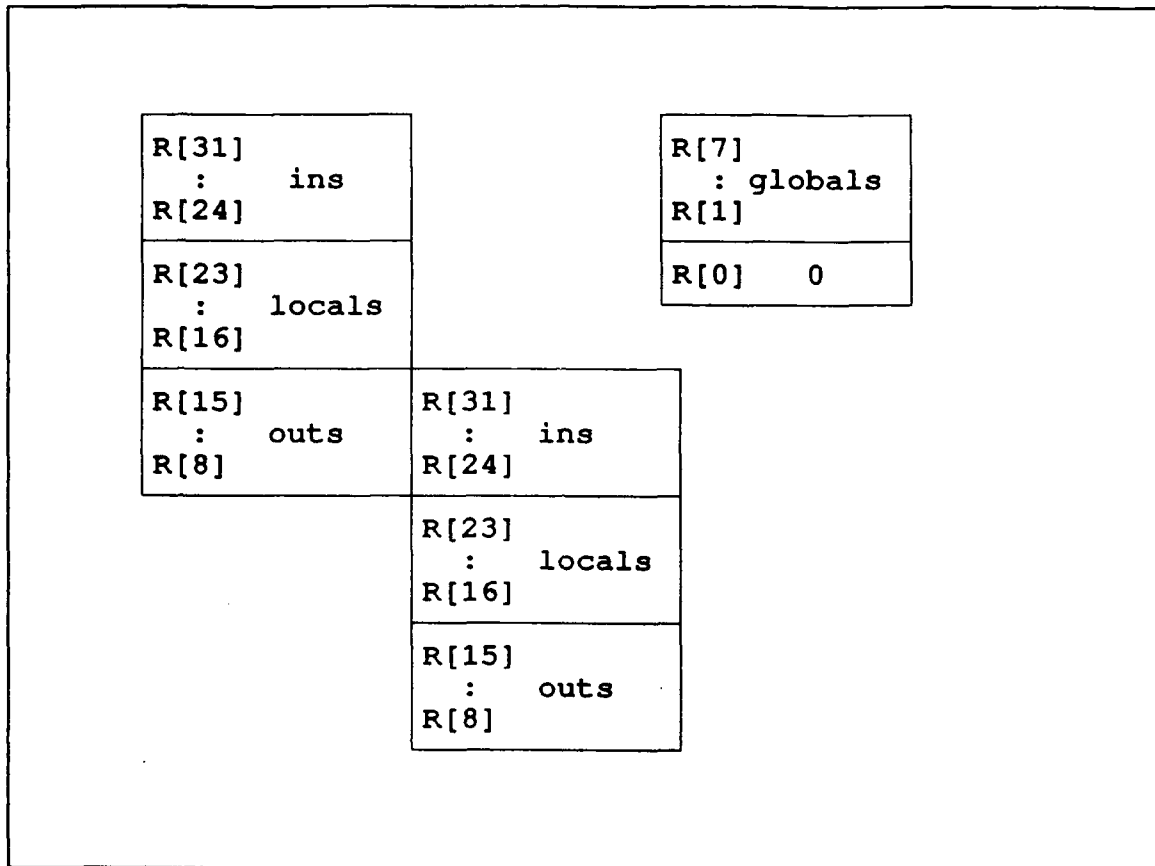
Figure 2.6   SPARC Register Windows.

As shown in Figure 2.6, 8 registers overlap each window.   Registers *R[8]* to *R[15]* of a procedure caller's window become *R[24]* to *R[31]* after the call.   *R[16]* through *R[23]* are unique registers to each window.   Global **register** *R[0]* always contains the value *0*, because it is the most frequently used constant and should be easily available at all times.   The window registers are sometimes labeled *I[0]* to *I[7]* for *R[24]* to *R[31]* respectively for *in* registers, *L[0]* to *L[7]* for *R[16]* to *R[23]* for *local* registers, *O[0]* to *O[7]* for

18

R[8] to *R[15]* for *out* registers, and *G[0]* and *G[7]* for the global registers *R[0]* to *R[7]*.

Advantages of using register windows include reductions in the number of load and store instructions and, consequently, a decrease in the number of cache misses. Register window operations are not without their drawbacks. When all windows are full and a procedure call occurs, an overflow occurs and the window trap handler must move 16 registers into memory. An underflow occurs when a procedure return occurs and the windows are empty, causing the trap handler to move 16 registers from memory. The cost of an overflow and an underflow are about 60 cycles each [HP90].

2. The SPARC Instruction Set

The SPARC instruction set consists of 55 basic integer and 13 floating-point instructions. All instructions are 32-bits wide and are identified by one of three different instruction formats. There are five basic categories of SPARC instructions: (1) load and store instructions, (2) arithmetic/logic/shift instructions, (3) control-transfer instructions, (4) read/write control register instructions, and (5) coprocessor operations [RT88].

a. SPARC Instruction Types

(1) Load and Store Instructions. Load and store instructions are also called memory reference instructions as they are the only instructions that access memory. These

19

instructions use byte, halfword, word, and doubleword operands. The load and store instructions can also be used to access up to 256 different address spaces in the system by the use of an address space identifier (asi). Figure 2.7 shows two different load instructions and two different store instructions.

```
(1)    ld     [%g1+520], %g1
(2)    ldd    [%o6+94], %g1
(3)    st     %o7, [%o7+140]
(4)    sth    %o5, [%o5+o7]
```

Figure 2.7   Sample SPARC Load and Store Instructions.

The first instruction is a load single integer instruction, which moves a word from memory into register %g1. In this example, the memory location is denoted by the sum of contents of register %g1 and the constant 94. The second instruction, the load doubleword, moves a doubleword from the memory location indicated to %g1. The store instruction in the example stores the value in %o7 into the memory address indicated by the sum of [%o6+140]. The last example, a store halfword, moves the least significant halfword from %o5 to the memory location specified by the sum of contents of %o5 and %o7.

(2) Arithmetic/Logic/Shift Instructions. The Arithmetic, logic, and shift instructions perform operations on two operands and put the results into a destination

register. The operands can be either constants or register contents. Figure 2.8 show examples of each of the three types of instructions.

```
add    %17,%g1,%13
or     %g0,71,%o4
sll    %o0,2,%o2
```

Figure 2.8   Sample Arithmetic/Logic/Shift Instructions.

The *add* instruction adds the contents of registers %17 and %g1, placing the result in %13. The *or* instruction implements a bitwise logical operation on the contents of %g0 and the constant 71, placing the results in %o4. The shift instruction, *sll*, shifts the value of the contents of %o0 by the number of bits indicated, 2, placing the result in %o2.

(3) Control Transfer Instructions. Control transfer instructions consist of conditional and unconditional branch, jump, call, trap, and return from call instructions. These instructions changes the value of the program counter. Figure 2.9 shows examples of the types of control transfer instructions.

```
bne     11, %g1
jmpl    %o7,8,%g0
call    75
rett    75
```

Figure 2.9 Sample Control Transfer Instructions

The branch instruction, *bne*, evaluates a condition code and the branch is taken if the condition is true. In this example the target address is the PC value plus 4 (the address of the next instruction) times the value of %g1. The *jmpl* instruction causes a control transfer to the address indicated by the sum of %o7 and 8, placing the PC in the destination register %g0. The *call* and *rett* instructions direct a control transfer to the indicated memory address.

(4) Special Registers Read/Write Instructions. These instructions read the contents or write new values to the four special registers defined by the SPARC: Processor State Register (PSR), Trap Base Register (TBR), Window Invalid Mask (WIM), and Y register which is used for 64-bit integer multiplication.

(5) Coprocessor Operations. These instructions perform floating-point calculations, as well as operations on floating-point registers. They also include instructions involving the optional coprocessor.

b. SPARC Instruction Formats

Figure 2.10 shows the three types of instruction formats and the fields and bit positions for each format used by the SPARC. The bit ordering in the formats is *little-endian*[1] and the byte ordering is *big-endian*[2]. SPARC instructions have two basic addressing modes: register+register and register+signed-immediate.

(1) Format 1 Instructions. Format 1 has a 30-bit displacement field for Call, and in certain situations, Branch instructions. A *call* may be made to a distant location in a single instruction.

(2) Format 2 Instructions. Format 2 supports Sethi (set high) and branch instructions. The Sethi instruction loads a 22-bit immediate value into the upper 22 bits of the destination register and clears its lower 10 bits. The 22-bit displacement field also accommodates a $\pm$8-Mbyte displacement for conditional branch instructions.

(3) Format 3 Instructions. Format 3 is used for the remaining SPARC instructions. It has fields for two source registers and a destination register. When the *i* bit

---

[1] Little-endian machines store words with the high-numbered bits as the most significant. For example, if the binary number *1000* were represented in litte-endian format, *1* is the high-ordered bit and the most significant bit, whereas. For big-endian representation, *1* would be the least significant bit.

[2] Big-endian byte ordering stores the words with the high-number byte as the least significant.

```
Format 1 (CALL)

┌─────┬───────────────────────────────────────────┐
│ op  │                  disp30                    │
└─────┴───────────────────────────────────────────┘
31    29                                           0

Format 2 (SETHI)

┌─────┬─────┬─────┬───────────────────────────────┐
│ op  │ rd  │ op2 │            imm22              │
└─────┴─────┴─────┴───────────────────────────────┘
31    29    28                                    0

Format 2

┌─────┬─────┬─────┬─────┬─────────────────────────┐
│ op  │  a  │cond │ op2 │          disp22         │
└─────┴─────┴─────┴─────┴─────────────────────────┘
31    29    28    24    21                        0

Format 3

┌─────┬─────┬─────┬─────┬─────┬───────────────┬─────┐
│ op  │ rd  │ op3 │ rs1 │ i=0 │ asi or fp-op  │ rs2 │
└─────┴─────┴─────┴─────┴─────┴───────────────┴─────┘
31    29    24    18    13                    4    0

Format 3

┌─────┬─────┬─────┬─────┬─────┬─────────────────────┐
│ op  │ rd  │ op3 │ rs1 │ i=1 │       simm13        │
└─────┴─────┴─────┴─────┴─────┴─────────────────────┘
31    29    24    18    13    12                    0
```
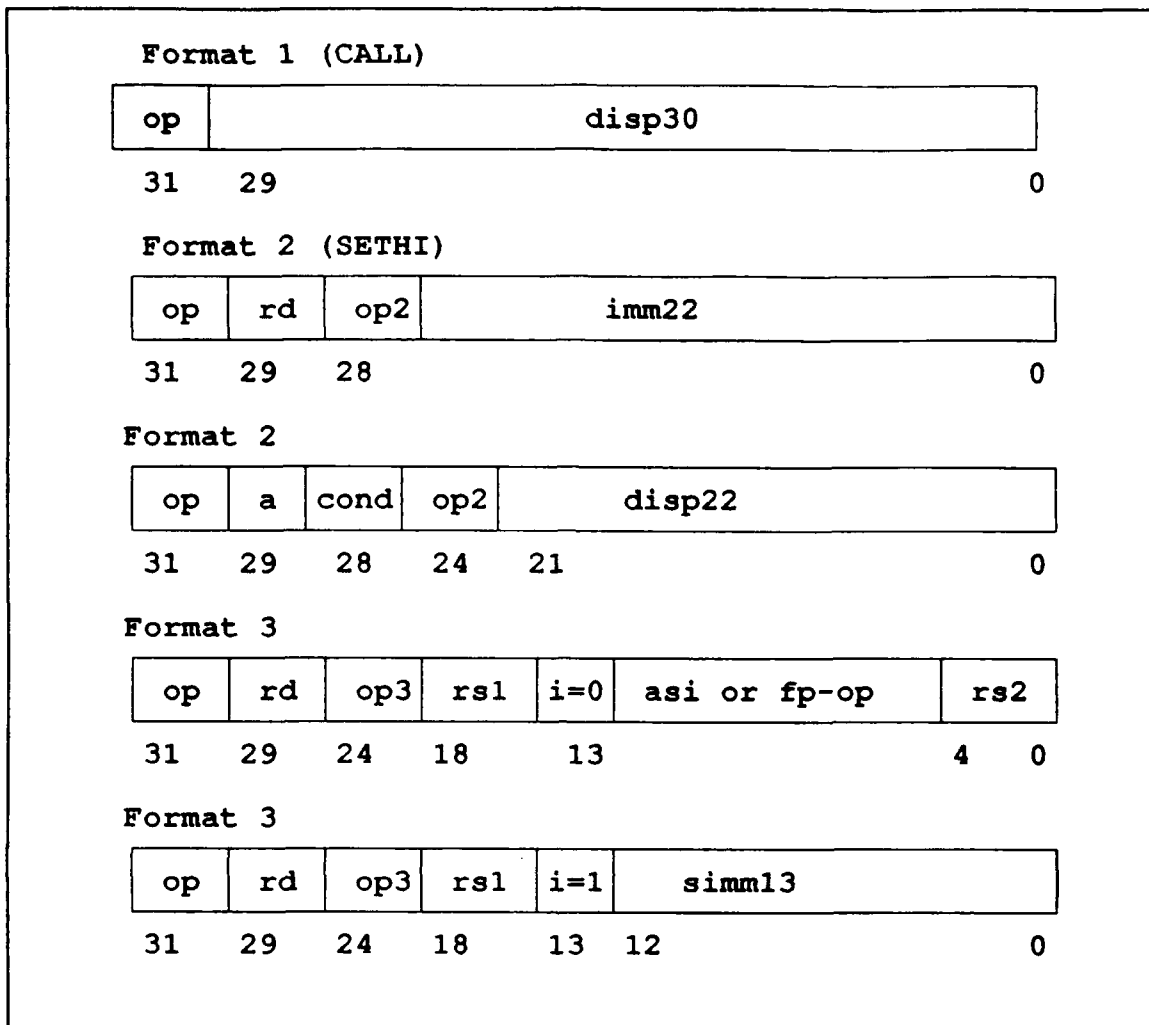
Figure 2.10   SPARC Instruction Format. *Courtesy of SUN Microsystems* [SPA88].

is set (i=1), the 13-bit immediate field value is used instead of the second source register.   The load and store instructions use the upper 8 bits of the immediate field as an extension to the opcode fields to define floating-point instructions.   Unused values for opcodes are reserved for future expansion and designated *unimplemented*.

3. SPARC Pipelines

The SPARC SF9010IU and CYC601 processors use a four-stage pipeline: fetch, decode, execute, and write. Each stage performs a subset of operations needed to complete the execution of an instruction as depicted in Figure 2.11. Each stage completes its operation in a given cycle.

```
             enter ──┐
                     │
              ┌──────────────┐
              │ Fetch Stage  │
              └──────────────┘
                     │
              ┌──────────────┐
              │ Decode Stage │
              └──────────────┘
                     │
              ┌──────────────┐
              │ Execute Stage│
              └──────────────┘
                     │
              ┌──────────────┐
              │ Write Stage  │
              └──────────────┘
                     │
                   exit
```
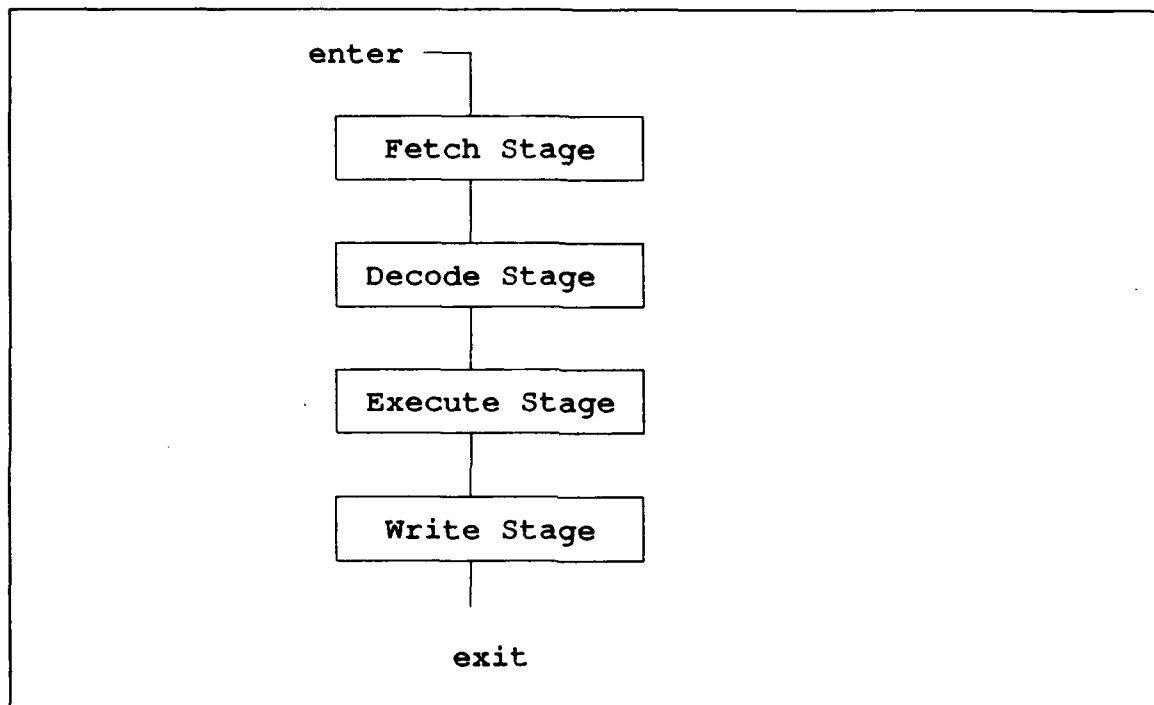
Figure 2.11 SPARC Pipelined Execution of Instructions.

At the fetch stage, the address of the instruction is sent out and the instruction is brought into the pipeline. During the decode stage the source operands are read from the registers and passed to both the execution unit and the instruction unit for later processing. Also, at this stage the address of the next instruction is calculated. In the execute stage, arithmetic and logic operations are performed.

25

The results of these calculations are stored in temporary registers before they are written into the appropriate destination registers. The write stage of the pipeline writes the results in the register file, and the instruction is not finished executing [NA91].

The four-stage pipeline is illustrated in Figure 2.12. Although it takes four cycles from start to finish of each individual instruction, after the initial instruction completes, an instruction is completed every cycle afterwards (ignoring pipeline hazards). Also notice that when the first instruction, I(1), is in the final stage of the pipe, instructions I(2), I(3), and I(4) have already entered the pipe and are being processed.

	1	2	CLOCK CYCLES 3	4	5	6	7	8
I(1)	FET	DEC	EXE	WRT				
I(2)		FET	DEC	EXE	WRT			
I(3)			FET	DEC	EXE	WRT		
I(4)				FET	DEC	EXE	WRT	

Figure 2.12 A SPARC Four-Stage Instruction Pipeline: Fetch (FET), Decode (DEC), Execute (EXE), Write (WRT).

The SPARC B5000 uses a five-stage pipeline: fetch, decode, execute, memory, and write. The memory stage is located between the execute and write stages of the previous

pipeline example. The memory stage is used for those instructions that have memory references. This stage performs the data transfers after the execute stage generates the memory address. The write stage places the results data from the memory stage into the register file [ABMP91].

C. THE LOCKUP-FREE CACHE INTERFACE

1. General

Although RISC has proven to be a high performance architecture, situations such as data dependencies between instructions, conditional branch instructions, and memory access penalties prevent RISC from achieving the goal of one instruction per cycle. The high performance of RISC architectures is partially attributed to their use of high speed cache systems. One important performance criteria of a cache is to maximize the probability that the requested data is present which is to attain a maximum hit ratio. Another criteria is to ensure that data access time from the cache is minimal. Thus, cache design is a major issue in computer performance. The parameters that are targeted in designing more efficient caches include cache size, cache associativity, cache replacement policy, line size, and hardware prefetching [Por89]. Most cache memories have hit rates between 85% and 95%, and cache memory access times are 5 to 10 times faster than main memory access times [LFK90].

Regardless of the hit rate of a cache memory system, a miss or a write instruction will require main memory access. Main memory access is a major factor in performance degradation of a computer system. Generally, there are two ways to reduce memory access penalties: (1) reducing the number of memory requests, and (2) reducing the average latency [Por89]. RISC approaches the first problem by efficient register allocation. The second problem must be solved by acquiring more memory bandwidth.

To further reduce the adverse effect of a non-cache access on a RISC architecture, a cache-to-main memory subsystem, called a lockup-free cache interface is proposed as a possible solution. Such scheme was used by Kroft for a uniprocessor architecture [Kro81], and by Scheurich and Dubois for a multiprocessor architecture [SD91]. As the name implies, a lockup-free cache interface prevents non-cache data requests from "locking up" the processor. The processor is allowed to continue processing instructions while memory requests are being handled. Figure 2.13 illustrates a memory hierarchy that includes a lockup-free cache interface.

2. The Lockup-Free Cache Interface Concept

A lockup-free cache interface is a component of cache-based memory systems used to control access to main memory. The objective of the lockup-free cache interface is to
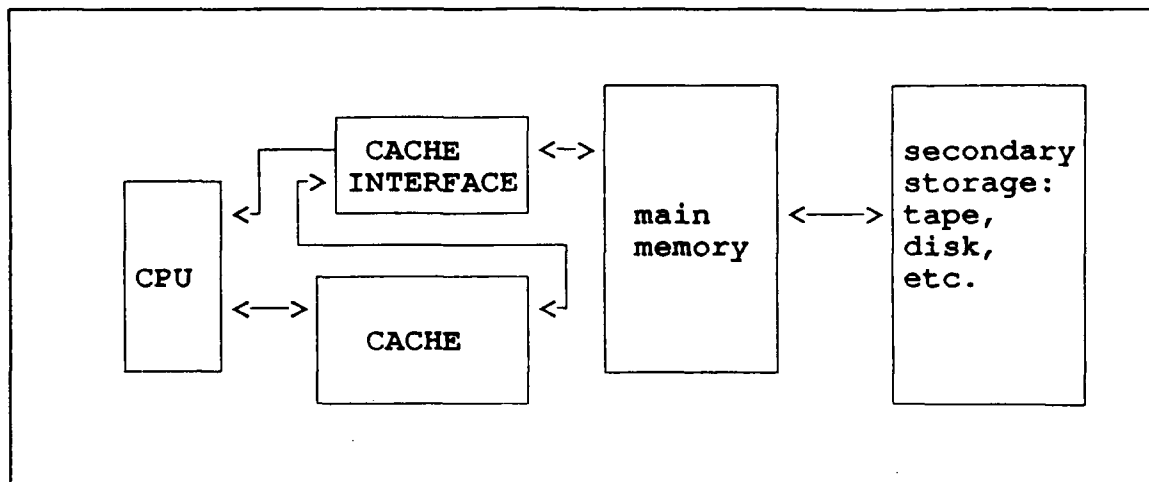
Figure 2.13   Memory Hierarchy with Lockup-Free Cache Interface.

increase the effectiveness of cache-based memory systems by minimizing the penalty for main memory accesses [Kro81]. The basic concept is to prevent the processor from freezing on non-cache accesses. On RISC machines, main memory accesses are required for cache misses and for write instructions.

3. Design Issues of Lockup-free Cache Interfaces

a. Memory Request Queue

A major design consideration of a lockup-free cache interface is the use of a waiting queue for main memory requests. During processing, when a cache miss or a write instruction is encountered, the request is placed in a queue for main memory requests. At the same time that memory requests are being served from the queue, the processor continues to issue new instructions until the memory request queue fills or an instruction is dependent on data in the memory request queue. If an issued instruction is dependent

on data in the memory queue, then the instruction is blocked or put on hold until the required data is available.

b. *Other Design Issues*

The effects of a lockup-free cache interface on RISC performance also depend on other important design issues. One design issue is whether to use a shared or separate memory request queue for misses and writes. Another issue is whether to use a queue for blocked instructions or to freeze the process when an instruction is dependent on a queued data request. The length of the queue for main memory requests is also a design issue that may determine the effects of the lockup-free interface.

III. A LOCKUP-FREE CACHE INTERFACE MODEL

A. THE LOCKUP-FREE CACHE INTERFACE DESIGN

1. General

In presenting a model for a lockup-free cache interface, we do not attempt to define a specific cache memory design. We also assume that there is a separate cache for instructions and data, which is the case in some SPARC implementations. Thus, the effect of instruction misses is not considered as it is assumed to be insignificant. With the high hit rates of most cache systems, most main memory accesses are likely to be writes instead of instruction misses. A write-through policy is also assumed.

2. The Major Components of the Cache Interface

Figure 3.1 is an overview of the components needed to implement the lockup-free cache interface. The interface has two queues: Memory Access Queue (MAQ), and Blocked Instruction Queue (BIQ). The MAQ is used for storing read misses and writes. It may either be a FIFO or priority queue, or it may be a split queue configuration with reads and writes in separate MAQs.

The BIQ holds target register numbers of the read instructions that are in the MAQ. The BIQ entries correspond to the read entries in the MAQ. Therefore, the BIQ may be a
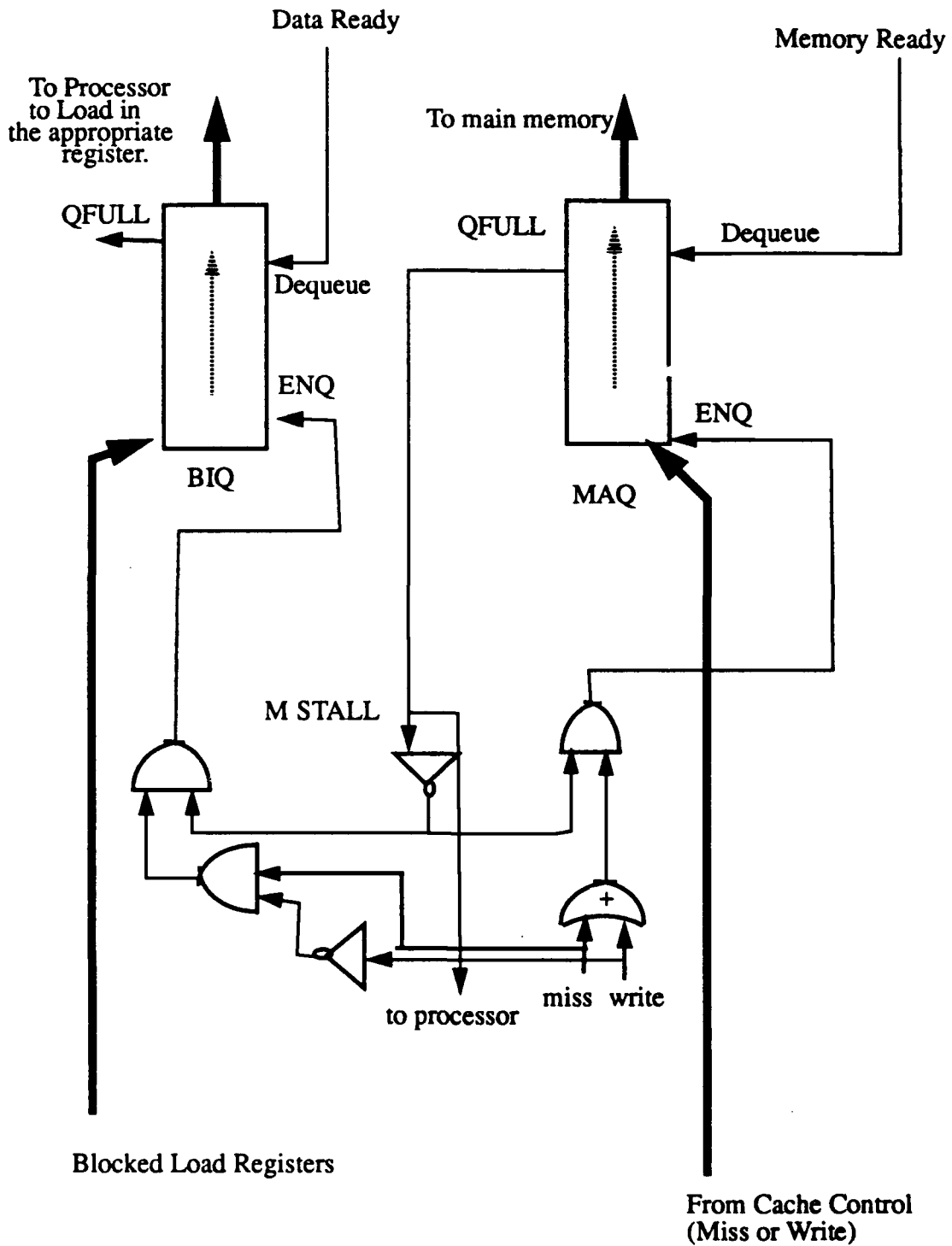
Figure 3.1 Structure of the Lockup-Free Cache Interface

FIFO or a priority queue, depending on the MAQ. Figure 3.2 illustrates typical entries in the MAQ. The main memory controller uses a Memory Ready (MR) signal and a Data Ready (DR) signal to communicate with the cache interface.

R	address	unused

W	address	data
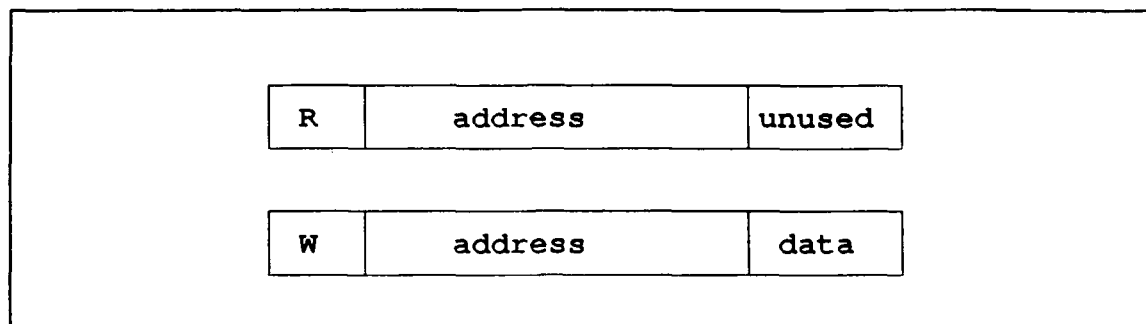
Figure 3.2 MAQ Formats for Reads and Writes

B. OVERVIEW OF THE SYSTEM OPERATION

1. Lockup-Free Cache Operation

The cache operates like a typical cache as long as no read misses or writes are encountered. On a read miss, an entry is added to the MAQ and the destination register of the instruction is entered into the BIQ. On a write instruction, an entry is enqueued in the MAQ.

The main memory controller sends a Memory Ready (MR) signal to the cache interface indicating that another memory access can be initiated. When the MR signal is received by the cache interface, the next entry in the MAQ is dequeued and sent to the memory controller. The memory controller also sends a Data Ready (DR) signal to the interface indicating that the data access from main memory is ready to be loaded.

33

Thus, an entry in the BIQ is then dequeued and the data loaded into the register.

2. The Processor Operation Model

The processor stalls on three conditions: (1) the instruction to be issued uses a register that is being used as a target by an instruction in the MAQ or main memory, (2) the instruction to be issued uses a register that is the target of a blocked load instruction, (3) and the MAQ fills up. When any of these situations occurs the processor stalls until the DR signal is received and the BIQ dequeues the appropriate register. The MAQ will continue to process requests until the target register causing the stall receives the required data.

Using the lockup-free cache, the processor is assumed to be able to issue instructions before the previous ones complete. Thus, the instructions can complete out-of-order. This is generally the case with writes and read misses that must wait to be served by main memory. While these instructions are waiting for main memory service, the processor continues to fetch and execute instructions. The processor cannot issue an instruction that depends on a previous instruction that is currently blocked.

IV. SIMULATION TOOLS

To provide the capabilities for an analysis and *view* of the lockup-free cache and the RISC architecture, three different simulation tools are used in this study. These tools allow us to observe the behaviors of several variations of the architecture. Additionally, these tools produce address traces of actual program executions. This allows for more accurate and realistic results from the modelled RISC architectures. Figure 4.1 illustrates the simulation environment for this research.

The first simulator is the SPARC Performance Analyzer (SPA) 1.0. The SPA is used to produce address traces of programs and to provide instruction count data for those traces. A second simulator is an address trace translator which produces readable instruction records and modified address trace files for use in other simulation tools. The instruction records provide the user with information such as instruction and data addresses, binary representations, opcodes, and registers used. The third simulator is a lockup-free cache interface which simulates a cache-to-main memory subsystem used to reduce the cost of main memory access.
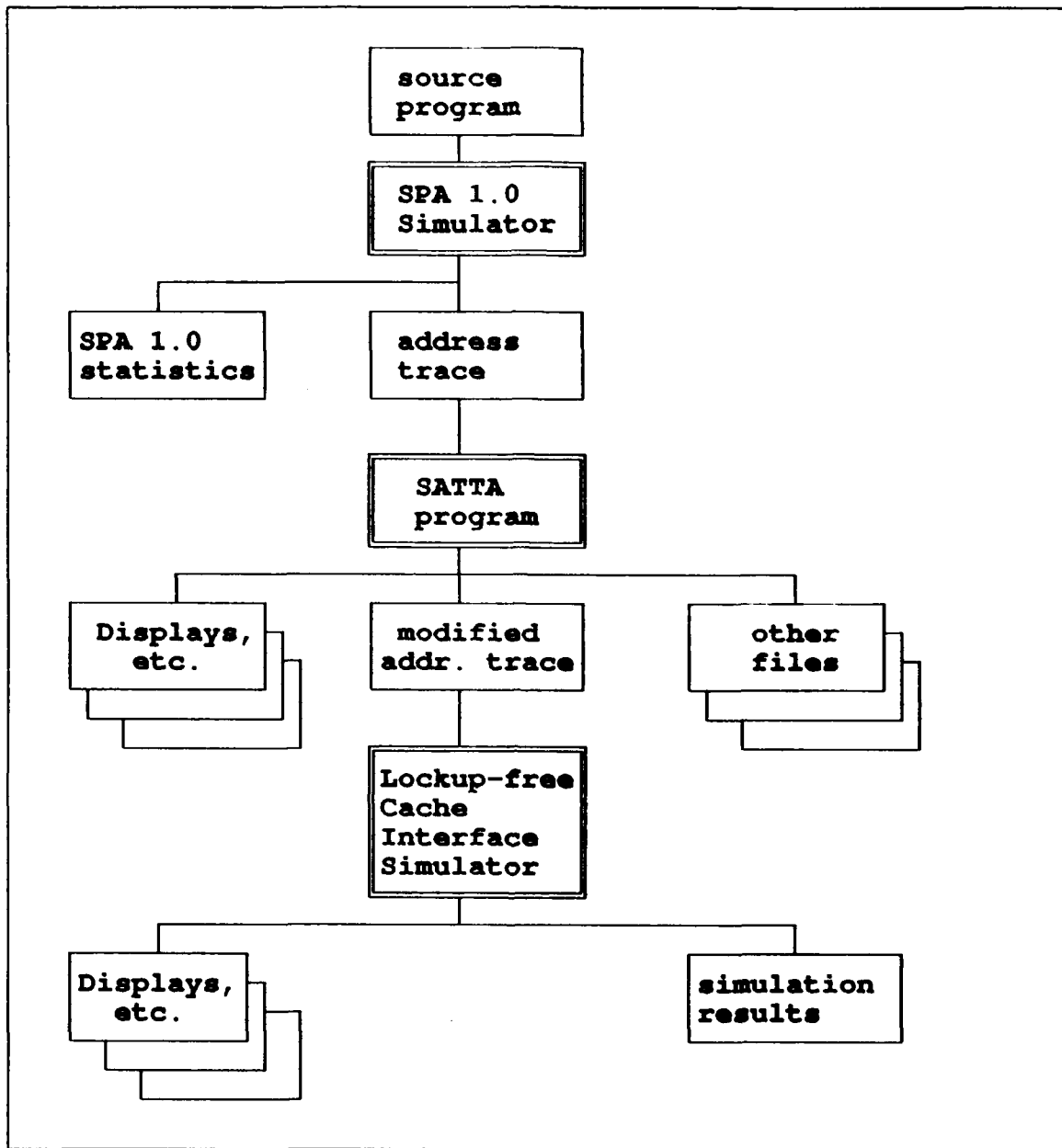
Figure 4.1 Simulation Environment

A. THE SPARC PERFORMANCE ANALYZER 1.0 SIMULATOR

The SPARC Performance Analyzer (SPA) 1.0 is a package of simulation tools used to analyze the performance of programs executed on SPARC machines. SPA can simulate two different

36

SPARC implementations: the Cypress CY7C601 and the Fujitsu MB86901. Simulations can be run on SPARCstations or any machine using a Sun OS4 operating system. The SPA was developed by Gordon Irlam and made available to users via file transfer protocol (ftp). The specific version we used was ported from *ftp.uu.net:/system/sun/spa-1.0.tar.Z*.

The SPA 1.0 consists of three major components: SPY, SPANNER, and SPOUT. The SPY is a tool that traces the execution of a program and produces an address trace file; the SPANNER is a tool that converts the address traces into instruction count files; and the SPOUT is a component that formats and displays the results of the instruction count.

There are numerous other tools in the SPA package that support the three major components. These tools add to the flexibility of SPA by allowing the user to set various parameters of the architecture and determining the effects on performance. Appendix A provides additional information on the major components of SPA and their uses.

B. THE SPARC ADDRESS TRACE TRANSLATOR/ANALYZER

1. General

The SPARC Address Trace Translator/Analyzer (SATTA) is a program that takes as input the address trace files generated by SPA and translates them into detailed readable instruction records. The SATTA also generates SPARC assembly

language files and specially modified address traces to be used in the lockup-free cache interface simulator.

2. Instruction Address Trace Format

The composition of each instruction record in the address trace file is illustrated in Figure 4.2. The execution trapped (*et*) field is a one that indicates the execution status of the instruction. A *0* means the instruction was executed, and a *1* indicates that it was not executed. The data address valid (*dav*) field is a one-character field, *0* or *1*, indicating whether or not the data address field is valid. The data address field is valid if the instruction is a load or store instruction.

```
struct Instruction {
    char   et;
    char   dav;
    short tn;
    unsigned long op;
    unsigned long ia;
    unsigned long da;
};
```

Figure 4.2 Trace Instruction Format

The *op* field contains the integer value of the actual SPARC instruction. The instruction address (*ia*) field is the address in memory where the instruction was referenced or fetched, and the data address (*da*) field indicates the memory location of the referenced or target data.

3. The SATTA Instruction Record

The SATTA program translates each address trace record into a very detailed, more readable display of information. A sample record is shown in Figure 4.3. This record includes all the information generated by the SPY component. In this example we see that the instruction was executed and that the value indicated in the data address field is valid.

```
record: 1
exec status: 0
valid addr: 1
trap no.: 0
instruction: -805166984
binary representation: 11010000000000010000100000001111000
op_field: 3
opcode value: 0
opcode: ld
rd value: 8
rs1 value: 8
index_bit: 1
simm13: 120
inst addr: 26b74
data addr: 26c78
```

Figure 4.3 Detailed Expansion of an Instruction Record

The value in the *instruction* field is the SPARC instruction in integer form. This integer value is somewhat vague to the user as displayed. However, *the binary representation field* provides a more visual means for determining the components of the instruction.

The 32-bit binary representation field is matched to the SPARC instruction format templates to determine the type of instruction, the opcode, registers used, and displacement

39

values. The *op_field* value is the operation type. The operation type also determines the instruction format type. The *opcode value* field is the integer value of the opcode within the operation type. This value is translated into the actual *opcode*. The *rd value* is the destination register; rs1 value is the source register; *index_bit* indicates whether or not index addressing is used; and *simm13* is a signed integer value used in calculating an immediate address. The *instruction* and *data address* values are translated directly from the trace file. Other instruction formats may contain different fields such as *rs2* for a second source register, or *immediate address*.

4. SATTA File Generators

In addition to producing detailed instruction record translations, the SATTA program also generates various types of files. These files may be used for additional tracing, further analysis, or as input files to other programs and simulators.

a. SPARC Assembly Language Files

One type of file generated by the SATTA is an assembly language program. The file is produced from information taken directly from the translated instruction record. Figure 3.4 is an excerpt from an assembly language file generated by SATTA. This capability allows the user to

see the assembly language equivalent of the traced program or
to manually trace parts of the program.

```
83:    nop
84:    bvs      8
85:    or       %17,628,%17
86:    sethi    8,%o0
87:    or       %o0,539,%o0
88:    call     35
89:    or       %g0,5,%g1
90:    ba       6,%o0
91:    or       %g0,5,%g1
92:    ta       %g0,0,%o0
```

Figure 4.4 Assembly Language Code Produced by SATTA

The assembly language file produced by **SATTA** can
also be used as input to other simulators to demonstrate other
features of RISC. Of particular use, the assembly language
file can be used in modelling other RISC architecture
components, such as pipelines, caches, or proposed add-ons.
The instructions are in the standard SPARC assembly language
syntax. The file is stored in ASCII format, thus can be
easily used by other programs on most any type of machine.

b. Cache Address Trace Files

Cache address trace files are specially tailored
files for use by the lockup-free cache interface simulator.
The files consist of only that information from instruction
records required to sufficiently simulate the cache interface.
The use of only pertinent information speeds up the
simulation. All data included in the files is in hexadecimal

41

form. Figure 4.5 shows records from a cache trace file.
Although the cache address trace files were created
specifically for the lockup-free cache interface simulator,
they can also be used as address trace input files for other
cache simulators.

Code	Address	Rs1	Rs2	Rd
2	0000213c	0e		01
0	f7fff9c0			01
2	00002140	00	01	00
2	0000000b			01
3	00002148			
2	00002170	0e		01
0	f7fff9c8			01
2	00002178	0e		13
1	f7fff9d4			13
2	0000217c	0e		15

Figure 4.5 Records from Cache Address Trace File

Each instruction generates a cache interface
record. Each record is assigned a code of *2*, except for
branch instructions which has a code *3*. The instruction
address and the source and destination registers (*Rs1*, *Rs2*,
and *Rd*) are also part of the cache address trace records.

All load and store instructions generate an
additional cache interface record. The records generated by
loads are given a code of *0*. The memory address of the needed
data, along with the target register for the load operation is
included in the additional load record. Similarly, the

42

additional records generated by the store instructions contain the memory address of where the data is to be stored, and the register number containing the data. The code for the store instructions is 1.

The different code types are used by the lockup-free cache interface to determine the number of cycles required to execute each type of instruction. Branch, load, and store instructions all generally require more than one cycle to execute. The cache interface simulator sets the simulated number of cycles required for these instructions.

C. THE RISC CACHE INTERFACE SIMULATOR

1. General

The RISC Cache Interface Simulator (RICIS) is a simulation tool that models a program executing on a RISC machine using a lockup-free cache interface. The primary objective of the RICIS is to calculate the performance of the RISC using the interface. The simulator is event-driven and uses the modified address trace files produced by SATTA as the input program. The results from the RICIS simulation is compared to the results from running the same program with the SPA simulator to determine the effects of the design.

2. The RICIS Program

The RICIS is designed to simulate several different configurations of a lockup-free cache interface. It can be

easily modified to simulate even more design alternatives and to perform various statistical functions. RICIS can simulate large program executions since the traces have been modified to consist of only a few characters of information per instruction, and the trace instructions are discarded after they are processed. Therefore, although the traces generated by SPA and SATTA consume a considerable amount of disk space, RICIS can run most simulations without requiring additional disk space.

3. The RICIS Operation

a. Assumptions and Constraints

The assumptions and constraints of the RICIS are as follows:

(1) *Floating-point instructions.* RICIS does not simulate floating point instructions. Floating point instructions are handled the same as integer instructions and are assumed to execute in a single cycle. Although this differs significantly from reality, this constraint is consistent with the SPA constraint. Therefore, comparing results produced by the two simulators using floating-point instructions should not present a problem.

(2) *Simulating cache hits and misses.* There is currently no cache simulator available to determine if a load instruction is a cache hit or miss, thus the determination of a load hit or miss is simulated using a random number

generator. The user determines the hit ratio to be simulated. Once a load instruction is encountered, the random number generator produces a number between 0.0 and 100.0. If the generated number is greater than the hit ratio entered by the user, the load is considered a cache miss, otherwise a hit. We realize that cache hits and misses are not random, but this feature should at least produce the same percentage of hits.

(3) *Instruction types*. The input to the RICIS is the modified address trace produced by SATTA. The RICIS does not need to distinguish between instruction opcodes. Thus, all instructions of the address are categorized into four different types: loads, stores, branches, and others. All instructions of each instruction type are assumed to execute in the same amount of time. Basically, the RICIS needs to know whether an instruction is a memory instruction or if it requires more than one cycle to execute.

b. *Setting Simulation Parameters*

To run the RICIS program, the user enters the command *RICIS*. The program then prompts the user to enter the name of the address trace file and to set the parameters of the lockup-free cache to be simulated. In setting the parameters for the simulation, RICIS offers a variety of design options for simulating a lockup-free cache interface. One parameter choice is the simulated cache hit ratio. The user may enter a percentage value from 0.0 to 100.0.

After entering the cache hit ratio, the user must specify the MAQ configuration. The choices are FIFO and priority. If priority queue is chosen, the user has a choice between simulating a single queue for writes and read misses, or a separate queue for each type of MAQ entries. The user then sets the length of the MAQ.

Another parameter the user must set is the main memory access penalty (in number of cycles) for cache misses and store instructions. The users may also set as a parameter the number of cycles to delay for branch instructions and for load dependency situations. A load dependency situation occurs when an instruction immediately following a load instruction requires the loaded results.

The final response the user must enter is whether or not to view a cycle-by-cycle execution of the simulation. If the user does not wish to view the simulation, performance results are provided at the end of the simulation run. The view capability lets the user observe the behaviors of the target architecture under varied workloads. With address traces containing hundreds of thousands of instruction records, the user may choose to view partial executions. This is accomplished by using the option of viewing the results in intervals. The user may elect to view interim results every 100, 700, 10,000, etc., instructions. At each interval, the option of terminating the simulation is offered.

c. RICIS features

(1) *The Priority Event Queue (PEQ).* The PEQ is a priority queue that stores the events that drives the program execution. The PEQ basically simulates the processor and the memory controller. There are two events that are required to run the simulation: *issue instruction (ii)* and *leave memory (lm)*. The *ii* event directs the simulator to issue another instruction from the address trace file. The *lm* event directs the simulator to remove the next request from the memory queue. Figure 4.6 shows a PEQ with events entered.

The time entry is the cycle number in which the event can occur. The time is also the priority in determining which event is to occur next. In the example, the next item (event) to be served from the PEQ is an instruction issue, occurring at cycle 22 of program execution. If this were a FIFO queue, the next item to be served would be the *lm* at cycle 29 of execution.

PEQ	
EVENT	TIME
lm	29
ii	22
ii	23
lm	33

Figure 4.6 View of Priority Event Queue

(2) Simulating the Different Memory Queue Schemes.
The RICIS allows the user the option of simulating a either a
FIFO or a priority MAQ queue. It also offers the option of
storing the reads and misses in the same queue or to use
separate queues. Figure 4.7 shows the contents of a simulated
FIFO MAQ with memory requests served on a first-come-first
serve basis. Figure 4.8 illustrates the combined priority MAQ
simulation with memory requests served by precedence to the
given priority value. Figure 4.9 shows the contents of a
separate read and write MAQ simulations. Using this
configuration, requests are served based on the priority
assigned to reads and writes. Requests are serviced FIFO
within their respective queues. The code entry is a *0* for a
read entry and a *1* for a write. The address is the location
in memory where the data is read from or written to.

The priority value for determining the
precedence of a read and a write is set by the user, or it may
be entered into the actual code as a constant. The priority
value determines the next request to be served from the MAQ.
All reads will have the same priority, as will all writes.

(3) Simulating Blocked Instructions. To simulate
the blocked registers that are awaiting main memory access, we
use an array consisting of boolean values for each of the 32
registers. When a read miss occurs, the target register of
the read instruction is marked as blocked (the array index

48

```
              MAQ
     CODE          ADDRESS

       1           0000a130
       1           0000a220
       0           f7fffa30
       0           f7fffa00
       1           0000a232
```

Figure 4.7 View of Simulated FIFO MAQ

```
              MAQ
     CODE    ADDRESS      PRIORITY

       0     f7fffa30        1
       0     f7fffa00        1
       1     0000a130        2
       1     0000a220        2
       1     0000a232        2
```

Figure 4.8 View of Simulated Priority MAQ

```
           Read MAQ
     CODE          ADDRESS

       0           f7fffa30
       0           f7fffa00
```

```
           Write MAQ
     CODE          ADDRESS

       1           0000a130
       1           0000a220
       1           0000a232
```

Figure 4.9 View of Simulated Separate MAQ Scheme

corresponding to the blocked register is set to *true*),

simulating the register waiting for the *data ready* signal from

49

main memory. The register is unblocked (array index set to *false*) when the simulation indicates that the instruction has completed its main memory access. The blocked register array prevents other instructions from using the blocked registers. If a blocked register is referenced by another instruction, a stall is simulated until the register is unblocked.

d. Calculating Performance Results

The CPI is calculated by dividing the number of cycles accumulated by the number of instructions issued. The cycle count includes memory access penalties and stall cycles for load dependencies on block registers. Interim results may also be obtained from the simulator, and additional statistical data can be obtained with minimal modifications.

V. SIMULATIONS AND RESULTS OF LOCKUP-FREE CACHE INTERFACE

A. METHODOLOGY

1. General

In this section we conduct performance evaluations of the lockup-free cache interface using the RICIS program. The simulations show the effectiveness of various design alternatives of the interface. Three different programs were executed to present various workloads. In addition to determining the performance (CPI value) using the interface, a cycle-by-cycle visual trace can be generated by the user to observe the behavior of the system.

2. Structures to be Evaluated

Evaluations are provided based on the following parameters: size of MAQ, type of MAQ (i.e., FIFO, single-queue-miss-priority, and separate-queue-miss-priority MAQ for loads and stores), and combinations thereof. The single-queue-miss-priority stores both read misses and writes in the same queue with read misses having a higher priority. The separate-queue-miss-priority stores read misses in one queue and writes in a separate queue with priority of service given to the read miss queue. For this type MAQ simulation, the combined size of the two queues is used as the queue size. Numerous possible configurations of the lockup-free cache can be

51

simulated. Due to time constraints, however, we can only simulate a few designs. In determining the effects of a particular interface parameter, for each base configuration simulated, a single parameter is varied at a time.

3. Fixed Parameters

The following parameters are fixed for the simulation experiments: (1) cache hit ratio is 0.9, (2) load dependency delay is one cycle, (3) experiments are done using two different memory access delay values, the first one is 50 cycles, and the second is 5 cycles, (4) branch delay is 3 cycles, (5) whenever a priority scheme is used, read misses have a higher priority. The read-over-write priority is chosen because with a high hit rate, most main memory accesses will be writes, thus reads would have to wait until all the writes are dequeued. This further increases the chance of a load dependency stall.

The base configuration used by SPA to compare the results with that of the RICIS is the SPARC CY7C601 processor with the SS2 cache memory. Appendices E, F, and G contain of the SPA generated statistical analysis reports of the three test programs. To determine the effects of a lockup-free cache interface on a RISC processor, we calculate the CPI of the test programs run on the SPA. Since we are using memory access penalty and cache hit ratio as parameters for the

lockup-free cache, we insure that the same parameters are used with the instruction count data from SPA to determine the CPI of the programs without using the lockup-free cache.

In calculating the CPI of the programs run on SPA, we must first know the percentage of the total instruction count that each instruction type (i.e., ALU, Branch, Load, and Store) makes up. This information is attained from the SPA report. We then use the following formula:

$$CPI = \%ALU*a + \%Branch*a*b + \%Store*(m+a) + \%Load*a*r + \%Load*(m+a)*(1-r)$$

where *a* is the number of cycles required to execute the instruction, *b* is the number of cycles for a branch delay, *m* is the memory access penalty, and *r* is the hit ratio. For our experiments, we use: *a=1*, *b=3*, *r=0.9*, *m=50* for the first set of experiments, and *m=5* for the second.

B. TEST PROGRAMS

1. General

To evaluate the lockup-free cache interface, three different types of programs are used. These are all relatively short programs ranging from about 300,000 to 600,000 SPARC assembly code instructions. All of the programs are run under the SunOS4. The programs used for this thesis are described below.

2. Pseudo Code Interpreter

This program translates and executes a specific pseudo-code program. This particular pseudo-code is designed for a simple computer with 2000 words of 10-digit memory. The program reads an instruction from a memory location, decodes it, and then executes it. This process continues until the last instruction is executed. For testing the simulator the pseudo-code program calculates the square and square root of each of the numbers read from locations in memory. The pseudo-code program execution trace consists of 359,777 SPARC assembly language instructions.

3. Launch Trajectory Calculator

This program reads rocket launch data, such as launch time and range, from a file, and calculates the altitude and trajectory of all the launches. The trace consists of 342,440 instructions.

4. Matrix Multiplication

This program performs a 20 X 20 matrix multiplication. The results of the matrix multiplication are put into a third matrix. The lack of sufficient disk space prevents the use of a larger trace. The program consists of 524,852 instructions.

C. SIMULATION EXPERIMENTS

1. General

In conducting the experiments, data was collected on each of the test programs, using MAQ sizes of 0, 1, 4, 8, 16,

and 32. These sizes were chosen to determine the trend of the performance and to determine the optimum queue size. This experiment was conducted for each of the MAQ schemes using the fixed parameters. Figure 5.1 shows the performance results of the FIFO MAQ scheme. Figure 5.2 shows the results of the single-queue-miss-priority MAQ, and Figure 5.3 shows the results of the separate-queue-miss-priority MAQ. For comparison, the CPI values of the test programs without using a lockup-free cache interface are:

(1) Pseudo Code - CPI=1.71 for m=5, 4.41 for m=50

(2) Matrix Mult. - CPI=1.35 for m=5, 2.58 for m=50

(3) Trajectory - CPI=1.71 for m=5, 4.33 for m=50.

2. MAQ Sizes

This experiment examines the effects on CPI of MAQ sizes across the three configurations. The MAQ sizes range from 0 to 30. For first set of experiments we use a memory access penalty of 50 cycles. For each of the configurations and each of the test programs, the CPI improved significantly as the queue size increased from 0 to 12. The CPI value remained virtually the same for queue sizes greater than 12. The average decreases in CPI from the queue size of 0 to 12 were 40% for the Pseudo-Code program, 41.1% for the trajectory program, and 15.5% for the Matrix Multiplication program.
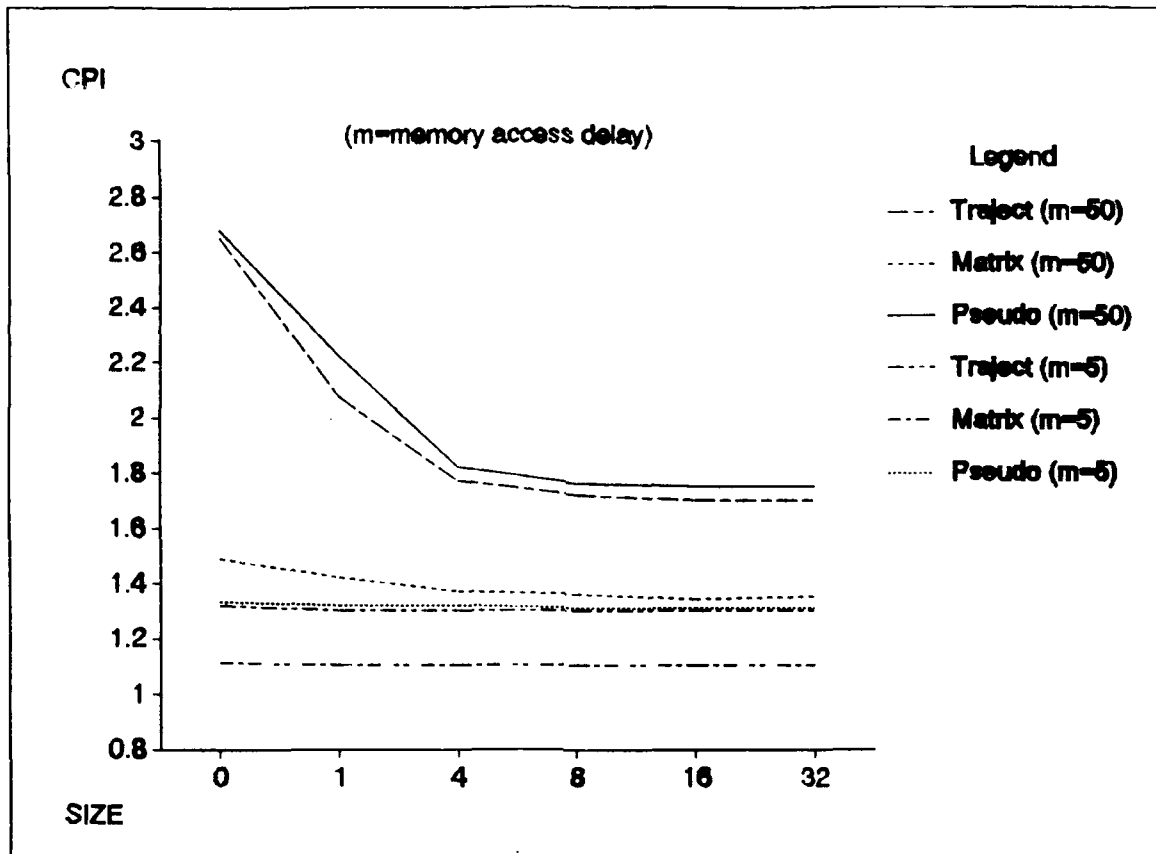
Figure 5.1 Effects of FIFO MAQ Scheme

The largest improvement in CPI occurred as the queue size went from 0 to 1. In this case, the average improvements across the different MAQ schemes were 27.0% for the Trajectory program, 12.7% for the Pseudo-Code program, and 10.0% for the Matrix Multiplication program. A queue size of zero basically simulates not using a queue. In this case the processor stalls when a main memory request occurs and a request is still in main memory.

For the second set of experiments we used a memory delay of 5 cycles with each of the schemes. The results show that there was an average improvement in CPI of less than 2.0% from
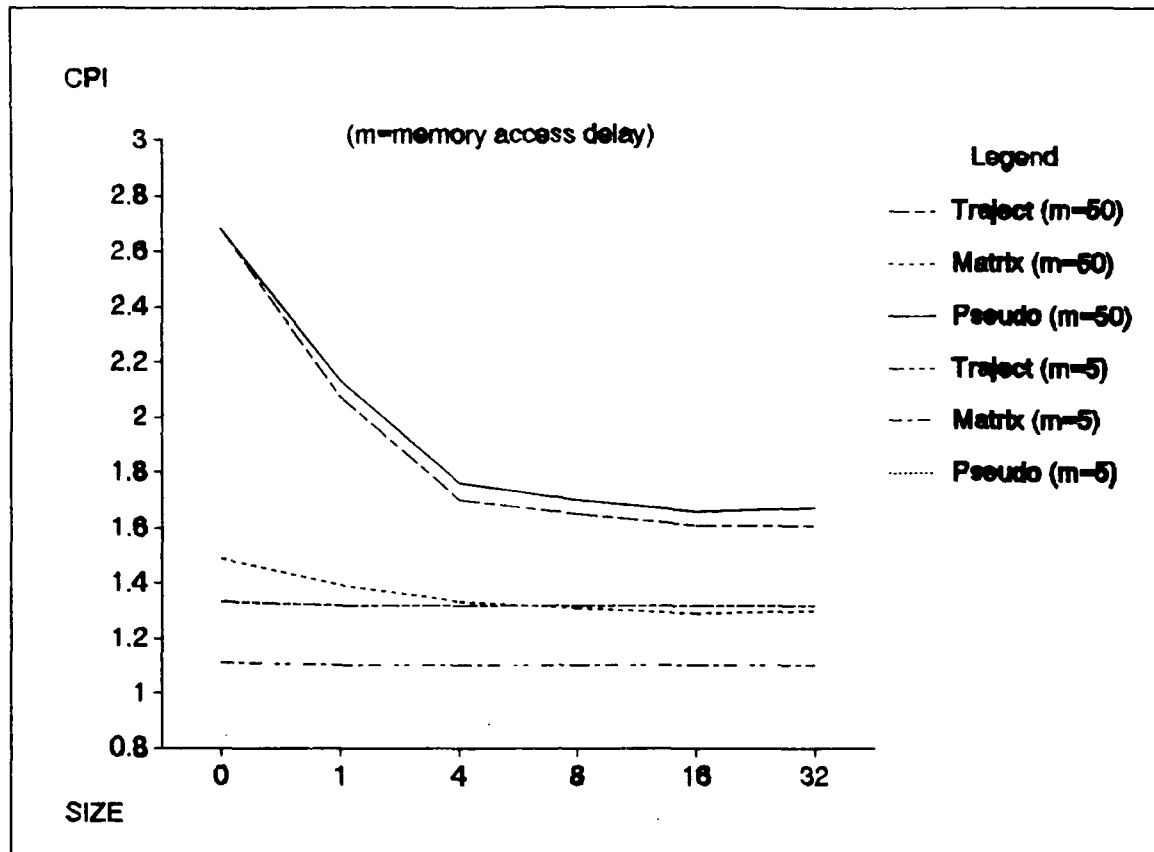
Figure 5.2 Effects of Single-Queue-Miss-Priority MAQ Scheme

a queue size of 0 to 1. There were no further improvement in any of the schemes with queue sizes greater than one. As with the previous experiments, the separate-queue-miss-priority MAQ configuration yielded the best performance, followed by the single-queue-miss-priority MAQ.

3. MAQ Configurations

This experiment examines the effects on the CPI of the configuration of the MAQ. Overall, the separate-queue-miss-priority MAQ configuration presented the best performance, followed by the single-queue-miss-priority MAQ. For the Pseudo-Code program with m=50, the CPI values ranged from 1.75
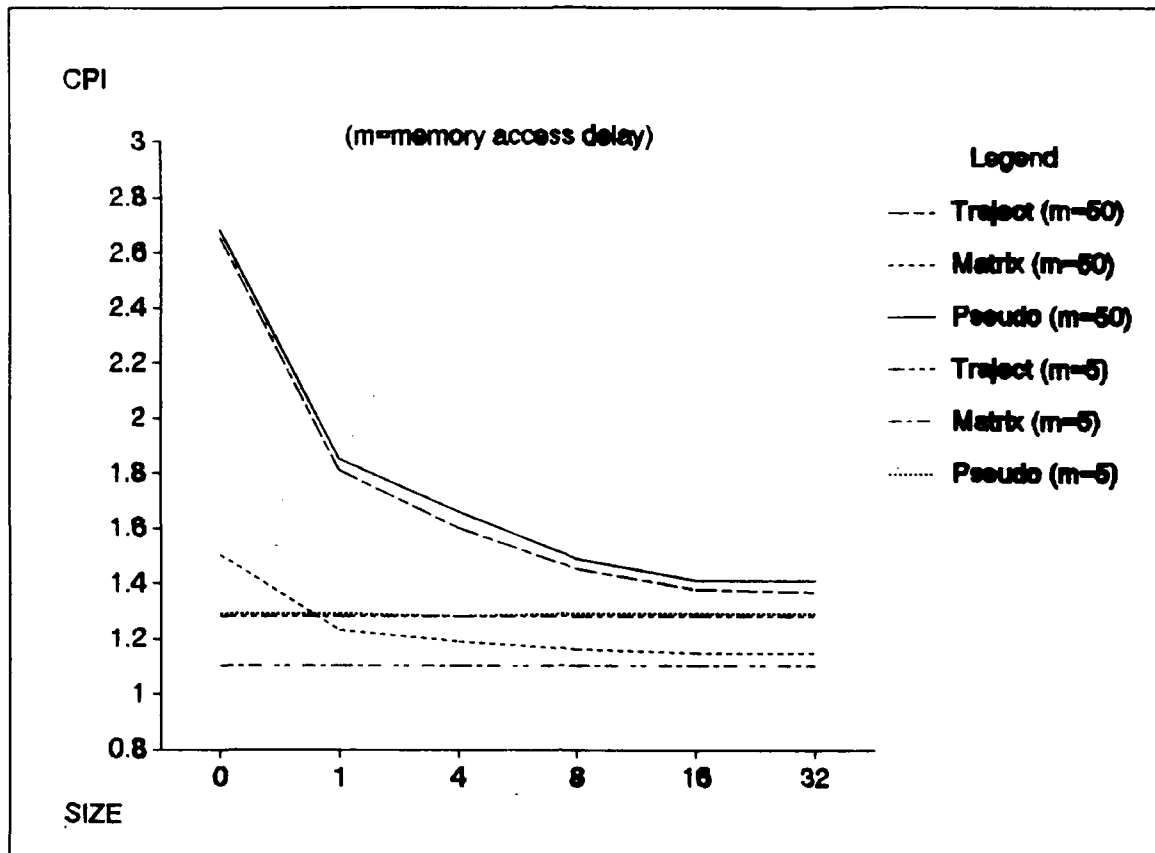
Figure 5.3 Effects of Separate-Queue-Miss-Priority MAQ Scheme

to 2.22 using the FIFO MAQ, 1.67 to 2.13 using the single-
queue-miss-priority MAQ, and 1.41 to 1.85 with the separate-
queue-miss-priority MAQ. For the Trajectory program with
m=50, the CPI values ranged from 1.70 to 2.07 (FIFO), 1.61 to
2.07 (single), and 1.37 to 1.81 (separate). The Matrix
Multiplication program with m=50 had CPI values of 1.35 to
1.42 (FIFO), 1.30 to 1.39 (single), and 1.15 to 1.23
(separate). Using the optimal queue size of 12 and m=50, the
separate MAQ scheme performed an average of 188% better than
not using a lockup-free cache; the single-queue-miss-priority

MAQ scheme performed an average of 148% better; and the FIFO MAQ performed an average of 135% better.

Using m=5, for the Matrix Multiplication program, CPI values ranged from 1.10 to 1.11 with the FIFO scheme, from 1.10 to 1.11 with the single-queue-miss-priority scheme, and unchanged at 1.10 throughout the different sizes with the separate-queue-miss-priority scheme. For the Pseudo Code program, the CPI ranges were 1.32 to 1.33 (FIFO), 1.32 to 1.33 (single), and unchanged at 1.29 for the separate scheme. The CPI results for the Trajectory program were 1.32 to 1.33 (FIFO), 1.32 to 1.33 (single), and unchanged at 1.28 for the separate scheme. We notice that there was little or no change in CPI using the different schemes for each of the programs. This is because with a small memory access penalty, the queue does not grow much and the turnaround time for dependent data is minimal, thus greatly reducing the chance of a memory delay stall.

Also with m=5, the separate MAQ scheme performed an average of 30% better than not using a lockup-free cache; the single-queue-miss-priority MAQ scheme performed an average of 20% better; and the FIFO MAQ performed an average of 19% better.

D. SUMMARY

In this chapter we have presented a high-level simulation to study the performance of a lockup-free cache interface on

a RISC architecture. The simulations provide indications of how various cache interface designs may perform. Overall, we found that the use of a lockup-free cache resulted in a performance improvement of up to nearly 200%.

One observation is that the size of the MAQ is a considerable factor for each of the designs. From a queue size of 0 to about 12, the CPI values improved. As the queue size exceeded farther past 12 there was little or no change in the CPI.

Another observation is that the design of the MAQ also had an effect on performance. Whereas each of the cache interface designs showed an improvement in CPI, the separate-queue-miss-priority MAQ configuration yielded the CPI values. This is probably attributed to the separate queues allowing both read misses and writes to have assured space. This is not the case with the single-queue-miss-priority queue where the MAQ may consist of all the same types of entries. The separate MAQ configuration may also further prevent a processor stall. For example, if the write MAQ is full and a read miss occurs, the processor will enqueue the read miss and continue processing if the read MAQ is not full.

Finally, we observed from the results of using the memory delays of 50 cycles and 5 cycles that the lockup-free cache interface is more effective as the memory access penalties increase. Also, as memory access penalties decrease, the optimal size of the MAQ also decreases. That fact emphasizes

that the need for a lockup-free cache interface will grow as the discrepancy between CPU speed and main memory access penalty grows. Memory access penalty in future systems are expected to exceed 140 cycles [Jou90].

VI. CONCLUSIONS

A. OVERVIEW

In this thesis we presented a study on the RISC architecture and its unique features. Our emphasis was on the effects on RISC performance of a lockup-free cache interface. The lockup-free cache interface features a queue to hold memory requests, allowing processing of program to continue while the memory requests are being served. We accomplished this through simulating the execution of actual programs.

To simulate and analyze the performance of a RISC and a lockup-free cache interface, we used several tools: SPA 1.0, SATTA, and RICIS. SPA is an available set of tools used to trace and analyze the execution of programs. We developed the SATTA to transform the address trace generated by SPA to detailed, more readable instruction records, and to produce modified address trace files for the RICIS. We developed RICIS to simulate and measure the effects of a lockup-free cache interface.

We examined various alternative schemes of the cache interface. The major design issues addressed were: (1) the size of the memory request queue, (2) whether to use a FIFO policy or one based on an assigned priority, and (3) whether

to use a single queue to hold both reads misses and writes in the same queue or to store them in separate queues.

Results of the experiments showed that the lockup-free cache considerably improved the performance of the RISC. The queue scheme and the queue size each had considerable effects on the performance of the cache interface. The separate-queue-miss-priority scheme yielded the best performance results, followed by the single-queue-miss-priority, and then the FIFO queue. For each scheme, the performance improved as the queue size went from 0 to about 12, after which the performance remained virtually unchanged. The greatest improvement was noted as the queue size advanced from 0 to 1. A queue size of zero has the same effects as having not used a queue in the lockup-free cache interface design.

B. FUTURE RESEARCH

The area that requires most emphasis for continued work is the simulation environment. The RICIS can be modified to simulate the fetching and executing of instructions out-of-program-order. This would determine if further processor stalling can be prevented by enqueuing instructions that are dependent on those instructions in the memory request queue. The dependent instructions are re-issued as the dependency problem is resolved. Meanwhile, the fetching and execution of new instructions continues. This feature is only partly implemented in RICIS as the problem of the dependency posed by

new instructions that are dependent on those instructions in the instruction queue needs addressing.

Another area for future research is the simulation of floating point instructions. Later versions of SPA may provide this feature. Minor modifications to SATTA and RICIS would then be required. Finally, the user interface for both the SATTA and the RICIS could be improved. An interface that combines the three simulation tools would greatly improve the user environment and conserve considerable computer resources, thus allowing experiments using larger, more reliable benchmarks.

APPENDIX A. USING THE SPA 1.0 SIMULATOR

The SPY Component: Address trace generator.

In tracing the execution of a program, the SPY can pass the address trace directly to a trace analyzer or to a file. For instance, the command

    % SPY *myprog*

traces the execution of the (executable) file *myprog* and passes the results to a file of the format *progname.pid.invocation_number*. The command

    % SPY -p 'spanner | spout' /usr/ls

generates an address trace of the *ls* command and passes it to the SPANNER program. The SPANNER then performs its functions and pipes the results to the SPOUT. The SPOUT display the results of the trace. The -p option directs SPY to pass the results to the analyzer. Similarly, the command

    % SPY -p spanner CC *myprog.C* -o *myprog*

traces the execution of the *CC* command and passes the results to SPANNER.

The SPANNER Component: SPARC instruction analyzer.

The SPANNER program reads the address trace file generated by the SPY program and compiles instruction count information. This information includes the number of times each type of

instruction was executed, and the number of cycles taken by each type of instructions. SPANNER also calculates and the number of cycles consumed by simulated cache misses, and it provide numerical data concerning conditional branches and window handlers.

SPANNER offers the user various options on system configurations as show in Figure A.1. The -c option lets the user choose the type of cache memory system to use for the simulation. The choices are the SS1, used in the SPARCstation 1, and the SS2 used in the SPARCstation 2. The -p option specifies the type of processor to simulate. The choices are the MB86901 processor, and the Cypress CY7C601.

```
spanner [-c cache] [-p processor] [-on]
        [-un ] [-rn] [-wn] filename
```

Figure A.1   SPANNER Command Format

The other options allow the user to set the specific number of cycles or the specific size for particular events. The -o option lets the user specify the number of cycles consumed by a register window overflow. The -u option sets the number of cycles for a window underflow. The -r lets the user set the interval, in cycles, to view interim output of the trace results. The -w option specifies the number of register windows to be simulated.

66

Default values are set by the SPANNER for each of the options. The default values closely resemble the features and characteristics of the SPARCstation 2. The SS2 is the default cache, the CY7C601 is the default processor, 170 cycles is the default for the register window overflow, 110 cycles is the default underflow cost, and the default for the number of register windows is 8.

The SPOUT Component - Instruction Count Tables Generator

The SPOUT component formats and displays tables of instruction count data obtained from the SPANNER. A discription of the simulated architecture configuration is displayed in the report heading as shown in Figure A.2.

```
Spanner - SPARC Performance Analyzer
     cpu:                     cy7c601
     cache:                       ss2
     register windows:              8
     overflow cost:         170 cycles
     underflow cost:        110 cycles
```

Figure A.2 SPOUT Report Heading and Parameter Settings

Figure A.3 is a table from a SPOUT report which shows an overview of the instruction and cycle count of the program trace. This table shows that 65.9% of all the cycles taken up by this program execution was taken up executing

instructions, 2.1% was taken up by window handlers, and 25.3%
by cache cycles.

```
    OVERALL             overall (%)    category (%)      raw
                        cycles  inst.  cycles  count  cycles  count
    instructions        65.9    100.0  65.9    -      187588  148554
    annulled slots      2.0     3.7    2.0     -      5501    5501
    load-use stalls     4.6     8.8    4.6     -      13024   13024
    trap cycles         0.1     0.0    0.1     -      284     71
    window handlers     2.1     0.0    2.1     -      6050    43
    cache cycles        25.3    2.1    25.3    -      71330   3147
    ----------------------------------------------------------------
    total               100.0   -      100.0   -      281977  -
```

Figure A.3 SPA Overall Instruction Count Listing

Figure A.4 is another table from the same SPOUT report.
This table displays the trace data of memory access
instructions only. Here, we see that 19.7% of the cycles
required to execute the program was consumed by load
instructions, and load instructions account for 86.6% of the
memory access cycles. Other information in this table is
18.7% of all the total number of instructions traced were
loads, as with 90.7% of all the memory access instructions.
The *raw* data column shows that there were 27720 load
instructions traced, consuming 55440 machine cycles.

SPOUT reports also contain similarly formatted tables of
data for each SPARC instruction, each window size, each type
of cache, and control transfers. The SPOUT report provides
the user with the data to determine what instructions, events,
and configurations have the greater effects on the
architecture performance. Also, CPI values can easily be

MEMORY ACCESS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
load	19.7	18.7	86.6	90.7	55440	27720
store	3.0	1.9	13.4	9.3	8556	2852
atomic	0.0	0.0	0.0	0.0	0	0
total	22.7	20.6	100.0	100.0	63996	30572

Figure A.4 SPOUT Memory Access Instruction Count

obtained and compared by dividing the *raw cycles* value by the corresponding *raw instructions* value.

69

APPENDIX B.   USING THE RICIS PROGRAM

Obtaining Address Traces.

In order to use the RICIS, an address trace of a executable program must first be obtained. This address trace can be produced using the SPA 1.0 package, as explained in Appendix A. After obtaining a trace file from SPA, a modified version of trace must be produced for use explicitly by the RICIS. This trace is produced by the SPARC Address Trace Transformer/Analyzer (SATTA) tool.

To produce the modified trace using SATTA, simply type the command *SATTA* at the command line prompt (%). The program will then produce the trace file, naming it *RICIS.FIL*. The user may rename the file to a more suitable file name after the file is generated.

The User Interface

To begin a RICIS session, type in the command *RICIS* at the command line prompt (%). The program then asks the user a series of questions to define the parameters and scheme of the lockup-free cache interface to simulate. Figure B.1 is an example of a start-up session for RICIS. The first input to the system is the address trace file name. Again, this is the file produced by SATTA. The next input is the cache hit ratio

to be simulated. The value entered must be represented as a percentage from 0.0 to 100.0. For example, to enter a hit ratio of 0.9, the user must enter 90.0.

```
$ RISIC

Enter name of file to parse
 rocket.luf

Enter simulated CACHE HIT RATE:    90.0

Simulate FIFO or Priority Queue? (f/p) : f

Enter Memory Queue Size:   5

Do you want to use Dependent Instruction Queue? (y/n) : n

Enter number of stall cycles for Load dependency: 1

Do you want to view Queues after every activity? (y/n)..n

Enter interval value for viewing Queues: 600000

Do you want to continue wit  simulation?   (y/n): n


Do you want to do another simulation? (y/n): y

Keep same parameters? (y/n): y
```

Figure B.1 RICIS Startup Session

The next input deals with the type of queue to simulate. The user must enter the character *f* for FIFO simulation, and *p* for priority queue simulation. If FIFO is chosen, the next input is the size of the queue to simulate. This value must be an integer between 0 and 50. However, if the priority queue is chosen, the user must chose whether to simulate a single queue or seperate queues for reads and writes. If the seperate queue is chosen, then the user must enter the size of

each queue. If the single-priority queue is chosen, the user enters the size of the queue.

Also, if a priority queue is chosen, the user must enter a priority number for a read and for a write. For example, if a read is to have priority over a write, then a value of 0 is enter for *Read Priority* and a 1 for *Write Priority*. The next question deals with the Dependent Instruction Queue. The user must enter the value *n* for this response, as this feature is **not** fully implemented at this time.

The user may elect to see a cycle-by-cycle trace of the simulation. If the user wants to see only the CPI results, the he/she must enter an *n*. The last start-up input to the simulation is the interval in number of instructions in which to view interim results. If the user wants only to see the final results, then a very large number must be entered. Since traces files contain hundreds of thousands of instrucion records, a value of 1,000,000 may do it. The user can obtain the number of instructions from running the original trace through the SATTA program.

At each interim result pause, the user is given the option of terminating the session or continuing. This feature is particularly useful for long, slow sessions. After each session terminates, the user is given a choice of conducting another session. If the user chooses to do another simulation, he/she may choose to keep the same parameter as the most previous session, or to enter new ones.

RICIS Output.

The follow is an example of RICIS results output:

```
Number of Instructions Executed:   642321
Number of Cycles Elapsed:          834232
CPI Value:                         1.30
```

The first line of the output shows the total number of instructions issued from the trace file. The second line shows the total number of (simulated) cycles consumed by the program execution. The last line shows the CPI value, attained by dividing the total cycles by the total instructions. For interim results, the values shown would be the results up to the instruction count.

APPENDIX C. SPARC ADDRESS TRACE TRANSFORMER/ANALYZER

```
//*******************************************************************************
// Title:     SPARC ADDRESS TRACE TRANSLATOR/ANALYZER (SATTA)
// Author:    Leonard Tharpe, Captain, U.S. Army
// Date:      September 1992
// Revised:
// Description: This program simulates a cycle-by-cycle execution of a program using using
// the the Sun 4 SPARC architecture.  The program takes as input a binary address trace of
// a compiled executable program and provides the user with a detailed instruction record for
// every cycle of program execution. This information includes the cycle/instruction number,
// the status of the instruction, a 32-bit representation of each instruction, the opcode,
// the location in memory of the instruction fetch, as well as the data fetch. This program
// also provides data and information on register use.  This information consists of the
// number of times each register is used as a source and as a destination, and it provides
// data on register dependency.
//*******************************************************************************
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

// ** This is the format of each instruction record produced by
//    the address trace.

struct Instruction {
    char et;
    char dav;
    short tn;
    unsigned long op;
    unsigned long ia;
    unsigned long da;
};

// ** This is the record format for register-use data.
struct reg_data {
    int source;
    int dest;
    int last_use;
    int lex_dist;
    int tot_dist;
    float avg_dist;
};

// ** This structure is used to hold and calculate register dependency data
struct dependency {
    int last_write;
    int ref_dist;
    int load_count;
    int ref_count;
    int tot_dist;
    int ld_used;
    float avg_dep_dist;
};

// ************************ Begin main program ************************
```

```
main()
{
        int ctr = 0;
        int loads = 0;
        int stores = 0;
        int avg_count = 0;
        float avg_dist = 0.0;
        float tot_dep = 0.0;
        unsigned long op_field;
        unsigned long op_value;
        unsigned long bicc_cond;
        unsigned long ticc_cond;
        unsigned long displ;
        unsigned long rd_value;
        unsigned long rs1_value;
        unsigned long rs2_value;
        unsigned long inst_hold;
        unsigned long index_bit;
        unsigned long annul_bit;
        unsigned long asi;
        unsigned long simm13;
        unsigned long imm22;
        unsigned long displ22;
        int ref_type;
        Instruction inst;
        reg_data reg_count[32];    // keeps data of register use
        dependency reg_dep[32];    // keeps data of register dependency

        // ---------- function definitions ----------------------
        void regcalc(reg_data& reg, int& count);
        void depcalc(dependency& reg, int& location, int& ref);
        void check_reg_dep(dependency& reg, int& count, int& ref);
        void clear_screen(int& i);
        void start_up(char *ifile, char *ofile);
        void instr_count(char *afile, Instruction& instr, int& icount);

        // **************** array of registers (notation) **************
         static char reg_sym[32] [5] =
              {"%g0","%g1","%g2","%g3","%g4","%g5","%g6","%g7",
               "%o0","%o1","%o2","%o3","%o4","%o5","%o6","%o7",
               "%l0","%l1","%l2","%l3","%l4","%l5","%l6","%l7",
               "%i0","%i1","%i2","%i3","%i4","%i5","%i6","%i7"
              };

        // ************* array of format 3, op = 11 opcodes ***************
         static char op11_inst[64] [10] =
              {"ld","ldub","lduh","ldd","st","stb",
               "sth","std","unimp","ldsb","ldsh","unimp",
               "unimp","ldstub","unimp","swap","lda","lduba",
               "lduha","ldda","sta","stba","stha","stda","unimp","ldsba",
               "ldsha","unimp","unimp","ldstuba","unimp",
               "swapa","ldf","ldfsr","unimp","lddf","stf","stfsr",
               "stdfq","stdf","unimp","unimp","unimp","unimp","unimp",
               "unimp","unimp","unimp","ldc","ldcsr","unimp","lddc",
               "stc","stcsr","stdcq","stdc","unimp","unimp","unimp",
               "unimp","unimp","unimp","unimp","unimp"};

        // ************* array of 3, format op = 10 opcodes ***************
         static char op10_inst[64] [10] =
              {"add","and","or","xor","sub","andn","orn","xnor","addx",
               "unimp","unimp","unimp","subx","unimp","unimp","unimp",
               "addcc","andcc","orcc","xorcc","subcc","andncc","orncc",
               "xnorcc","addxcc","unimp","unimp","unimp","subxcc","unimp",
```

75

```cpp
           "unimp","unimp","taddcc","tsubcc","taddcctv","tsubcctv","mulscc",
           "sll","srl","sra","rdy","rdpsr","rdwim","rdtbr","unimp","unimp",
           "unimp","unimp","wry","wrpsr","wrwim","wrtbr","fpop1","fpop2",
           "cpop1","cpop2","jmpl","rett","ticc","iflush","save","restore",
           "unimp","unimp"};

// ***************** array of Trap (ticc) opcodes ********************
 static char op10_ticc[16] [6] =
       {"tn","te","tle","tl","tleu","tcs","tneg","tvs","ta","tne",
        "tg","tge","tgu","tcc","tpos","tvc"};


// ************** array of format 2, op = 00 opcodes *****************
 static char op00_inst[8] [15] = {"unimp","unimp","bicc",
         "unimp","sethi","unimp","fbfcc","cbccc"};

// ************* array of branch condition opcodes ******************
 static char op001_inst[16] [15] = {"bn","be","ble","bl","bleu","bcs",
        "bneg","bvs","ba","bne","bg","bge","bgu","bcc","bpos","bvc"};

// ------------- initialize register data arrays -----------
for (int k = 0; k < 32; k++)
  {
   reg_count[k].source = 0;
   reg_count[k].dest = 0;
   reg_count[k].last_use = 0;
   reg_count[k].lex_dist = 0;
   reg_count[k].tot_dist = 0;
   reg_count[k].avg_dist = 0.0;
  };

// -------- initialize register dependency arrays ---------
for (int n = 0; n < 32; n++)
  {
   reg_dep[n].last_write = -1; // last line register written to
   reg_dep[n].ref_dist = -1;    // distance between reference and last write
   reg_dep[n].tot_dist = 0;
   reg_dep[n].load_count = 0;
   reg_dep[n].ref_count = 0;
   reg_dep[n].ld_used = 0;       // was register's last use a load?
   reg_dep[n].avg_dep_dist = -1.0;
  };

ofstream recfile("records.dat");
ofstream cachefile("RICIS.FIL");
fstream infile;
float load_percent;
char cont;
char view_reg;
char view_dep;
char nop = 'n';
int i, interval;
int ic;
int l_i = 0;  // load instruction 0 = no, 1 = yes
int s_i = 0;  // store instuction
char file1[20],file2[20];
clear_screen(i);
start_up(file1,file2);
instr_count(file1,inst,ic);
cout << "\nEnter cycle intervals to view output: ";
cin  >> interval;
ofstream asmfile(file2);
// --- cout << "file pointer declared " << '\n' << flush;
```

```
infile.open(file1,ios::in|ios::nocreate);
// --- cout << "file is now open " << '\n' << flush;
infile.seekg(ctr*sizeof(inst),ios::beg);
// --- cout << "seek invoked " << '\n' << flush;

// ** Read executable address trace until end of file reached.
while (infile.read((char *) (&inst), sizeof(inst)))   {
  recfile << "record : " << dec << ctr << '\n' << flush;
  recfile << "exec status : " << int(inst.et)   << '\n' << flush;
  recfile << "valid addr : " << int(inst.dav) << '\n' << flush;
  recfile << "trap no. : " << short(inst.tn) << '\n' << flush;
  recfile << "instruction : ";
  recfile << int(inst.op) << '\n' << flush;
  recfile << "binary representation : ";
  // -------- print the bit representation of opcode ------
  int i, mask;
  mask = 1;
  mask <<= 31;
  inst_hold = inst.op;
  op_field = inst.op;
  annul_bit = inst.op;
  for (i = 1; i <= 32; ++i)
   {
        recfile << (((int(inst.op) & mask) == 0) ? '0' : '1');
        inst.op <<= 1;
   }
// recfile << '\n';
// ----------------------------------------------------

// ------------- extract "op" field -------------------
op_field = (op_field >> 30);
recfile << "\nop_field : " << int(op_field) << '\n' << flush;
// ----------------------------------------------------

// ------------- examine format 1 - op = 01 ------------
if (op_field == 1)
  {
   displ = inst_hold & 0x3fffffff;
   recfile << "opcode : call " << '\n' << flush;
   recfile << "displacement : " << int(displ) << '\n' << flush;
   asmfile << setw(5) << ctr << ":     ";
   asmfile << setw(-10) << "call   ";
   asmfile << int(displ) << '\n' << flush;
   cachefile << "  2   " << setfill('0');
   cachefile << setw(8) << hex << inst.ia;
   cachefile << "             " << '\n';
  };

// ------------- examine format 2 - op = 00 -------------
if (op_field == 0)
  {
   // --------------- extract opcode field ----------------
   nop = 'n';
   op_value = (inst_hold & 0x1c00000) >> 22;
   recfile << "opcode value : " << int(op_value) << '\n' << flush;
   // --- check for bicc instructions -------
   if (op_value == 2)
     {
       bicc_cond = (inst_hold & 0x1e000000) >> 25;
       recfile << "opcode : " << op001_inst[bicc_cond] << '\n' << flush;
     };

   // ---- check for 'nop' instruction -----
```

77

```cpp
if (strncmp("sethi",op00_inst[op_value],5) == 1)
    recfile << "opcode :   "  << op00_inst[op_value] << '\n' << flush;
else
  nop = 'y';   // this instruction is a potential 'nop'
// ------------------------------------------------------


// ------- check annul bit --------------
annul_bit = (annul_bit >> 29);

if (annul_bit == 0)
 {

    // --------------- extract rd register field ------------
    rd_value = (inst_hold & 0x3e000000) >> 25;
    recfile << "rd value : " << int(rd_value) << '\n' << flush;
    if (nop == 'n')
    {
      reg_count[rd_value].dest += 1;
      regcalc(reg_count[rd_value], ctr);
    };
    // ----------------------------------------------------

    // --------------- extract immediate address -----------
    imm22 = inst_hold & 0x3fffff;
    if (((nop == 'y') && (imm22 == 0)) && (rd_value == 0))
    {
      recfile << "opcode: nop " << '\n' << flush;
    }
    else
     {
      if (op_value != 2)
       recfile << "opcode :   "  << op00_inst[op_value] << '\n' << flush;
      recfile << "immediate address : " << hex << imm22 << '\n' << flush;
      reg_count[rd_value].dest += 1;
      regcalc(reg_count[rd_value], ctr);
      cachefile << "  2  " << setfill('0')
                << setw(8) << hex << imm22 << "          "
                << setw(2) << hex << rd_value << '\n';
      nop = 'n';
     };
    // ----------------------------------------------------

    asmfile << setw(5) << ctr << ":    ";
    if (((strncmp("sethi",op00_inst[op_value],5) == 0)
        && (imm22== 0)) && (rd_value == 0))
    {
      asmfile << setw(-10) << "nop " << '\n' << flush;
    }
    else
    {
      if (op_value == 2)
        asmfile << setw(-10) << op001_inst[bicc_cond] << "    ";
      else
        asmfile << setw(-10) << op00_inst[op_value] << "    ";
      asmfile << imm22 << ",";
      asmfile << reg_sym[rd_value] << '\n' << flush;
    };
  }
else  // annul bit is set
 {
   bicc_cond = (inst_hold & 0x1e000000) >> 25;
   recfile << "annulled " << '\n' << flush;
```

```
        recfile << "condition: " << op001_inst[bicc_cond] << '\n' << flush;
        // --------------- extract displacement value -------------
        displ22 = inst_hold & 0x3fffff;
        recfile << "displacement value : " << dec << displ22 << '\n' << flush;
        // -------------------------------------------------------

        asmfile << setw(5) << ctr << ":    ";
        asmfile << setw(-10) << op001_inst[bicc_cond] << "    ";
        asmfile << displ22 << "    ";
        asmfile << "(annulled)" << '\n' << flush;
        cachefile << "   3   " << setfill('0')
                  << setw(8) << hex << inst.ia
                  << "              " << '\n';
    }
};

// -------------- examine format 3 - op = 10 ----------------
if (op_field == 2)
  {
  // --------------- extract opcode field ----------------
  op_value = (inst_hold & 0x1f80000) >> 19;
  recfile << "opcode value : " << int(op_value) << '\n' << flush;
  recfile << "opcode :   " << op10_inst[op_value] << '\n' << flush;

  // ---- check for ticc instruction --------------------
  if (op_value == 58)
    {
      ticc_cond = (inst_hold & 0x1e000000) << 3;
      ticc_cond = ticc_cond >> 28;
      recfile << "ticc cond : " << op10_ticc[ticc_cond] << '\n' << flush;
    };

  // ------------------------------------------------------- .

  // --------------- extract rd register field -------------
  rd_value = (inst_hold & 0x3e000000) >> 25;
  recfile << "rd value : " << int(rd_value) << '\n' << flush; ;
  reg_count[rd_value].dest += 1;
  regcalc(reg_count[rd_value], ctr);
  // ------------------------------------------------------

  // ----------- extract rs1 register field -------------
  rs1_value = (inst_hold & 0x7c000) >> 14;
  recfile << "rs1 value : " << int(rs1_value) << '\n' << flush;
  ref_type = 0;         // reference type is a read
  depcalc(reg_dep[rs1_value], ctr, ref_type);
  reg_count[rs1_value].source += 1;
  regcalc(reg_count[rs1_value], ctr);
  // ------------------------------------------------------

  // ---------------- extract index bit -----------------
  index_bit = (inst_hold & 0x2000) >> 13;
  recfile << "index_bit : " << int(index_bit) << '\n' << flush;
  // ------------------------------------------------------

  if (index_bit == 0)
    {
    // ----------- extract rs2 register field -------------
    rs2_value = inst_hold & 0x1f;
    recfile << "rs2 value : " << int(rs2_value) << '\n' << flush;
    ref_type = 0;   // reference type is a read
    depcalc(reg_dep[rs2_value], ctr, ref_type);
    reg_count[rs2_value].source += 1; ;
```

```
    regcalc(reg_count[rs2_value], ctr);
    // ------------------------------------------------------
    asmfile << setw(5) << ctr << ":    " << setfill(' ');
    if (op_value == 58)
     asmfile << setw(-10) << op10_ticc[ticc_cond] << "    ";
    else
     asmfile << setw(-10) << op10_inst[op_value] << "    ";
    asmfile << reg_sym[rs1_value] << ",";
    asmfile << reg_sym[rs2_value] << ",";
    asmfile << reg_sym[rd_value] << '\n' << flush;
    cachefile << "  2   " << setfill('0')
              << setw(8) << hex << inst.ia
              << "    " << hex << setw(2) << rs1_value
              << "    " << hex << setw(2) << rs2_value
              << "    " << hex << setw(2) << rd_value << '\n';

    }
  else  // index bit is set
    {
    simm13 = inst_hold & 0x1fff;
    recfile << "simm13 : " << int(simm13) << '\n' << flush;
    asmfile << setw(5) << ctr << ":    ";
    if (op_value == 58)
     asmfile << setw(-10) << op10_ticc[ticc_cond] << "    ";
    else
     asmfile << setw(-10) << op10_inst[op_value] << "    ";
    asmfile << reg_sym[rs1_value] << ",";
    asmfile << int(simm13) << ",";
    asmfile << reg_sym[rd_value] << '\n' << flush;
    cachefile << "  2   " << setfill('0')
              << setw(8) << hex << simm13
              << "    " << setw(2) << rs1_value << "    "
              << "    " << setw(2) << hex << rd_value << '\n';

    }
};

// -------------- examine format 3 - op = 11 ---------------
if (op_field == 3)
  {
  // --------------- extract opcode field ---------------
  op_value = (inst_hold & 0x1f80000) >> 19;
  recfile << "opcode value : " << int(op_value) << '\n' << flush;
  recfile << "opcode :   " << op11_inst[op_value] << '\n' << flush;
  // ----------------------------------------------------

  // --------------- extract rd register field -----------
  rd_value = (inst_hold & 0x3e000000) >> 25;

  // ---------- isolate load instructions ---------------
  l_i = 0;
  if (strncmp("ld",op11_inst[op_value],2) == 0)
    {
    l_i = 1;
    loads += 1;
    ref_type = 1;   // reference type is a load
    reg_dep[rd_value].ld_used = 1;
    depcalc(reg_dep[rd_value], ctr, ref_type);
    };
  // ---------- isolate store instructions ---------------
  s_i = 0;
  if (strncmp("st",op11_inst[op_value],2) == 0)
    {
    s_i = 1;
    stores += 1;
```

```
    };

  recfile << "rd value : " << int(rd_value) << '\n' << flush;
  reg_count[rd_value].dest += 1;
  regcalc(reg_count[rd_value], ctr);
  // ----------------------------------------------------------

// ------------ extract rs1 register field -------------
rs1_value = (inst_hold & 0x7c000) >> 14;
recfile << "rs1 value : " << int(rs1_value) << '\n' << flush;
ref_type = 0;
depcalc(reg_dep[rs1_value], ctr, ref_type);
reg_count[rs1_value].source += 1;
regcalc(reg_count[rs1_value], ctr);
// ----------------------------------------------------------

// ---------------- extract index bit ----------------
index_bit = (inst_hold & 0x2000) >> 13;
recfile << "index_bit : " << int(index_bit) << '\n' << flush;
// ----------------------------------------------------------

if (index_bit == 0)
  {
    // ------------ extract rs2 register field -------------
    rs2_value = inst_hold & 0x1f;
    recfile << "rs2 value : " << int(rs2_value) << '\n' << flush;
    ref_type = 0;
    depcalc(reg_dep[rs1_value], ctr, ref_type);
    reg_count[rs2_value].source += 1;
    regcalc(reg_count[rs2_value], ctr);
    // ----------------------------------------------------------

    // ------------ extract asi field -------------------
    asi = (inst_hold & 0x1fe0) >> 5;
    recfile << "asi value : " << int(asi) << '\n' << flush;
    // ----------------------------------------------------------
    asmfile << setw(5) << ctr << ":    ";
    asmfile << setw(-10) << op11_inst[op_value] << "   ";
    asmfile << reg_sym[rs1_value] << "+"
            << reg_sym[rs2_value] << ",";
    asmfile << reg_sym[rd_value] << '\n' << flush;
    if ((l_i == 0) && (s_i == 0))
      {
        cachefile << "  2   " << setfill('0')
                  << setw(8) << hex << inst.ia
                  << "  " << setw(2) << hex << rs1_value
                  << "  " << setw(2) << hex << rs2_value
                  << "  " << setw(2) << hex << rd_value << '\n';
      };
    if (l_i == 1)
      {
        cachefile << "  2   " << setfill('0')
                  << setw(8) << hex << inst.ia
                  << "  " << setw(2) << hex << rs1_value
                  << "  " << setw(2) << hex << rs2_value
                  << "  " << setw(2) << hex << rd_value << '\n';
        cachefile << "  0   " << setfill('0')
                  << setw(8) << hex << inst.da
                  << "          "
                  << setw(2) << hex << rd_value << '\n';
      };
    if (s_i == 1)
      {
```

```
            cachefile << "  2   " << setfill('0')
                      << setw(8) << hex << inst.ia
                      << "   " << setw(2) << hex << rs1_value
                      << "   " << setw(2) << hex << rs2_value
                      << "   " << setw(2) << hex << rd_value << '\n';
        cachefile << "  1   " << setfill('0')
                      << setw(8) << hex << inst.da
                      << "          "
                      << setw(2) << hex << rd_value << '\n';
      };
    }
  else  // index bit set
    {
    simm13 = inst_hold & 0x1fff;
    recfile << "simm13 : " << int(simm13) << '\n' << flush;
    asmfile << setw(5) << ctr << ":   " << setfill(' ');
    asmfile << setw(-10) << op11_inst[op_value] << "    ";
    asmfile << reg_sym[rs1_value] << "+" << int(simm13);
    asmfile << "," << reg_sym[rd_value] << '\n' << flush;
    if ((l_i == 0) && (s_i == 0))
      {
        cachefile << "  2   " << setfill('0')
                      << setw(8) << hex << inst.ia
                      << "   " << setw(2) << hex << rs1_value
                      << "        "
                      << setw(2) << hex << rd_value << '\n';
      };
    if (l_i == 1)
      {
        cachefile << "  2   " << setfill('0')
                      << setw(8) << hex << inst.ia
                      << "   " << setw(2) << hex << rs1_value
                      << "        "
                      << setw(2) << hex << rd_value << '\n';
        cachefile << "  0   " << setfill('0')
                      << setw(8) << hex << inst.da
                      << "        "
                      << setw(2) << hex << rd_value << '\n';
      };
    if (s_i == 1)
      {
        cachefile << "  2   " << setfill('0')
                      << setw(8) << hex << inst.ia
                      << "   " << setw(2) << hex << rs1_value
                      << "        "
                      << setw(2) << hex << rd_value << '\n';
        cachefile << "  1   " << setfill('0')
                      << setw(8) << hex << inst.da
                      << "        "
                      << setw(2) << hex << rd_value << '\n';
      };
    }
  };
    recfile << "inst addr : " << hex << inst.ia << '\n' << flush;
    recfile << "data addr : " << hex << inst.da << '\n' << flush;
    recfile  << "***************************" << '\n' << flush;

// ----------------------------------------------------------

  if ((ctr+1) % interval == 0)
    {
    cout << "\nDo you want to see register usage data? (y/n) ";
    cin >> view_reg;
```

```
    };
    if ((view_reg == 'y') && (((ctr+1) % interval) == 0))
    {
        cout << setiosflags(ios::left) << setw(10) << "Register"
            << setw(12) << "Register"
            << setw(12) << "Source"
            << setw(12) << "Destination"
            << setw(12) << "Average" << '\n';
        cout << setiosflags(ios::left) << setw(10) << "Number"
            << setw(12) << "Symbol"
            << setw(12) << "References"
            << setw(12) << "References"
            << setw(12) << "Lex. Distance" << '\n';
        for (int i = 0; i < 32; i++)
        {
          if ( i < 10 )
            cout << "R[" << int(i) << "]    ";
          else
            cout << "R[" << int(i) << "] ";
          cout << resetiosflags(ios::left)
              << setw(12) << dec << reg_sym[i]
              << setw(12) << dec << reg_count[i].source
              << setw(12) << reg_count[i].dest
              << setiosflags(ios::fixed)
              << setw(12) << setprecision(1)
              << reg_count[i].avg_dist << '\n';
        };
        cout << "\nNumber of instructions processed: " << ctr + 1 << '\n';
    };
    if ((ctr+1) % interval == 0)
    {
      cout << '\n' << ctr+1 << " instruction records processed so far !!!";
      cout << "\nDo you want to see register dependency data? (y/n) ";
      cin >> view_dep;
    };
    if ((view_dep == 'y') && ((ctr+1) % interval == 0))
    {
        cout << setiosflags(ios::left) << setw(10) << "Register"
            << setw(12) << "Register"
            << setw(12) << "    Load"
            << setw(12) << "    Source"
            << setw(12) << "Average"
            << setw(12) << "Percent" << '\n';
        cout << setiosflags(ios::left) << setw(10) << "Number"
            << setw(12) << "  Symbol"
            << setw(12) << "    Count"
            << setw(12) << "    Count"
            << setw(12) << "Distance"
            << setw(12) << "Load Use" << '\n';
      tot_dep = 0.0;
      avg_count = 0;
      for (int i = 0; i < 32; i++)
      {
        if ( i < 10 )
          cout << "R[" << int(i) << "]    ";
        else
          cout << "R[" << int(i) << "] ";
        cout << resetiosflags(ios::left);
        cout << setw(12) << dec << reg_sym[i];
        cout << setw(12) << dec << reg_dep[i].load_count;
        if (reg_dep[i].ref_count < 0)
          cout << "***";
        else
```

83

```cpp
                cout << setw(12) << reg_dep[i].ref_count;
            cout << setiosflags(ios::fixed);
            cout << setw(12) << setprecision(1);
            if (reg_dep[i].avg_dep_dist < 0.0)
                cout << "***";
            else
                cout << reg_dep[i].avg_dep_dist;
            if (reg_dep[i].load_count < 0)
                cout << "            ***" << '\n';
            else
                load_percent = float(reg_dep[i].load_count)/float(loads);
                load_percent = load_percent * 100.0;
                cout << setw(12) << setprecision(1);
                cout << load_percent << '\n',

            // ** calculate average dependency distance **
            if (reg_dep[i].avg_dep_dist > 0.0)
            {
                tot_dep += reg_dep[i].avg_dep_dist;
                avg_count++;
            };

        };
        avg_dist = tot_dep/avg_count;
        cout << "Average load dependency distance is: " << avg_dist << '\n';
        cout << "Total number of load instructions: " << loads << '\n';
        cout << '\n' << '\n';
        cout << "\nNumber of instructions processed: " << ctr+1 << '\n';
        cout << "\nPress any key to continue: ";
        cin >> cont;
    };
    ctr++;
    .infile.seekg(ctr*sizeof(inst),ios::beg);
    }
infile.close();
cout << "\nTotal number of instruction records in file is: " << ctr-1 << '\n';
}


void check_reg_dep(dependency& reg, int& count, int& ref)
{
   void depcalc(dependency& reg, int& location, int& ref);
   if (reg.ld_used == 1)
   {
     depcalc(reg, count, ref);
   };
}
// ------- this functions calculates register usage ata
void regcalc(reg_data& reg, int& count)
{
    reg.lex_dist = count - reg.last_use;
    reg.last_use = count;
    reg.tot_dist += reg.lex_dist;
    reg.avg_dist = float(reg.tot_dist)/float(reg.source + reg.dest);
}


// ------- this functions calculates register dependency data
void depcalc(dependency& reg, int& location, int& ref)
{
   if (reg.ld_used == 1)
   {
      if (ref == 1)
      {
       reg.last_write = location;
```

```
        reg.load_count++;
      }
      else
      {
        if (reg.last_write != -1)
          {
          reg.ref_count++;
          reg.ref_dist = location - reg.last_write;
          reg.tot_dist += reg.ref_dist;
          reg.avg_dep_dist = float(reg.tot_dist)/float(reg.ref_count);
          reg.ld_used = 0;
          reg.last_write = -1;
          };
      };
  };
}


// ----------------- this function clears the screen
void clear_screen(int& i)
{
  for (i = 1; i <= 26; ++i)
    cout << '\n';
}

// -------- this function accepts input from user to guide simulation
void start_up(char *ifile, char *ofile)
{
  cout << "\nEnter address trace input file: ";
  cin >> ifile;
  cout << "\nEnter assembly code output file: ";
  cin >> ofile;
  cout << "\nCounting instructions...please wait..." << '\n';
}

// -------- this function counts the total number of instructions in file
void instr_count(char *afile, Instruction& instr, int& icount)
{
    fstream ifile;
    icount = 0;
    ifile.open(afile,ios::in|ios::nocreate);
    ifile.seekg(icount*sizeof(instr),ios::beg);
    while (ifile.read((char *) (&instr), sizeof(instr))) {
      icount++;
      ifile.seekg(icount*sizeof(instr),ios::beg);
//        if ((icount+1) % 1000 == 0)
//          cout << '\n' << icount+1 << " records counted" << '\n';
    }
    ifile.close();
    cout << "\nTotal number of instruction records in file is: ";
    cout << icount << '\n';
}
```

APPENDIX D. RISC CACHE INTERFACE SIMULATOR (RICIS) CODE

```
-- ****************************************************************************
-- Thesis Project          :RISC Cache Interface Simulator (RICIS)
-- Author                  :Leonard Tharpe
-- Date                    :September 1992
-- System                  :UNIX
-- Compiler                :VERDIX Ada
-- Description             :This program is a simulation of a lockup-free
--     cache interface.  It simulates the fetching and execution of a program
--     trace.  The trace input files are generated by the SPARC Address Trace
--     Transformer/Analyzer (SATTA) program. This program uses a generic queue
--     package, along with random number generator and hexidecimal-to-decimal
--     conversion packages.
-- ********************* NOTICE !!!!!!!!!!!!!!! ***************************
--     Out-of-Order fetching/execution is "partially" implemented.  This feature
--     uses the Dependent Instruction Queue (DIQ).  All references to DIQ apply
--     to this feature.
-- ****************************************************************************

with TEXT_IO, QUEUES, RANDOM, HEX;
use TEXT_IO, RANDOM, HEX;

procedure SPLIT is

  package FLOAT_INOUT is new FLOAT_IO(FLOAT);
  use FLOAT_INOUT;
  package INTEGER_INOUT is new INTEGER_IO(INTEGER);
  use INTEGER_INOUT;

  -- This array defines the status of the registers used by the
  -- trace instructions.  TRUE means the register is ready for use,
  -- FALSE means the register is blocked and cannot be used.
  type REGISTER_STATUS is array (0..31) of BOOLEAN;

  -- The following is the format of the instruction from the address
  -- trace used by this program.  CODE indicates what type of instruction,
  -- ADDRESS indicates the address from which the instruction is taken, or
  -- to where the data is to be stored or retrieved from.
  type TRACE_RECORD is
    record
      CODE                :CHARACTER := ' ';
      ADDRESS             :STRING(1..8) := (others => ' ');
      SOURCE1_REGISTER    :STRING(1..2) := (others => ' ');
      SOURCE2_REGISTER    :STRING(1..2) := (others => ' ');
      TARGET_REGISTER     :STRING(1..2) := (others => ' ');
    end record;

  -- This record defines the format for entries in the Memory Access
  -- Queue (MAQ). The TRACE_LINE is the trace instruction, and the
  -- TIME_VALUE hold the priority assigned to each MAQ entry.
  type MAQ_RECORD is
    record
      TRACE_LINE          :TRACE_RECORD;
      TIME_VALUE          :INTEGER;
    end record;
```

```
-- This record defines the entries to the Priority Event Queue (PEQ).

type EVENT_RECORD is
  record
    EVENT_ID        :STRING(1..2) := (others => ' ');
    PRIORITY        :INTEGER := 0;
  end record;


-- Memory Access Queue
package MAQ is new QUEUES (ITEM => MAQ_RECORD);
use MAQ;


-- Dependent Instruction Queue
package DIQ is new QUEUES (ITEM => MAQ_RECORD);
use DIQ;


-- Priority Event Queue
package PEQ is new QUEUES (ITEM => EVENT_RECORD);
use PEQ;


-- Auxillary MAQ for viewing contents of the MAQ
package MAQ_VIEW is new QUEUES (ITEM => MAQ_RECORD);
use MAQ_VIEW;


-- Auxillary PEQ for viewing contents of the PEQ
package PEQ_VIEW is new QUEUES (ITEM => EVENT_RECORD);
use PEQ_VIEW;


-- MAQ for load instructions in a split-queue configuration
package Q0 is new QUEUES (ITEM => MAQ_RECORD);
use Q0;


-- MAQ for store instructions in a split-queue configuration
package Q1 is new QUEUES (ITEM => MAQ_RECORD);
use Q1;
```
-- ******************** variable declarations **************************

```
ANOTHER             :CHARACTER := 'y'; -- used to perform another simulation without re-
                                       -- running program
DIQ_FETCHED         :BOOLEAN := FALSE; -- a flag that indicate if an instruction was taken
                                       -- from the DIQ
DIQ_SIZE            :POSITIVE := 50;    -- the pre-set maximum size of the DIQ
DIQ_USED            :CHARACTER := 'n'; -- a flag that indicates if the DIQ was used in the
                                          simulation instead of stalling when an instruction
                                          depends on a queued request
BLANK               :STRING(1..2) := (others => ' '); -- used for checking a field to see if
                                                      -- a register is used
BLOCKED             :BOOLEAN := FALSE; -- a flag that indicates if an instruction must be
                                       -- blocked for dependency
BLOCKED_REGISTER    :REGISTER_STATUS := (others => FALSE); -- an array of 32 flags that
                                                           -- indicates whether a register
                                                           -- is blocked
BLOCKS              :DIQ.QUEUE(DIQ_SIZE); -- the name of the entries of the DIQ
BR_CPI              :INTEGER := 3; -- the number of cycles required for a branch instruction
EVENT               :EVENT_RECORD; -- a temporary store for a PEQ record
EVENTS              :POSITIVE := 100; -- the pre-set maximum size of a PEQ
EXECUTE_TIME        :INTEGER := 0; -- the cumulative execution time (in cycles) of a
                                   -- simulation session
FILE_NAME           :STRING(1..30) := (others => ' ');
FINISHED            :BOOLEAN := FALSE; -- this flag indicates that both the instruction file
                                       -- and PEQ are empty
HIT_RATE            :FLOAT := 90.0; -- % the percentage of load instructions that are cache
                                    -- hits
HOLD_VALUE          :INTEGER := 0; -- temporary storage for queue position counter
```

87

```
INPUT_FILE            :FILE_TYPE;  -- trace file from SATTA
INTERVAL              :INTEGER := 0;  -- the instruction count intervals in which to obtain
                                      interim results
-- LATENCY is the penalty assessed to read misses and writes
LATENCY               :INTEGER := 50;  -- preset main memory access penalty in cycles
LOAD_DEP              :INTEGER := 1;  -- this is the register number of an instruction that
                                      -- immediately follows a load instruction and
                                      -- references the destination register of the load
LOAD_REG              :INTEGER := 1; -- this is the destination register of a load instruction,
                                      -- cache hit or miss.
LOAD_SWITCH           :BOOLEAN := FALSE; -- this flag indicates that the issued instruction is
                                         -- a load and the next instruction's source
                                         -- register(s) must be checked for load dependency
MAIN_MEMORY           :MAQ_RECORD;       -- store for the main memory simulation.  Since only
                                         -- one instruction
                                         -- can be in main memory at any one time, only one
                                         -- MAQ record size is needed
MAQ_SIZE              :POSITIVE := 100  -- preset maximum size of the MAQ
MAQ_COUNT             :MAQ_VIEW.QUEUE(MAQ_SIZE); -- entry names of a temporary MAQ used to view
                                                 -- contents
MAQ_FULL              :BOOLEAN := FALSE;  -- flag indicating that MAQ is full
MAQ_LENGTH            :INTEGER := 0;     -- current size of the MAQ.
NAME_LENGTH           :INTEGER := 0;     -- length of filename entered by user.  Derived by
                                          -- interal function
PEQ_COUNT             :PEQ_VIEW.QUEUE(PEQ_SIZE); -- entry names of tempory PEQ to view contents
                                                 -- of PEQ
PEQ_SIZE              :POSITIVE := 200; -- the pre-set maximum size of the PEQ
Q_READS               :INTEGER := 0; -- the number of read or load instructions in the MAQ
Q_WRITES              :INTEGER := 0; -- the number of write or store instructions in the MAQ
Q0_SIZE               :POSITIVE := 20; -- the pre-set maximum size of the Q0
Q1_SIZE               :POSITIVE := 20; -- the pre-set maximum size of the Q1
READ_PRI              :INTEGER := 0;  -- the priority value assigned to read or load misses
READS                 :Q0.QUEUE(Q0_SIZE); -- the name of the elements in Q0
RECORD_COUNTER        :INTEGER := 0; -- accumulates the number of instructions read from address
                                      -- trace file
RECORD_REMOVED        :MAQ_RECORD; -- temporary storage for elements removed from the MAQ
REQUESTS              :MAQ.QUEUE(MAQ_SIZE);
RESPONSE              :CHARACTER := 'y'; -- used to get user yes/no response
SAME_DATA             :CHARACTER := 'n'; -- for using the same parameters of a previous session
SEPERATE_Q            :CHARACTER := 'n';  -- value that determines whether to use a single or
                                          -- seperate MAQ scheme
TOTAL_CYCLES          :INTEGER := 0; -- accumulates the total number of cycles of a session
TOTAL_PENALTY         :INTEGER := 0; -- accumulates the total cost of main memory access during
                                      -- a session
TRACE                 :MAQ_RECORD; -- hold data in the format of an MAQ entry
TRACE_REC             :TRACE_RECORD; -- formatted store of line from the address trace file
TYPE_Q                :CHARACTER := 'f'; -- the MAQ scheme to simulate: 'f'-FIFO, 'p' - Priority
UPDATE_RECORD         :MAQ_RECORD;  -- temporary store for MAQ entry
VIEW                  :CHARACTER := 'n';  -- value that determines whether to show cycle-by-cycle
                                          -- simulation
WAITING               :PEQ.QUEUE(EVENTS);  -- the name of the elements in the PEQ
WRITE_PRI             :INTEGER := 0;  -- the priority value assigned to write or store
                                      -- instructions
WRITES                :Q1.QUEUE(Q1_SIZE); -- the name of the elements in Q1
```

```
-- This procedure clears the CRT screen
procedure CLEARSCREEN is
begin
  PUT(ASCII.ESC);
  PUT("[2J");
end CLEARSCREEN;
```

```
-- This procedure reads in file name and opens the input file
```

```
procedure GET_INPUT_FILE(INPUT_FILE        :in out FILE_TYPE) is
   FILE_NAME                                :STRING(1..30);
   NAME_LENGTH                              :INTEGER;
begin
   PUT_LINE("Enter name of file to parse ");
   GET_LINE(FILE_NAME,NAME_LENGTH);
   OPEN(INPUT_FILE, MODE => IN_FILE, NAME => FILE_NAME(1..NAME_LENGTH));
   ------------------------------------------------------------
   --   The simulator is initiated by inserting the first event into
   --   the PEQ.  The first event being to issue an instruction (ii).
   PEQ.CLEAR(WAITING);
   EVENT.EVENT_ID := "ii";
   EVENT.PRIORITY := 0;
   PEQ.ADD(EVENT,WAITING);
end GET_INPUT_FILE;


-------------------------------------------------------------------
-- This procedure allows the user the option of viewing the cycle-
-- by-cycle transactions of the simulation of to execute
-- without displaying transactions. It also allows the user
-- to view interim results.
procedure GET_VIEW_METHOD  is

begin

   NEW_LINE;
   PUT("Do you want to view Queues after every activity ? (y/n)..");
   GET(VIEW);
   NEW_LINE;
   PUT("Enter interval value for viewing Queues: ");
   GET(INTERVAL);
   NEW_LINE;

 end GET_VIEW_METHOD;


-----------------------------------------------------------------
-- This procedure lets the user set the parameters for the
-- target configuration.
procedure GET_INITIAL_DATA  is

begin
   NEW_LINE;
   PUT("Enter simulated CACHE HIT RATE:    ");
   GET(HIT_RATE);
   NEW_LINE;
   PUT("Simulate FIFO or Priority Queue? (f/p) : ");
   GET(TYPE_Q);
   NEW_LINE;
   PUT("Enter Memory Queue Size:   ");
   GET(MAQ_SIZE);
   NEW_LINE;
   if TYPE_Q = 'p' then
     PUT("Use seperate memory queues for Reads and Writes? (y/n) : ");
     GET(SEPERATE_Q);
     if SEPERATE_Q = 'y' then
       PUT("ENTER READ QUEUE SIZE: ");
       GET(Q0_SIZE);
       NEW_LINE;
       PUT("ENTER WRITE QUEUE SIZE: ");
       GET(Q1_SIZE);
       NEW_LINE;
     end if;
```

89

```
        PUT("Enter Read Priority (0 to 1):   ");
        GET(READ_PRI);
        NEW_LINE;
        PUT("Enter Write Priority (0 to 1):   ");
        GET(WRITE_PRI);
        NEW_LINE;
      end if;
      NEW_LINE;
      PUT("Do you want to use Dependent Instruction Queue? (y/n) : ");
      GET(DIQ_USED);
      NEW_LINE;
      PUT("Enter number of stall cycles for Load dependency: ");
      GET(LOAD_DEP);
      GET_VIEW_METHOD;
      MAQ.CLEAR(REQUESTS);
   end GET_INITIAL_DATA;


-------------------------------------------------------------------
   -- This procedure parses the lines from the address trace file.  Each
   -- line represents an instruction.  The instruction is broken down
   -- into components.
   procedure GET_FIELDS(PARSE_LINE              :in out STRING;
                        NR_OF_CHARS_IN_LINE :in out INTEGER;
                        TRACE_REC               :in out TRACE_RECORD) is

      VALID_ADDRESS,
      VALID_CODE                          :BOOLEAN := FALSE;
   begin
      TRACE_REC.CODE := PARSE_LINE(3);
      TRACE_REC.ADDRESS := PARSE_LINE(6..13);
      TRACE_REC.SOURCE1_REGISTER := PARSE_LINE(16..17);
      TRACE_REC.SOURCE2_REGISTER := PARSE_LINE(20..21);
      TRACE_REC.TARGET_REGISTER := PARSE_LINE(24..25);
   end GET_FIELDS;


-------------------------------------------------------------------
   -- Parses the lines read in from the input file.
   procedure DO_LINE_PARSING(INPUT_FILE   : in out FILE_TYPE;
                             TRACE_REC     :in out TRACE_RECORD) is

      PARSE_LINE              :STRING(1..80) := (others => ' ');
      NR_OF_CHARS_IN_LINE   :INTEGER;

   begin
      GET_LINE(INPUT_FILE,PARSE_LINE,NR_OF_CHARS_IN_LINE);
      GET_FIELDS(PARSE_LINE,NR_OF_CHARS_IN_LINE,TRACE_REC);
      TRACE.TRACE_LINE := TRACE_REC;
   end DO_LINE_PARSING;


-------------------------------------------------------------------
   -- This procedure is a viewing option for the users, allowing the
   -- viewing of the status of each register.
   procedure VIEW_REGISTER_STATUS is

      COL          :INTEGER;
      begin
      PUT(" --------------------------------------------------------------");
      NEW_LINE;
      PUT("| 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15 |");
      NEW_LINE;
      for I in 0..15 loop
         if BLOCKED_REGISTER(I) then
            PUT("   x");
```

90

```
        else
          PUT("      ");
        end if;
      end loop;
      NEW_LINE;
      PUT("------------------------------------------------------------");
      NEW_LINE;
      PUT("| 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 |");
      NEW_LINE;
      for I in 16..31  loop
        if BLOCKED_REGISTER(I) then
          PUT("   x");
        else
          PUT("    ");
        end if;
      end loop;
      NEW_LINE;
      PUT("------------------------------------------------------------");
      NEW_LINE;

  end VIEW_REGISTER_STATUS;

----------------------------------------------------------------------
-- This procedure allows the viewing of the contents of the PEQ.
procedure VIEW_MAQ is

    LENGTH                          :INTEGER;
    MAQ_HOLD                        :MAQ_RECORD;
    VIEW                            :CHARACTER := 'n';
    KEY                             :CHARACTER;
  begin

--**************** VIEW OUTSTANDING MEMORY REQUEST QUEUE  ********************

    NEW_LINE;
    LENGTH := MAQ.LENGTH_OF(REQUESTS);
    if SEPERATE_Q = 'n' then
      SET_COL(20);
      PUT_LINE("***************MAQ****************");
      SET_COL(20);
      PUT_LINE("*  CODE         ADDRESS     PRIORITY* ");
      SET_COL(20);
      PUT_LINE("*------------------------------------*");
      PUT("MAIN MEMORY ->");
      for I in 1..LENGTH loop
        MAQ.REMOVE(MAQ_HOLD,REQUESTS);
        SET_COL(20);
        PUT("*      ");
        PUT(MAQ_HOLD.TRACE_LINE.CODE);
        PUT("          ");
        PUT(MAQ_HOLD.TRACE_LINE.ADDRESS);
        PUT("        ");
        PUT(MAQ_HOLD.TIME_VALUE, WIDTH => 3);
        PUT_LINE("    * ");
        MAQ.ADD(MAQ_HOLD,REQUESTS);
        SET_COL(20);
        PUT_LINE("************************************");
      end loop;
      NEW_LINE;
    else
      SET_COL(20);
      PUT_LINE("******* MAIN MEMORY *******");
      SET_COL(20);
```

91

```
PUT_LINE("* CODE              ADDRESS  *");
SET_COL(20);
PUT_LINE("*----------------------*");
SET_COL(20);
PUT("*        ");
PUT(MAIN_MEMORY.TRACE_LINE.CODE);
PUT("          ");
PUT(MAIN_MEMORY.TRACE_LINE.ADDRESS);
PUT_LINE("   * ");
SET_COL(20);
PUT_LINE("**************************");
NEW_LINE;
SET_COL(20);
PUT_LINE("********** READ QUEUE ************");
SET_COL(20);
PUT_LINE("*  CODE        ADDRESS    PRIORITY* ");
SET_COL(20);
PUT_LINE("*---------------------------*");
for I in 1..(Q0.LENGTH_OF(READS)) loop
  Q0.REMOVE(MAQ_HOLD,READS);
  SET_COL(20);
  PUT("*      ");
  PUT(MAQ_HOLD.TRACE_LINE.CODE);
  PUT("          ");
  PUT(MAQ_HOLD.TRACE_LINE.ADDRESS);
  PUT("      ");
  PUT(MAQ_HOLD.TIME_VALUE, WIDTH => 3);
  PUT_LINE("    * ");
  SET_COL(20);
  PUT_LINE("********************************");
  Q0.ADD(MAQ_HOLD,READS);
end loop;
if Q0.IS_EMPTY(READS) then
  SET_COL(20);
  PUT_LINE("*              (empty)              *");
  SET_COL(20);
  PUT_LINE("********************************");
end if;
NEW_LINE;
SET_COL(20);
PUT_LINE("********** WRITE QUEUE ***********");
SET_COL(20);
PUT_LINE("*  CODE        ADDRESS    PRIORITY* ");
SET_COL(20);
PUT_LINE("*---------------------------*");
for I in 1..(Q1.LENGTH_OF(WRITES)) loop
  Q1.REMOVE(MAQ_HOLD,WRITES);
  SET_COL(20);
  PUT("*      ");
  PUT(MAQ_HOLD.TRACE_LINE.CODE);
  PUT("          ");
  PUT(MAQ_HOLD.TRACE_LINE.ADDRESS);
  PUT("      ");
  PUT(MAQ_HOLD.TIME_VALUE, WIDTH => 3);
  PUT_LINE("    * ");
  SET_COL(20);
  PUT_LINE("********************************");
  Q1.ADD(MAQ_HOLD,WRITES);
end loop;
if Q1.IS_EMPTY(WRITES) then
  SET_COL(20);
  PUT_LINE("*              (empty)              *");
  SET_COL(20);
```

```
        PUT_LINE("**********************************");
      end if;
     NEW_LINE(2);
    end if;
  end VIEW_MAQ;
-----------------------------------------------------------------------
  -- This procedure allows the viewing of the contents of the DIQ
  procedure VIEW_DIQ is

    LENGTH                          :INTEGER;
    DIQ_HOLD                        :MAQ_RECORD;
    VIEW                            :CHARACTER := 'n';
    KEY                             :CHARACTER;
  begin

--*************** VIEW Dependent INSTRUCTION QUEUE  *******************
      NEW_LINE;
      LENGTH := DIQ.LENGTH_OF(BLOCKS);
      SET_COL(20);
      PUT_LINE("********************DIQ*********************");
      SET_COL(20);
      PUT_LINE("*  CODE        ADDRESS     TIME   RS1   RS2   * ");
      SET_COL(20);
      PUT_LINE("*---------------------------------------------*");
      for I in 1..LENGTH loop
        DIQ.REMOVE(DIQ_HOLD,BLOCKS);
        SET_COL(20);
        PUT("*      ");
        PUT(DIQ_HOLD.TRACE_LINE.CODE);
        PUT("          ");
        PUT(DIQ_HOLD.TRACE_LINE.ADDRESS);
        PUT("     ");
        PUT(DIQ_HOLD.TIME_VALUE, WIDTH => 3);
        PUT("     ");
        if DIQ_HOLD.TRACE_LINE.SOURCE1_REGISTER /= BLANK then
          PUT(HEX_TO_INTEGER(DIQ_HOLD.TRACE_LINE.SOURCE1_REGISTER),WIDTH => 2);
        else
          PUT("  ");
        end if;
        PUT("     ");
        if DIQ_HOLD.TRACE_LINE.SOURCE2_REGISTER /= BLANK  then
          PUT(HEX_TO_INTEGER(DIQ_HOLD.TRACE_LINE.SOURCE2_REGISTER),WIDTH => 2);
        else
          PUT("  ");
        end if;
        PUT_LINE(" * ");
        DIQ.ADD(DIQ_HOLD,BLOCKS);
        SET_COL(20);
        PUT_LINE("**********************************************");
      end loop;
      NEW_LINE;

  end VIEW_DIQ;


-----------------------------------------------------------------------
  -- This procedure allow the viewing of the PEQ contents.
  procedure VIEW_PEQ is

    LENGTH                          :INTEGER;
    PEQ_HOLD                        :EVENT_RECORD;

  begin
```

```
    SET_COL(20);
    PUT_LINE("*********** PEQ ***********");
    SET_COL(20);
    PUT_LINE("*  EVENT              TIME  *");
    SET_COL(20);
    PUT_LINE("*------------------------------*");
    LENGTH := PEQ.LENGTH_OF(WAITING);
    if not PEQ.IS_EMPTY(WAITING) then
       for I in 1..LENGTH loop
          PEQ.REMOVE(PEQ_HOLD,WAITING);
          SET_COL(20);
          PUT("*    ");
          PUT(PEQ_HOLD.EVENT_ID);
          PUT("            ");
          PUT(PEQ_HOLD.PRIORITY,WIDTH => 5);
          PUT_LINE("    *");
          SET_COL(20);
          PUT_LINE("***************************");
          PEQ.ADD(PEQ_HOLD,WAITING);
       end loop;
    end if;
 end VIEW_PEQ;

---------------------------------------------------------------------
  -- This procedure controls the viewing of all queues and provides
  -- interim results.
  procedure VIEW_RESULTS is

    VIEW                        :CHARACTER := 'n';
    CPI_VALUE                   :FLOAT;

  begin

     NEW_LINE;
     PUT("INSTRUCTION COUNT: ");
     PUT(RECORD_COUNTER, WIDTH => 1);
     NEW_LINE;
     PUT("PROGRAM EXECUTION TIME IN CYCLES: ");
     PUT(EXECUTE_TIME,WIDTH => 1);
     NEW_LINE;
     CPI_VALUE := FLOAT(EXECUTE_TIME)/FLOAT(RECORD_COUNTER);
     PUT("CURRENT CPI VALUE: ");
     PUT(CPI_VALUE, FORE => 3, AFT => 2, EXP => 0);
     NEW_LINE;

  end VIEW_RESULTS;

---------------------------------------------------------------------
  -- This procedure displays the contents of each address trace line (record)
  procedure VIEW_TRACE_LINE(TRACE :in MAQ_RECORD) is

  begin

    if VIEW = 'y' then
      PUT("INSTRUCTION FETCH:  ");
      PUT(TRACE.TRACE_LINE.CODE); PUT("  ");
      PUT(TRACE.TRACE_LINE.ADDRESS);
      NEW_LINE;
    end if;

  end VIEW_TRACE_LINE;

---------------------------------------------------------------------
```

```
-- This procedure puts entries into the MAQ
procedure ENTER_MAQ(TRACE   :in out MAQ_RECORD)   is

begin

   -- PUT_LINE("ENQUEUING MAQ");
   MAQ.ADD(TRACE,REQUESTS);

   -- if the entry is a load instruction, the destination register is
   -- marked as blocked.
   if TRACE.TRACE_LINE.CODE = '0' then
     BLOCKED_REGISTER(HEX_TO_INTEGER(TRACE.TRACE_LINE.TARGET_REGISTER)) := TRUE;
     Q_READS := Q_READS + 1;
   else
     Q_WRITES := Q_WRITES + 1;
   end if;
end ENTER_MAQ;

--------------------------------------------------------------------------
-- This procedure puts entries into the LOAD queue
procedure ENTER_Q0(TRACE   :in out MAQ_RECORD)   is

begin

   -- PUT_LINE("ENQUEUING 'read Q' ");
   Q0.ADD(TRACE,READS);
   BLOCKED_REGISTER(HEX_TO_INTEGER(TRACE.TRACE_LINE.TARGET_REGISTER)) := TRUE;
     Q_READS := Q_READS + 1;
end ENTER_Q0;
--------------------------------------------------------------------------
-- This procedure put elements into the STORE queue
procedure ENTER_Q1(TRACE   :in out MAQ_RECORD)   is

begin

   -- PUT_LINE("ENQUEUING 'write Q' ");
   Q1.ADD(TRACE,WRITES);
   Q_WRITES := Q_WRITES + 1;
end ENTER_Q1;
--------------------------------------------------------------------------
-- This procedure puts entries into the PEQ
procedure ENTER_PEQ(EVENT   :in out EVENT_RECORD)   is

begin
   PEQ.ADD(EVENT,WAITING);
end ENTER_PEQ;

--------------------------------------------------------------------------
-- This procedure puts elements into the DIQ
procedure ENTER_DIQ(INSTRUCTION   :in out MAQ_RECORD)   is

begin
   -- PUT_LINE("ENQUEUING DIQ");
   DIQ.ADD(INSTRUCTION,BLOCKS);
--     BLOCKED_REGISTER(HEX_TO_INTEGER(INSTRUCTION.TRACE_LINE.TARGET_REGISTER))
--        := TRUE;
 end ENTER_DIQ;
--------------------------------------------------------------------------
-- This procedure take instructions from the MAQ
procedure SERVE_MAQ(TRACE   :in out MAQ_RECORD)   is

   TARGET                         :MAQ_RECORD;
   MAQ_HOLD                       :MAQ_RECORD;
```

95

```ada
  Q_LENGTH                         :INTEGER;
  HI_POSITION                      :INTEGER := 0;
  MAQ_POSITION                     :INTEGER := 0;
  QUEUE_HEAD                       :INTEGER := 999999;
  FOUND                            :BOOLEAN := FALSE;
begin
  MAQ.REMOVE(MAQ_HOLD,REQUESTS);   -- request leaving main memory
  Q_LENGTH := MAQ.LENGTH_OF(REQUESTS); -- getting the length of the queue
  ------------------------------------------------------------------
  -- If the removed instruction is a load instruction, the destination
  -- is unblocked.
  if MAQ_HOLD.TRACE_LINE.CODE = '0' then
    BLOCKED_REGISTER(HEX_TO_INTEGER(MAQ_HOLD.TRACE_LINE.TARGET_REGISTER))
      := FALSE;
  end if;
  ------------------------------------------------------------------

  -- this statement isfor determining the next item by priority to
  -- remove from the queue.  The target item is identified by its
  -- position in the queue.  Each item is removed, compared, and
  -- re-entered into the queue.
  if not MAQ.IS_EMPTY(REQUESTS) then
    for I in 1..Q_LENGTH loop
      HI_POSITION := HI_POSITION + 1;
      MAQ.REMOVE(MAQ_HOLD,REQUESTS);
      if MAQ_HOLD.TIME_VALUE < QUEUE_HEAD then
        QUEUE_HEAD := MAQ_HOLD.TIME_VALUE;
        MAQ_POSITION := HI_POSITION;
        TARGET := MAQ_HOLD;
      end if;
      MAQ.ADD(MAQ_HOLD,REQUESTS);
    end loop;

    HOLD_VALUE := TARGET.TIME_VALUE;
    if VIEW = 'y' then
      NEW_LINE;
      PUT("ENTERING MAIN MEMORY:   ");
      PUT(TARGET.TRACE_LINE.CODE); PUT("   ");
      PUT(TARGET.TRACE_LINE.ADDRESS); PUT("   ");
      PUT(TARGET.TIME_VALUE,WIDTH => 1);
    end if;

    -- The value of MAQ_POSITION is the position in the queue of the
    -- item to be removed.
    HI_POSITION := 0;
    MAQ.ADD(TARGET,REQUESTS);
    for I in 1..Q_LENGTH  loop
      MAQ.REMOVE(MAQ_HOLD,REQUESTS);
      HI_POSITION := HI_POSITION + 1;
      if MAQ_POSITION /= HI_POSITION then
        MAQ.ADD(MAQ_HOLD,REQUESTS);
      end if;
    end loop;

    --  Numerical data collection statements
    if TARGET.TRACE_LINE.CODE = '0' then
      Q_READS := Q_READS - 1;
      TOTAL_CYCLES := TOTAL_CYCLES + LATENCY;
    else
      Q_WRITES := Q_WRITES - 1;
      TOTAL_CYCLES := TOTAL_CYCLES + LATENCY;
    end if;
  end if;
```

```
  end SERVE_MAQ;
------------------------------------------------------------------------
-- This procedure removes items from the DIQ.  This is a FIFO
-- queue, thus the generic REMOVE function is used.
procedure SERVE_BIQ(TRACE  :in out MAQ_RECORD)  is

    TARGET                          :MAQ_RECORD;
    DIQ_HOLD                        :MAQ_RECORD;
    Q_LENGTH                        :INTEGER;

begin
  if not DIQ.IS_EMPTY(BLOCKS) then
    DIQ.REMOVE(TARGET,BLOCKS);
    if TARGET.TRACE_LINE.SOURCE1_REGISTER = BLANK  then
      if BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.SOURCE2_REGISTER))
         = FALSE then
        DIQ_FETCHED := TRUE;
        -- The removed instruction becomes the active instruction executed
        TRACE := TARGET;
        -- Unblocks the destination register
        BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.TARGET_REGISTER))
          := FALSE;
        if VIEW = 'y' then
          NEW_LINE;
          PUT_LINE("FETCHING FROM DIQ");
          PUT(TARGET.TRACE_LINE.CODE); PUT("   ");
          PUT(TARGET.TRACE_LINE.ADDRESS); PUT("   ");
          PUT(TARGET.TIME_VALUE,WIDTH => 1);
        end if;
      else
        -- Put removed item back into DIQ
        DIQ.ADD(TARGET,BLOCKS);
        DIQ_FETCHED := FALSE;
      end if;
    end if;
    if TARGET.TRACE_LINE.SOURCE2_REGISTER = BLANK then
      if BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.SOURCE1_REGISTER))
         = FALSE then
        DIQ_FETCHED := TRUE;
        -- The removed instruction becomes the active instruction executed
        TRACE := TARGET;
        -- Unblocks the destination register
        BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.TARGET_REGISTER))
          := FALSE;
        if VIEW = 'y' then
          NEW_LINE;
          PUT_LINE("FETCHING FROM DIQ");
          PUT(TARGET.TRACE_LINE.CODE); PUT("   ");
          PUT(TARGET.TRACE_LINE.ADDRESS); PUT("   ");
          PUT(TARGET.TIME_VALUE,WIDTH => 1);
        end if;
      else
        -- Put removed item back into DIQ
        DIQ.ADD(TARGET,BLOCKS);
        DIQ_FETCHED := FALSE;
      end if;
    end if;
    if TARGET.TRACE_LINE.SOURCE1_REGISTER /= BLANK  and
       TARGET.TRACE_LINE.SOURCE2_REGISTER /= BLANK  then
      if BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.SOURCE1_REGISTER))
         = FALSE
         and BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.SOURCE2_REGISTER))
         = FALSE  then
```

```
        DIQ_FETCHED := TRUE;
        -- The removed instruction becomes the active instruction executed
        TRACE := TARGET;
        -- Unblocks the destination register
        BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.TARGET_REGISTER))
          := FALSE;
        if VIEW = 'y' then
          NEW_LINE;
          PUT_LINE("FETCHING FROM DIQ");
          PUT(TARGET.TRACE_LINE.CODE); PUT("   ");
          PUT(TARGET.TRACE_LINE.ADDRESS); PUT("   ");
          PUT(TARGET.TIME_VALUE,WIDTH => 1);
        end if;
      else
        -- Put removed item back into DIQ
        DIQ.ADD(TARGET,BLOCKS);
        DIQ_FETCHED := FALSE;
      end if;
    end if;
  end if;
end SERVE_DIQ;
------------------------------------------------------------------------

-- This procedure removes entries from the LOAD MAQ.  The items are
-- removed FIFO.
procedure SERVE_Q0(TRACE   :in out MAQ_RECORD)   is

  TARGET                        :MAQ_RECORD;
  Q0_HOLD                       :MAQ_RECORD;
  Q_LENGTH                      : INTEGER;

begin
  if not Q0.IS_EMPTY(READS) then
    Q0.REMOVE(TARGET,READS);
    if VIEW = 'y' then
      NEW_LINE;
      PUT_LINE("FETCHING FROM READ Q");
      PUT(TARGET.TRACE_LINE.CODE); PUT("   ");
      PUT(TARGET.TRACE_LINE.ADDRESS); PUT("   ");
      PUT(TARGET.TIME_VALUE,WIDTH => 1);
    end if;
    TRACE := TARGET;
    BLOCKED_REGISTER(HEX_TO_INTEGER(TARGET.TRACE_LINE.TARGET_REGISTER)) := FALSE;
  end if;

end SERVE_Q0;
------------------------------------------------------------------------

-- This procedure removes entries from the STORE MAQ (FIFO).
procedure SERVE_Q1(TRACE   :in out MAQ_RECORD)   is

  TARGET                        :MAQ_RECORD;
  Q1_HOLD                       :MAQ_RECORD;
  Q_LENGTH                      : INTEGER;

begin
  if not Q1.IS_EMPTY(WRITES) then
    Q1.REMOVE(TARGET,WRITES);
    if VIEW = 'y' then
      NEW_LINE;
      PUT_LINE("FETCHING FROM WRITE Q");
      PUT(TARGET.TRACE_LINE.CODE); PUT("   ");
      PUT(TARGET.TRACE_LINE.ADDRESS); PUT("   ");
```

```
              PUT(TARGET.TIME_VALUE,WIDTH => 1);
           end if;
           TRACE := TARGET;
        end if;

      end SERVE_Q1;
-------------------------------------------------------------------
      -- This procedure takes items from the PEQ.  This is a priority
      -- queue, so the item to be remove is identified by its priority
      -- value.
      procedure SERVE_PEQ(TARGET         in out EVENT_RECORD)   is

          --   TARGET                   :EVENT_RECORD;
          EVENT_HOLD               :EVENT_RECORD;
          HI_POSITION              :INTEGER := 0;
          EVENT_POSITION           :INTEGER := 0;
          QUEUE_HEAD               :INTEGER := 999999;
          FOUND                    :BOOLEAN := FALSE;
      begin

        PEQ_VIEW.CLEAR(PEQ_COUNT);
        while not PEQ.IS_EMPTY(WAITING)  loop
          HI_POSITION := HI_POSITION + 1;
          PEQ.REMOVE(EVENT_HOLD,WAITING);
          if EVENT_HOLD.PRIORITY < QUEUE_HEAD then
             QUEUE_HEAD := EVENT_HOLD.PRIORITY;
             EVENT_POSITION := HI_POSITION;
             TARGET := EVENT_HOLD;
          end if;
          PEQ_VIEW.ADD(EVENT_HOLD,PEQ_COUNT);
        end loop;

        HOLD_VALUE := TARGET.PRIORITY;
        if VIEW = 'y' then
           NEW_LINE;
           PUT("ITEM SERVICED : ");
           PUT(TARGET.EVENT_ID); PUT("   ");
           PUT(TARGET.PRIORITY,WIDTH => 1);
           NEW_LINE;
        end if;
        PEQ.CLEAR(WAITING);

        HI_POSITION := 0;
        while not PEQ_VIEW.IS_EMPTY(PEQ_COUNT)  loop
          PEQ_VIEW.REMOVE(EVENT_HOLD,PEQ_COUNT);
          HI_POSITION := HI_POSITION + 1;
          if EVENT_POSITION /= HI_POSITION then
             PEQ.ADD(EVENT_HOLD,WAITING);
          end if;
        end loop;

      end SERVE_PEQ;
-------------------------------------------------------------------
      -- This procedure displays interim results of the simulation.
      procedure INTERVAL_CHECK(INTERVAL     :in out INTEGER;
                               MAQ_LENGTH :in out INTEGER) is
         begin
           if (RECORD_COUNTER mod INTERVAL) = 0    then
              NEW_LINE;
              MAQ_LENGTH := MAQ.LENGTH_OF(REQUESTS);
              PUT("Number of records processed: ");
              PUT(RECORD_COUNTER, WIDTH => 1);
              NEW_LINE;
```

99

```
        PUT("NUMBER OF RECORDS IN MAQ:   ");
        if MAQ_LENGTH /= 0 then
          PUT((MAQ_LENGTH-1),WIDTH => 1);
        else
          PUT(MAQ_LENGTH, WIDTH => 1);
        end if;
        NEW_LINE;
        VIEW_QUEUES;
        PUT("Do you want to continue with simulation?  (y/n): ");
        GET(RESPONSE);
        NEW_LINE;
      end if;
  end INTERVAL_CHECK;
-------------------------------------------------------------------------
  -- This procedure checks the status of a particular register.  TRUE
  -- means the register is available for access, FALSE means the register
  -- is blocked and awaiting new data.
  procedure CHECK_BLOCKED_REGISTER(REGISTER   :in out STRING) is

    REG_NO                 :INTEGER;

    begin

      if BLOCKED_REGISTER(HEX_TO_INTEGER(REGISTER)) then
        --   ENTER_DIQ(TRACE);
        BLOCKED := TRUE;
      end if;

      -- The next statements determine if the instruction immediately following
      -- a LOAD instruction requires the data from the destination register.
      -- The LOAD_SWITCH is set when a load instruction occurs. If the switch
      -- is set (TRUE) then the following instruction's source registers are
      -- are checked for dependency against the previous load. If there is
      -- a load dependency, the execution time is incremented by the amount
      -- of the dependency penalty (simulates a stall)
      if LOAD_SWITCH and LOAD_REG = HEX_TO_INTEGER(REGISTER)   then
        EXECUTE_TIME := EXECUTE_TIME + LOAD_DEP;
        if VIEW = 'y' then
          PUT("Load dependency stall..");
          PUT(LOAD_DEP,WIDTH => 1);
          PUT_LINE(" cycles");
        end if;
      end if;
  end CHECK_BLOCKED_REGISTER;


-------------------------------------------------------------------------
  -- This procedure handles a MAQ full situation.
  procedure CHECK_MAQ_FULL   is

    CHOICE                  :INTEGER;
    N                       :INTEGER;

    begin

      SERVE_MAQ(TRACE);
      if VIEW = 'y' then
        VIEW_QUEUES;
        NEW_LINE;
      end if;
  end CHECK_MAQ_FULL;
-------------------------------------------------------------------------
  -- This procedure handles situations when the processor stalls
  -- because of data dependency; required data is in the MAQ.
```

100

```
-- In essence the procedure performs "lm" (leave memory) events
-- until the required data is available.
procedure LM2 is

   TRACE2                        :MAQ_RECORD;

begin

   -- if the MAQ has separate queues for loads and stores, the
   -- procedure ensures the correct queue is served.
   if SEPERATE_Q = 'y'  then

      -- If LOADS have a higher priority (lower value) then serve the
      -- load queue first.  If the load queue is empty, then proceed to
      -- the store queue.  If both are empty, then serve main memory queue,
      -- which means that the needed item is currently retrieving data
      -- from main memory.  Registers released from main memory are
      -- unblocked, as usual.
      if READ_PRI < WRITE_PRI then
        if not Q0.IS_EMPTY(READS) then
          SERVE_Q0(TRACE2);
          MAIN_MEMORY := TRACE2;
        elsif not Q1.IS_EMPTY(WRITES) then
          SERVE_Q1(TRACE2);
          MAIN_MEMORY := TRACE2;
        else
          if MAIN_MEMORY.TRACE_LINE.CODE = '0' then
            BLOCKED_REGISTER(HEX_TO_INTEGER(MAIN_MEMORY.
              TRACE_LINE.TARGET_REGISTER)) := FALSE;
          end if;
          MAIN_MEMORY.TRACE_LINE.CODE := ' ';
          MAIN_MEMORY.TRACE_LINE.ADDRESS := " (EMPTY)";
        end if;

      else
        -- Else STORES have priority.  Same logic as above applies.
        if not Q1.IS_EMPTY(WRITES) then
          SERVE_Q1(TRACE2);
          MAIN_MEMORY := TRACE2;
        elsif not Q0.IS_EMPTY(READS) then
          SERVE_Q0(TRACE2);
          MAIN_MEMORY := TRACE2;
        else
          if MAIN_MEMORY.TRACE_LINE.CODE = '0' then
            BLOCKED_REGISTER(HEX_TO_INTEGER(MAIN_MEMORY.
              TRACE_LINE.TARGET_REGISTER)) := FALSE;
          end if;
          MAIN_MEMORY.TRACE_LINE.CODE := ' ';
          MAIN_MEMORY.TRACE_LINE.ADDRESS := " (EMPTY)";
        end if;
      end if;

   else
      -- A single MAQ is used
      if not MAQ.IS_EMPTY(REQUESTS) then
        SERVE_MAQ(TRACE2);
      end if;
   end if;
end LM2;
----------------------------------------------------------------
procedure ISSUE_INSTRUCTION(HIT_RATE    :in out FLOAT;
                            TRACE       :in out MAQ_RECORD;
                            VIEW        :in out CHARACTER)  is
```

```
    CACHE_VALUE                     :FLOAT := 0.0;
    HOLD,
    EVENT                           :EVENT_RECORD;
    MISS_REGISTER                   :INTEGER := 0;
    PAUSE                           :CHARACTER;
    TRACE2                          :MAQ_RECORD;
    CHOICE                          :INTEGER;
    R1,R2                           :STRING(1..2);
    LM                              :STRING(1..2) := "lm";
    II                              :STRING(1..2) := "ii";
    N                               :INTEGER;
    RANDOM_DIQ                      :FLOAT;
begin

    if DIQ_USED = 'y' then
      DIQ_FETCHED := FALSE;
      SERVE_DIQ(TRACE);
      if DIQ_FETCHED = FALSE then
        -- An instruction is issued from the address trace file
        if not END_OF_FILE(INPUT_FILE) then
          DO_LINE_PARSING(INPUT_FILE,TRACE_REC);
          VIEW_TRACE_LINE(TRACE);
          if TRACE.TRACE_LINE.CODE = '2' or TRACE.TRACE_LINE.CODE = '3' then
            RECORD_COUNTER := RECORD_COUNTER + 1;
          end if;
        end if;
      end if;
    else
        -- An instruction is issued from the address trace file
        if not END_OF_FILE(INPUT_FILE) then
          DO_LINE_PARSING(INPUT_FILE,TRACE_REC);
          VIEW_TRACE_LINE(TRACE);
          if TRACE.TRACE_LINE.CODE = '2' or TRACE.TRACE_LINE.CODE = '3' then
            RECORD_COUNTER := RECORD_COUNTER + 1;
          end if;
        end if;
    end if;

    -- checking to see if source registers of the fetched instruction
    -- are blocked or waiting for memory access
    BLOCKED := FALSE;
    if TRACE.TRACE_LINE.SOURCE1_REGISTER /= BLANK then
      CHECK_BLOCKED_REGISTER(TRACE.TRACE_LINE.SOURCE1_REGISTER);
    end if;
    if TRACE.TRACE_LINE.SOURCE2_REGISTER /= BLANK then
      CHECK_BLOCKED_REGISTER(TRACE.TRACE_LINE.SOURCE2_REGISTER);
    end if;
    R1 := TRACE.TRACE_LINE.SOURCE1_REGISTER;
    R2 := TRACE.TRACE_LINE.SOURCE2_REGISTER;

    -- If the instruction is dependent on blocked data the system
    -- stalls until the data is available.
    if BLOCKED then

      -- R1 and R2 are the source registers.  If source registers are
      -- used, they must be checked.
      ---------------------------------------------------------------
      if DIQ_USED = 'y'  then
        ENTER_DIQ(TRACE);
      else
        -- if R1 and R2 are used in the instruction
        if (R1 /= BLANK) and (R2 /= BLANK)  then
          -- serve the MAQ until the data is available
```

102

```
        while BLOCKED_REGISTER(HEX_TO_INTEGER(R1)) or
             BLOCKED_REGISTER(HEX_TO_INTEGER(R2))  loop

            -- serve the next lm event to see if it has the dependent
            -- data.  Effects a stall equal to the lm priority value
            -- minus the current execution time.  The priority value
            -- of a lm event means that is the time the data will be
            -- available for use.
            SERVE_PEQ(HOLD);
            if VIEW = 'y' then
              PUT("Process stalled for blocked memory request..");
              PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
              PUT_LINE(" cycles elapsed.");
            end if;
            if HOLD.EVENT_ID = "lm" then
              LM2;
              EXECUTE_TIME := HOLD.PRIORITY;
            end if;
          end loop;
      end if;    -- R1 not blank and R2 not blank
-----------------------------------------------------------------
      -- if R1 is used and R2 is not used
      if (R1 /= BLANK) and (R2 = BLANK)  then
        while BLOCKED_REGISTER(HEX_TO_INTEGER(R1)) loop
          SERVE_PEQ(HOLD);
          if VIEW = 'y' then
            PUT("Process stalled for blocked memory request..");
            PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
            PUT_LINE(" cycles elapsed.");
          end if;
          if HOLD.EVENT_ID = "lm" then
            LM2;
            EXECUTE_TIME := HOLD.PRIORITY;
          end if;
        end loop;
      end if;    -- R1 not blank and R2 is blank
-----------------------------------------------------------------
      -- if R1 is not used and R2 is used
      if (R1 = BLANK) and (R2 /= BLANK)  then
        while BLOCKED_REGISTER(HEX_TO_INTEGER(R2)) loop
          SERVE_PEQ(HOLD);
          if VIEW = 'y' then
            PUT("Process stalled for blocked memory request..");
            PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
            PUT_LINE(" cycles elapsed.");
          end if;
          if HOLD.EVENT_ID = "lm" then
            LM2;
            EXECUTE_TIME := HOLD.PRIORITY;
          end if;
        end loop;
      end if; -- R1 blank and R2 not blank
    end if;  -- If/then else DIQ used
  end if;    -- if blocked
-----------------------------------------------------------------

    -- The following statements processes the fetched instruction as
    -- as a store command (code = 1).  An lm event is entered into the
    -- PEQ with a priority value equal to the current execution time
    -- plus the number of cycles required for a write to main memory.
    -- Since the instruction is a main memory request, it is entered
    -- into the MAQ.
    if (TRACE.TRACE_LINE.CODE = '1') then
```

```
TOTAL_PENALTY := TOTAL_PENALTY + LATENCY;
EVENT.EVENT_ID := LM;
EVENT.PRIORITY := EXECUTE_TIME + LATENCY;
ENTER_PEQ(EVENT);
TRACE.TIME_VALUE := WRITE_PRI;
-----------------------------------------------------------
-- if using seperate queues for loads and stores, put into
-- store queue (Q1).  If Q1 is empty then put into main memory.
if SEPERATE_Q = 'y' then
   if Q1_SIZE = Q1.LENGTH_OF(WRITES) then
      SERVE_PEQ(HOLD);
      if VIEW = 'y'  then
         PUT("Process stalled for blocked memory request...");
         PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
         PUT_LINE(" cycles elapsed. ");
      end if;
      SERVE_Q1(TRACE2);
      MAIN_MEMORY := TRACE2;
      EXECUTE_TIME := HOLD.PRIORITY;
   end if;
   if MAIN_MEMORY.TRACE_LINE.CODE = ' ' then
      MAIN_MEMORY := TRACE;
   else
      ENTER_Q1(TRACE);
   end if;
else
   if MAQ_SIZE = MAQ.LENGTH_OF(REQUESTS) then
      SERVE_PEQ(HOLD);
      if VIEW = 'y'  then
         PUT("Process stalled...MAQ full...serving request...");
         PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
         PUT_LINE(" cycles elapsed. ");
      end if;
      EXECUTE_TIME := HOLD.PRIORITY;
      SERVE_MAQ(TRACE2);
   end if;
   ENTER_MAQ(TRACE);
end if;
-----------------------------------------------------------
-- Since this is Not a load instruction, the load dependency
-- switch is turned off.
LOAD_SWITCH := FALSE;

-- for viewing program execution
if VIEW = 'y'  then
   PUT_LINE("WRITE ADDED");
end if;


-----------------------------------------------------------
-- Processes the fetched instruction as a load instruction.  The
-- load is either a cache hit or miss.  This is determined by the
-- hit-rate input by the user.  A random number generator produces
-- a value betwee 0.0 and 100.0.  If the number generated is greater
-- than the hit-rate, then the load is a miss.
-- Sorry, but that's the best I can do without a cache simulator.
elsif (TRACE.TRACE_LINE.CODE = '0') then
   -- Since this is a load instruction, the next instruction fetched
   -- must be checked for dependency on this load statement, therefore,
   -- the load destination register is identified and the load switch
   -- is set to alert the processor to check the next instruction.
   LOAD_REG := HEX_TO_INTEGER(TRACE.TRACE_LINE.TARGET_REGISTER);
   LOAD_SWITCH := TRUE;
   CACHE_VALUE := NUMBER_IN_RANGE(0.0,100.0);
```

```
-- the following statements processes the load statement as a cache
-- miss.  An lm event is enqueued to the PEQ with a priority value
-- equal to the current execution time plus the time it takes to
-- to load from main memory. An entry must also be placed into MAQ.
if CACHE_VALUE > HIT_RATE then
   TOTAL_PENALTY := TOTAL_PENALTY + LATENCY;
   EVENT.EVENT_ID := LM;
   EVENT.PRIORITY := EXECUTE_TIME + LATENCY;
   ENTER_PEQ(EVENT);
   TRACE.TIME_VALUE := READ_PRI;
   -- Put into appropriate MAQ
   if SEPERATE_Q = 'y'   then
      if Q0_SIZE = Q0.LENGTH_OF(READS) then
         SERVE_PEQ(HOLD);
         if VIEW = 'y'   then
            PUT("Process stalled...Write Queue full...");
            PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
            PUT_LINE(" cycles elapsed. ");
         end if;
         SERVE_Q0(TRACE2);
         MAIN_MEMORY := TRACE2;
         EXECUTE_TIME := HOLD.PRIORITY;
      end if;
      if MAIN_MEMORY.TRACE_LINE.CODE = ' '   then
         MAIN_MEMORY := TRACE;
      else
         ENTER_Q0(TRACE);
      end if;
   else
      if MAQ_SIZE = MAQ.LENGTH_OF(REQUESTS) then
         SERVE_PEQ(HOLD);
         if VIEW = 'y'   then
            PUT("Process stalled...MAQ full...");
            PUT((HOLD.PRIORITY-EXECUTE_TIME),WIDTH => 1);
            PUT_LINE(" cycles elapsed. ");
         end if;
         SERVE_MAQ(TRACE2);
         EXECUTE_TIME := HOLD.PRIORITY;
      end if;
      ENTER_MAQ(TRACE);
   end if;
   if VIEW = 'y'   then
      PUT_LINE("READ MISS ADDED");
   end if;


-------------------------------------------------------------
-- If the load is a cache miss then the process continues as a
-- non-memory access.  An event ii is enqueued at the end of
-- the ISSUE_INSTRUCTION procedure.
else     -- READ HIT
  TOTAL_CYCLES := TOTAL_CYCLES + 1;
end if;

-------------------------------------------------------------
-- The following statements simulates a processor stall for a
-- branch instruction (code = 3).  The execution time is incremented
-- by the amount of branch penalty previously specified.
elsif (TRACE.TRACE_LINE.CODE = '3') then     -- BRANCH instruction
   TOTAL_CYCLES := TOTAL_CYCLES + BF_CPI;
   EXECUTE_TIME := EXECUTE_TIME + BF_CPI - 1;
   LOAD_SWITCH := FALSE;


-------------------------------------------------------------
```

```
   -- The code = 2 presents a one-cycle-execution statement. Therefore,
   -- the next instruction can be executed at execution time + 1.
   -- This is handled at the end of the II procedure.
   else   -- (TRACE.CODE = '2')
     TOTAL_CYCLES := TOTAL_CYCLES + 1;
     LOAD_SWITCH := FALSE;
   end if;


   ------------------------------------------------------------------
   -- After processing every ii event, another ii event is put into the
   -- PEQ with a priority value of the current execution time + 1, which
   -- means the next instruction can be fetched on the next clock cycle.
   if not END_OF_FILE(INPUT_FILE) then
     EXECUTE_TIME := EXECUTE_TIME + 1;
     EVENT.EVENT_ID := II;
     EVENT.PRIORITY := EXECUTE_TIME;
     ENTER_PEQ(EVENT);
    end if;
   ----------------------------------------------------------------
 end ISSUE_INSTRUCTION;
------------------------------------------------------------------------


   -- This procedure processes the lm (leave memory) event.  It basically
   -- takes the appropriate item from the appropriate MAQ.  This simulates
   -- that the request has completed its main memory access and is available
   -- for use. When the item leaves main memory, the next item enters.
   procedure LEAVE_MEMORY is

   begin

     -- If seperate queues are used, then either the load or the store request
     -- has priority to enter memory next.  If the loads have priority, then
     -- the load queue (Q0) is served. If Q0 is empty, then the write queue
     -- is served.  If both queues are empty, then there are no main memory
     -- requests, and no requests are currently in main memory.

     -- if loads have priority over stores.

     if SEPERATE_Q = 'y'   then
      if READ_PRI < WRITE_PRI then
        if not Q0.IS_EMPTY(READS) then
          SERVE_Q0(TRACE);
          MAIN_MEMORY := TRACE;
        elsif not Q1.IS_EMPTY(WRITES) then
          SERVE_Q1(TRACE);
          MAIN_MEMORY := TRACE;
        else
          if MAIN_MEMORY.TRACE_LINE.CODE = '0' then
            BLOCKED_REGISTER(HEX_TO_INTEGER(MAIN_MEMORY.
             TRACE_LINE.TARGET_REGISTER)) := FALSE;
          end if;
          MAIN_MEMORY.TRACE_LINE.CODE := ' ';
          MAIN_MEMORY.TRACE_LINE.ADDRESS := " (EMPTY)";
        end if;

     -- if stores have priority over loads
     else
        if not Q1.IS_EMPTY(WRITES) then
          SERVE_Q1(TRACE);
          MAIN_MEMORY := TRACE;
        elsif not Q0.IS_EMPTY(READS) then
          SERVE_Q0(TRACE);
          MAIN_MEMORY := TRACE;
```

106

```
        else
          if MAIN_MEMORY.TRACE_LINE.CODE = '0' then
            BLOCKED_REGISTER(HEX_TO_INTEGER(MAIN_MEMORY.
             TRACE_LINE.TARGET_REGISTER)) := FALSE;
          end if;
          MAIN_MEMORY.TRACE_LINE.CODE := ' ';
          MAIN_MEMORY.TRACE_LINE.ADDRESS := " (EMPTY)";
        end if;
      end if;
   _____
   -- if only one queue is used for loads and stores, then the
   -- next item in the queue is served (enters main memory).
   else
     if not MAQ.IS_EMPTY(REQUESTS) then
       SERVE_MAQ(TRACE);
     end if;
   end if;
 end LEAVE_MEMORY;
_____
 -- This procedure serves items from the PEQ and processes them.
 procedure PROCESS_REQUEST is

   PEQ_HOLD            :EVENT_RECORD;
   II                  :STRING(1..2) := "ii";
   LM                  :STRING(1..2) := "lm";
 begin

   SERVE_PEQ(PEQ_HOLD);
   if PEQ_HOLD.EVENT_ID = II   then
     ISSUE_INSTRUCTION(HIT_RATE,TRACE,VIEW);
   elsif PEQ_HOLD.EVENT_ID = LM   then
     LEAVE_MEMORY;
     if END_OF_FILE(INPUT_FILE) then
       EXECUTE_TIME := PEQ_HOLD.PRIORITY;
     end if;
   else
     PUT_LINE("UNKNOWN EVENT...IGNORING");
   end if;

 end PROCESS_REQUEST;

 --****************************************************************
 --*                 main prograM BEGINS HERE !!!!!!!!!!!!        *
 --****************************************************************
begin--main procedure

 CLEARSCREEN;
 while ANOTHER = 'y'  loop
  GET_INPUT_FILE(INPUT_FILE);
  if SAME_DATA = 'n' then
    GET_INITIAL_DATA;
  end if;
  while not PEQ.IS_EMPTY(WAITING) loop
    PROCESS_REQUEST;
    INTERVAL_CHECK(INTERVAL,MAQ_LENGTH);
    if (RESPONSE = 'n') then
      NEW_LINE;
      PUT_LINE("PROGRAM TERMINATED !!! ");
    end if;
    exit when (RESPONSE = 'n');
    if VIEW = 'y'   then
      VIEW_MAQ;
      NEW_LINE;
```

```
      VIEW_PEQ;
      NEW_LINE;
      VIEW_DIQ;
      NEW_LINE;
   end if;
 end loop;
 NEW_LINE;
 VIEW_QUEUES;
 CLOSE(INPUT_FILE);
 -----------------Clear queues for next calculations-----------------
 PEQ.CLEAR(WAITING);
 Q1.CLEAR(WRITES);
 Q0.CLEAR(READS);
 MAQ.CLEAR(REQUESTS);
 EXECUTE_TIME := 0;
 RECORD_COUNTER := 0;
 RESPONSE := 'y';
 --------------------------------------------------------------------
 NEW_LINE(2);
 PUT("Do you want to do another simulation? (y/n): ");
 GET(ANOTHER);
 NEW_LINE;
 PUT("Keep same parameters? (y/n): ");
 GET(SAME_DATA);
 NEW_LINE;
 SKIP_LINE;
 end loop;

end SPLIT;
```

APPENDIX E.   SPA RESULTS OF MATRIX MULTIPLICATION TRACE

Spanner - Sparc performance analyzer
----

cpu:	cy7c601
cache:	ss2
register windows:	8
overflow cost:	170 cycles
underflow cost:	110 cycles

OVERALL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
instructions	92.4	100.0	92.4	−	584500	524964
annulled delay slots	0.0	0.0	0.0	−	119	119
load-use stalls	1.7	2.0	1.7	−	10494	10494
trap cycles	0.0	0.0	0.0	−	168	42
window handlers	0.2	0.0	0.2	−	1570	11
cache cycles	5.6	0.6	5.6	−	35430	3188
total	100.0	−	100.0	−	632281	−

INSTRUCTIONS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
memory access	14.3	7.5	15.5	7.5	90331	39380
alu	64.7	77.9	70.0	77.9	409110	409110
floating point	0.0	0.0	0.0	0.0	0	0
control transfer	10.5	11.0	11.4	11.0	66591	58006
other instructions	2.9	3.5	3.2	3.5	18468	18468
total	92.4	100.0	100.0	100.0	584500	524964

MEMORY ACCESS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
load	8.8	5.3	61.7	70.8	55740	27864
store	5.5	2.2	38.3	29.2	34587	11515
atomic	0.0	0.0	0.0	0.0	4	1
total	14.3	7.5	100.0	100.0	90331	39380

LOAD	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
ldb	0.6	0.4	6.8	6.8	3814	1907
ldh	0.0	0.0	0.2	0.2	124	62
ld	8.2	4.9	92.9	92.9	51764	25882
ldd	0.0	0.0	0.1	0.0	36	12
ldf	0.0	0.0	0.0	0.0	2	1
lddf	0.0	0.0	0.0	0.0	0	0
total	8.8	5.3	100.0	100.0	55740	27864

STORE	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count

	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
stb	0.2	0.1	3.6	3.6	1245	415
sth	0.0	0.0	0.1	0.1	39	13
st	5.2	2.1	95.8	95.9	33135	11045
std	0.0	0.0	0.5	0.4	168	42
stf	0.0	0.0	0.0	0.0	0	0
stdf	0.0	0.0	0.0	0.0	0	0
total	5.5	2.2	100.0	100.0	34587	11515

ALU	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
arithmetic	18.8	22.6	29.0	29.0	118607	118607
logical	14.6	17.6	22.5	22.5	92196	92196
shift	9.3	11.2	14.4	14.4	58895	58895
multiply	19.1	23.0	29.5	29.5	120873	120873
sethi	2.9	3.5	4.5	4.5	18539	18539
total	64.7	77.9	100.0	100.0	409110	409110

ARITHMETIC	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
add	6.9	8.3	36.7	36.7	43513	43513
addcc	8.1	9.8	43.4	43.4	51497	51497
addx	0.0	0.0	0.0	0.0	0	0
addxcc	0.0	0.0	0.0	0.0	0	0
sub	0.0	0.0	0.2	0.2	243	243
subcc	1.7	2.0	9.0	9.0	10647	10647
subx	0.0	0.0	0.0	0.0	0	0
subxcc	0.0	0.0	0.0	0.0	0	0
taddcc	0.0	0.0	0.0	0.0	0	0
taddcctv	0.0	0.0	0.0	0.0	0	0
tsubcc	0.0	0.0	0.0	0.0	0	0
tsubcctv	0.0	0.0	0.0	0.0	0	0
cmp (subcc)	1.7	2.1	9.3	9.3	10987	10987
tst (subcc)	0.3	0.3	1.5	1.5	1720	1720
total	18.8	22.6	100.0	100.0	118607	118607

LOGICAL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
and	0.0	0.0	0.2	0.2	203	203
andcc	2.6	3.1	17.6	17.6	16263	16263
andn	0.0	0.0	0.1	0.1	55	55
andncc	1.3	1.5	8.7	8.7	8060	8060
or	4.3	5.2	29.7	29.7	27338	27338
orcc	1.3	1.6	9.0	9.0	8261	8261
orn	0.0	0.0	0.0	0.0	0	0
orncc	0.0	0.0	0.0	0.0	0	0
xor	0.0	0.0	0.0	0.0	4	4
xorcc	0.0	0.0	0.0	0.0	0	0
xorn	0.0	0.0	0.0	0.0	0	0
xorncc	0.0	0.0	0.0	0.0	0	0
mov (or)	5.1	6.1	34.6	34.6	31936	31936
tst (orcc)	0.0	0.0	0.1	0.1	76	76
total	14.6	17.6	100.0	100.0	92196	92196

SHIFT	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
left	6.7	8.0	71.5	71.5	42137	42137
right logical	1.4	1.7	14.7	14.7	8679	8679
right arithmetic	1.3	1.5	13.7	13.7	8079	8079
total	9.3	11.2	100.0	100.0	58895	58895

MULTIPLY	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
single step	16.6	20.0	86.7	86.7	104755	104755
read y	1.3	1.5	6.7	6.7	8059	8059
write y	1.3	1.5	6.7	6.7	8059	8059
total	19.1	23.0	100.0	100.0	120873	120873

SETHI	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
sethi	2.9	3.5	98.9	98.9	18336	18336
nop	0.0	0.0	1.1	1.1	203	203
total	2.9	3.5	100.0	100.0	18539	18539

CONTROL TRANSFER	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
conditional branch	4.9	5.9	46.6	53.6	31064	31064
unconditional branch	1.5	1.9	14.7	16.9	9795	9795
jmpl	2.7	1.6	25.8	14.8	17170	8585
call	1.4	1.6	12.9	14.8	8562	8562
total	10.5	11.0	100.0	100.0	66591	58006

COND. BR.: MB86901	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
backward taken	0.3	0.3	3.6	5.8	1791	1791
backward untaken	0.0	0.0	0.6	0.5	306	153
forward taken	1.7	2.1	21.7	34.7	10776	10776
forward untaken	5.8	3.5	74.0	59.1	36688	18344
total	7.8	5.9	100.0	100.0	49561	31064

COND. BR.: CY7C601	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
backward taken	0.3	0.3	5.8	5.8	1791	1791
backward untaken	0.0	0.0	0.5	0.5	153	153
forward taken	1.7	2.1	34.7	34.7	10776	10776
forward untaken	2.9	3.5	59.1	59.1	18344	18344
total	4.9	5.9	100.0	100.0	31064	31064

JMPL	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
call (jmpl)	0.0	0.0	0.1	0.1	24	12
ret	0.0	0.0	0.3	0.3	48	24
retl	2.7	1.6	99.5	99.5	17090	8545
jmp	0.0	0.0	0.0	0.0	8	4
other jmpl	0.0	0.0	0.0	0.0	0	0

total	2.7	1.6	100.0	100.0	17170	8585

OTHER INSTRUCTIONS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
save	0.0	0.1	1.5	1.5	270	270
restore	0.0	0.1	1.4	1.4	267	267
ticc untaken	2.8	3.4	97.1	97.1	17931	17931
other	0.0	0.0	0.0	0.0	0	0
total	2.9	3.5	100.0	100.0	18468	18468

TRAP CYCLES	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
overflow trap	0.0	0.0	14.3	14.3	24	6
underflow trap	0.0	0.0	11.9	11.9	20	5
system call trap	0.0	0.0	73.8	73.8	124	31
other traps	0.0	0.0	0.0	0.0	0	0
total	0.0	0.0	100.0	100.0	168	42

WINDOW HANDLERS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
overflow	0.2	0.0	65.0	54.5	1020	6
underflow	0.1	0.0	35.0	45.5	550	5
flush	0.0	0.0	0.0	0.0	0	0
total	0.2	0.0	100.0	100.0	1570	11

WINDOW SIZES	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
trace	0.0	0.0	0.0	0.0	0	0
2 windows	11.9	0.1	100.0	100.0	75270	537
3 windows	5.2	0.0	43.5	43.4	32710	233
4 windows	3.3	0.0	27.5	27.4	20670	147
5 windows	1.7	0.0	14.4	14.3	10870	77
6 windows	0.6	0.0	4.8	4.7	3590	25
7 windows	0.4	0.0	3.1	3.0	2300	16
8 windows	0.2	0.0	2.1	2.0	1570	11
9 windows	0.1	0.0	1.1	1.1	840	6
10 windows	0.0	0.0	0.4	0.4	280	2
11 windows	0.0	0.0	0.0	0.0	0	0
12 windows	0.0	0.0	0.0	0.0	0	0
13 windows	0.0	0.0	0.0	0.0	0	0
14 windows	0.0	0.0	0.0	0.0	0	0
15 windows	0.0	0.0	0.0	0.0	0	0
16 windows	0.0	0.0	0.0	0.0	0	0
total	−	−	−	−	−	−

CACHE CYCLES: SS1	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
I-read miss	2.2	0.2	49.0	24.7	13596	1133
D-read miss	1.3	0.1	29.3	14.8	8126	678
D-write miss	0.7	0.4	16.7	50.4	4630	2315
write buffer stalls	0.2	0.1	5.1	10.1	1422	466
total	4.4	0.9	100.0	100.0	27774	4592

CACHE CYCLES: SS2	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
I-read miss	2.5	0.1	43.7	19.9	15492	633
D-read miss	1.5	0.1	27.4	12.5	9724	400
D-write miss	1.6	0.4	28.5	65.2	10101	2080
write buffer stalls	0.0	0.0	0.3	2.4	113	75
total	5.6	0.6	100.0	100.0	35430	3188

APPENDIX F. SPA RESULTS OF PSEUDO CODE TRACE

Spanner - Sparc performance analyzer
_____

cpu:	cy7c601
cache:	ss2
register windows:	8
overflow cost:	170 cycles
underflow cost:	110 cycles

OVERALL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
instructions	76.6	100.0	76.6	-	466217	376324
annulled delay slots	0.7	1.1	0.7	-	4053	4053
load-use stalls	4.6	7.4	4.6	-	27914	27914
trap cycles	0.1	0.0	0.1	-	564	141
window handlers	1.0	0.0	1.0	-	6050	43
cache cycles	17.1	2.4	17.1	-	104153	9212
total	100.0	-	100.0	-	608951	-

INSTRUCTIONS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
memory access	24.2	16.6	31.6	16.6	147185	62332
alu	35.5	57.5	46.4	57.5	216411	216411
floating point	0.4	0.6	0.5	0.6	2397	2397
control transfer	14.0	21.3	18.2	21.3	85062	80022
other instructions	2.5	4.0	3.3	4.0	15162	15162
total	76.6	100.0	100.0	100.0	466217	376324

MEMORY ACCESS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
load	15.0	11.8	62.1	71.1	91406	44303
store	9.2	4.8	37.9	28.9	55775	18028
atomic	0.0	0.0	0.0	0.0	4	1
total	24.2	16.6	100.0	100.0	147185	62332

LOAD	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
ldb	5.6	4.5	37.3	38.5	34136	17068
ldh	0.1	0.1	0.6	0.6	510	255
ld	7.8	6.3	52.2	53.9	47750	23875
ldd	0.1	0.0	0.4	0.3	363	121
ldf	0.1	0.1	0.7	0.7	610	305
lddf	1.3	0.7	8.8	6.0	8037	2679
total	15.0	11.8	100.0	100.0	91406	44303

STORE	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
stb	2.6	1.4	28.1	28.9	15645	5215
sth	0.0	0.0	0.1	0.1	75	25
st	5.4	2.9	58.6	60.5	32697	10899
std	0.2	0.1	2.1	1.6	1148	287
stf	0.1	0.1	1.1	1.1	594	198
stdf	0.9	0.4	10.1	7.8	5616	1404
total	9.2	4.8	100.0	100.0	55775	18028

ALU	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
arithmetic	13.7	22.1	38.5	38.5	83343	83343
logical	17.0	27.6	47.9	47.9	103743	103743
shift	1.3	2.0	3.5	3.5	7644	7644
multiply	1.9	3.0	5.2	5.2	11278	11278
sethi	1.7	2.8	4.8	4.8	10403	10403
total	35.5	57.5	100.0	100.0	216411	216411

ARITHMETIC	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
add	3.3	5.3	24.1	24.1	20126	20126
addcc	0.8	1.3	5.9	5.9	4915	4915
addx	0.0	0.1	0.2	0.2	189	189
addxcc	0.0	0.0	0.0	0.0	0	0
sub	0.3	0.5	2.3	2.3	1947	1947
subcc	1.6	2.7	12.0	12.0	10003	10003
subx	0.0	0.0	0.0	0.0	0	0
subxcc	0.0	0.0	0.0	0.0	0	0
taddcc	0.0	0.0	0.0	0.0	0	0
taddcctv	0.0	0.0	0.0	0.0	0	0
tsubcc	0.0	0.0	0.0	0.0	0	0
tsubcctv	0.0	0.0	0.0	0.0	0	0
cmp (subcc)	6.1	9.9	44.6	44.6	37149	37149
tst (subcc)	1.5	2.4	10.8	10.8	9014	9014
total	13.7	22.1	100.0	100.0	83343	83343

LOGICAL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
and	0.3	0.6	2.0	2.0	2123	2123
andcc	0.9	1.4	5.2	5.2	5428	5428
andn	0.0	0.1	0.3	0.3	282	282
andncc	0.1	0.2	0.7	0.7	763	763
or	2.4	3.9	14.1	14.1	14624	14624
orcc	0.2	0.3	1.0	1.0	1048	1048
orn	0.0	0.0	0.0	0.0	0	0
orncc	0.0	0.0	0.0	0.0	0	0
xor	0.2	0.4	1.5	1.5	1505	1505
xorcc	0.0	0.0	0.0	0.0	0	0
xorn	0.0	0.0	0.0	0.0	0	0
xorncc	0.0	0.0	0.0	0.0	0	0
mov (or)	12.0	19.5	70.6	70.6	73213	73213
tst (orcc)	0.8	1.3	4.6	4.6	4757	4757
total	17.0	27.6	100.0	100.0	103743	103743

SHIFT	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
left	0.8	1.3	62.3	62.3	4761	4761
right logical	0.3	0.5	25.4	25.4	1944	1944
right arithmetic	0.2	0.2	12.3	12.3	939	939
total	1.3	2.0	100.0	100.0	7644	7644

MULTIPLY	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
single step	1.6	2.6	86.8	86.8	9794	9794
read y	0.1	0.2	6.6	6.6	742	742
write y	0.1	0.2	6.6	6.6	742	742
total	1.9	3.0	100.0	100.0	11278	11278

SETHI	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
sethi	1.5	2.4	86.6	86.6	9014	9014
nop	0.2	0.4	13.4	13.4	1389	1389
total	1.7	2.8	100.0	100.0	10403	10403

CONTROL TRANSFER	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
conditional branch	10.2	16.5	72.9	77.5	61981	61981
unconditional branch	1.3	2.1	9.4	10.0	8019	8019
jmpl	1.7	1.3	11.9	6.3	10080	5040
call	0.8	1.3	5.9	6.2	4982	4982
total	14.0	21.3	100.0	100.0	85062	80022

COND. BR.: MB86901	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
backward taken	1.4	2.3	8.9	13.7	8475	8475
backward untaken	0.2	0.2	1.6	1.2	1474	737
forward taken	3.4	5.5	21.9	33.5	20735	20735
forward untaken	10.5	8.5	67.6	51.7	64068	32034
total	15.6	16.5	100.0	100.0	94752	61981

COND. BR.: CY7C601	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
backward taken	1.4	2.3	13.7	13.7	8475	8475
backward untaken	0.1	0.2	1.2	1.2	737	737
forward taken	3.4	5.5	33.5	33.5	20735	20735
forward untaken	5.3	8.5	51.7	51.7	32034	32034
total	10.2	16.5	100.0	100.0	61981	61981

JMPL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
call (jmpl)	0.0	0.0	0.2	0.2	24	12

	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
ret	0.2	0.1	10.3	10.3	1034	517
retl	1.5	1.2	88.7	88.7	8944	4472
jmp	0.0	0.0	0.4	0.4	44	22
other jmpl	0.0	0.0	0.3	0.3	34	17
total	1.7	1.3	100.0	100.0	10080	5040

OTHER INSTRUCTIONS	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
save	0.5	0.8	18.7	18.7	2833	2833
restore	0.5	0.8	18.7	18.7	2830	2830
ticc untaken	1.6	2.5	62.7	62.7	9499	9499
other	0.0	0.0	0.0	0.0	0	0
total	2.5	4.0	100.0	100.0	15162	15162

TRAP CYCLES	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
overflow trap	0.0	0.0	15.6	15.6	88	22
underflow trap	0.0	0.0	14.9	14.9	84	21
system call trap	0.1	0.0	69.5	69.5	392	98
other traps	0.0	0.0	0.0	0.0	0	0
total	0.1	0.0	100.0	100.0	564	141

WINDOW HANDLERS	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
overflow	0.6	0.0	61.8	51.2	3740	22
underflow	0.4	0.0	38.2	48.8	2310	21
flush	0.0	0.0	0.0	0.0	0	0
total	1.0	0.0	100.0	100.0	6050	43

WINDOW SIZES	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
trace	0.0	0.0	0.0	0.0	0	0
2 windows	130.2	1.5	100.0	100.0	792910	5663
3 windows	49.2	0.6	37.8	37.8	299830	2141
4 windows	27.6	0.3	21.2	21.2	168230	1201
5 windows	15.8	0.2	12.1	12.1	96270	687
6 windows	7.9	0.1	6.1	6.1	48390	345
7 windows	1.4	0.0	1.1	1.1	8460	60
8 windows	1.0	0.0	0.8	0.8	6050	43
9 windows	0.7	0.0	0.5	0.5	4200	30
10 windows	0.5	0.0	0.4	0.4	3080	22
11 windows	0.4	0.0	0.3	0.3	2520	18
12 windows	0.4	0.0	0.3	0.3	2240	16
13 windows	0.3	0.0	0.2	0.2	1960	14
14 windows	0.3	0.0	0.2	0.2	1680	12
15 windows	0.2	0.0	0.2	0.2	1400	10
16 windows	0.2	0.0	0.1	0.1	1120	8
total	-	-	-	-	-	-

CACHE CYCLES: SS1	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
I-read miss	6.2	0.8	45.2	19.9	37560	3130
D-read miss	2.6	0.3	18.9	8.3	15722	1311

D-write miss	2.0	1.6	14.8	38.9	12250	6125
write buffer stalls	2.9	1.4	21.1	32.9	17518	5172
total	13.6	4.2	100.0	100.0	83050	15738

CACHE CYCLES: SS2	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
I-read miss	8.4	0.6	49.1	22.6	51157	2085
D-read miss	4.3	0.3	24.9	11.6	25963	1068
D-write miss	4.3	1.5	24.9	59.7	25981	5497
write buffer stalls	0.2	0.1	1.0	6.1	1052	562
total	17.1	2.4	100.0	100.0	104153	9212

118

APPENDIX G.   SPA RESULTS OF TRAJECTORY PROGRAM TRACE

Spanner - Sparc performance analyzer
_____

```
        cpu:                    cy7c601
        cache:                      ss2
        register windows:             8
        overflow cost:      170 cycles
        underflow cost:     110 cycles
```

```
*******************************************************************************
*                                                                             *
* WARNING: More than 1% of instructions are floating point instructions.      *
*     Spanner does not simulate the floating point pipeline.                  *
*                                                                             *
*******************************************************************************
```

OVERALL	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
instructions	77.1	100.0	77.1	–	598746	489447
annulled delay slots	0.5	0.8	0.5	–	3731	3731
load-use stalls	3.4	5.3	3.4	–	26141	26141
trap cycles	0.2	0.1	0.2	–	1672	418
window handlers	1.5	0.0	1.5	–	11540	82
cache cycles	17.3	1.9	17.3	–	134443	9513
total	100.0	–	100.0	–	776273	–

INSTRUCTIONS	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
memory access	21.9	14.0	28.3	14.0	169647	68638
alu	37.9	60.0	49.1	60.0	293876	293876
floating point	0.9	1.4	1.1	1.4	6673	6673
control transfer	14.8	21.8	19.2	21.8	115123	106833
other instructions	1.7	2.7	2.2	2.7	13427	13427
total	77.1	100.0	100.0	100.0	598746	489447

MEMORY ACCESS	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
load	12.3	9.1	56.1	65.2	95168	44774
store	9.6	4.9	43.9	34.8	74475	23863
atomic	0.0	0.0	0.0	0.0	4	1
total	21.9	14.0	100.0	100.0	169647	68638

LOAD	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
ldb	4.6	3.6	37.5	39.9	35728	17864
ldh	0.1	0.1	0.5	0.6	512	256
ld	5.2	4.2	42.8	45.5	40706	20353
ldd	0.0	0.0	0.1	0.0	51	17

	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
ldf	0.2	0.1	1.4	1.5	1362	681
lddf	2.2	1.1	17.7	12.5	16809	5603
total	12.3	9.1	100.0	100.0	95168	44774

STORE	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
stb	3.5	1.9	36.6	38.1	27285	9095
sth	0.0	0.0	0.1	0.1	75	25
st	4.4	2.4	46.4	48.2	34536	11512
std	0.0	0.0	0.3	0.3	260	65
stf	0.1	0.1	1.4	1.4	1035	345
stdf	1.5	0.6	15.2	11.8	11284	2821
total	9.6	4.9	100.0	100.0	74475	23863

ALU	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
arithmetic	14.5	23.0	38.3	38.3	112649	112649
logical	20.5	32.5	54.1	54.1	158945	158945
shift	0.5	0.8	1.4	1.4	4070	4070
multiply	0.7	1.1	1.8	1.8	5173	5173
sethi	1.7	2.7	4.4	4.4	13039	13039
total	37.9	60.0	100.0	100.0	293876	293876

ARITHMETIC	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
add	2.7	4.2	18.4	18.4	20672	20672
addcc	0.8	1.3	5.8	5.8	6532	6532
addx	0.0	0.0	0.0	0.0	0	0
addxcc	0.0	0.0	0.0	0.0	0	0
sub	0.4	0.6	2.5	2.5	2832	2832
subcc	2.4	3.8	16.4	16.4	18502	18502
subx	0.0	0.0	0.0	0.0	0	0
subxcc	0.0	0.0	0.0	0.0	0	0
taddcc	0.0	0.0	0.0	0.0	0	0
taddcctv	0.0	0.0	0.0	0.0	0	0
tsubcc	0.0	0.0	0.0	0.0	0	0
tsubcctv	0.0	0.0	0.0	0.0	0	0
cmp (subcc)	6.7	10.6	45.9	45.9	51726	51726
tst (subcc)	1.6	2.5	11.0	11.0	12385	12385
total	14.5	23.0	100.0	100.0	112649	112649

LOGICAL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
and	0.7	1.1	3.3	3.3	5323	5323
andcc	0.5	0.8	2.4	2.4	3791	3791
andn	0.0	0.0	0.1	0.1	88	88
andncc	0.1	0.1	0.2	0.2	390	390
or	2.3	3.7	11.4	11.4	18165	18165
orcc	0.2	0.3	1.0	1.0	1663	1663
orn	0.0	0.0	0.0	0.0	0	0
orncc	0.0	0.0	0.0	0.0	0	0
xor	0.6	1.0	3.0	3.0	4785	4785
xorcc	0.0	0.0	0.0	0.0	0	0

	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
xorn	0.0	0.0	0.0	0.0	0	0
xorncc	0.0	0.0	0.0	0.0	0	0
mov (or)	15.5	24.7	75.9	75.9	120701	120701
tst (orcc)	0.5	0.8	2.5	2.5	4039	4039
total	20.5	32.5	100.0	100.0	158945	158945

SHIFT	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
left	0.2	0.3	36.9	36.9	1500	1500
right logical	0.3	0.4	50.0	50.0	2036	2036
right arithmetic	0.1	0.1	13.1	13.1	534	534
total	0.5	0.8	100.0	100.0	4070	4070

MULTIPLY	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
single step	0.6	0.9	86.3	86.3	4463	4463
read y	0.0	0.1	6.9	6.9	355	355
write y	0.0	0.1	6.9	6.9	355	355
total	0.7	1.1	100.0	100.0	5173	5173

SETHI	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
sethi	1.5	2.4	88.8	88.8	11584	11584
nop	0.2	0.3	11.2	11.2	1455	1455
total	1.7	2.7	100.0	100.0	13039	13039

CONTROL TRANSFER	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
conditional branch	10.2	16.2	68.9	74.2	79264	79264
unconditional branch	1.4	2.3	9.7	10.5	11211	11211
jmpl	2.1	1.7	14.4	7.8	16580	8290
call	1.0	1.6	7.0	7.6	8068	8068
total	14.8	21.8	100.0	100.0	115123	106833

COND. BR.: MB86901	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
backward taken	0.9	1.5	6.7	9.1	7244	7244
backward untaken	0.3	0.2	2.1	1.4	2286	1143
forward taken	5.6	8.9	40.2	54.7	43390	43390
forward untaken	7.1	5.6	51.0	34.7	54974	27487
total	13.9	16.2	100.0	100.0	107894	79264

COND. BR.: CY7C601	overall (%) cycles	inst.	category (%) cycles	count	raw cycles	count
backward taken	0.9	1.5	9.1	9.1	7244	7244
backward untaken	0.1	0.2	1.4	1.4	1143	1143
forward taken	5.6	8.9	54.7	54.7	43390	43390
forward untaken	3.5	5.6	34.7	34.7	27487	27487
total	10.2	16.2	100.0	100.0	79264	79264

JMPL	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
call (jmpl)	0.0	0.0	0.1	0.1	24	12
ret	0.1	0.1	5.1	5.1	840	420
retl	2.0	1.6	92.3	92.3	15306	7653
jmp	0.0	0.0	1.3	1.3	216	108
other jmpl	0.0	0.0	1.2	1.2	194	97
total	2.1	1.7	100.0	100.0	16580	8290

OTHER INSTRUCTIONS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
save	0.7	1.1	40.7	40.7	5463	5463
restore	0.7	1.1	40.7	40.7	5459	5459
ticc untaken	0.3	0.5	18.7	18.7	2505	2505
other	0.0	0.0	0.0	0.0	0	0
total	1.7	2.7	100.0	100.0	13427	13427

TRAP CYCLES	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
overflow trap	0.0	0.0	10.0	10.0	168	42
underflow trap	0.0	0.0	9.6	9.6	160	40
system call trap	0.2	0.1	80.1	80.1	1340	335
other traps	.'	0.0	0.2	0.2	4	1
total	0.2	0.1	100.0	100.0	1672	418

WINDOW HANDLERS	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
overflow	0.9	0.0	61.9	51.2	7140	42
underflow	0.6	0.0	38.1	48.8	4400	40
flush	0.0	0.0	0.0	0.0	0	0
total	1.5	0.0	100.0	100.0	11540	82

WINDOW SIZES	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
trace	0.0	0.0	0.0	0.0	0	0
2 windows	197.0	2.2	100.0	100.0	1529200	10922
3 windows	71.9	0.8	36.5	36.5	557880	3984
4 windows	47.7	0.5	24.2	24.2	370560	2646
5 windows	28.9	0.3	14.7	14.7	224680	1604
6 windows	15.9	0.2	8.0	8.0	123040	878
7 windows	3.9	0.0	2.0	2.0	30190	215
8 windows	1.5	0.0	0.8	0.8	11540	82
9 windows	0.5	0.0	0.3	0.3	4090	29
10 windows	0.4	0.0	0.2	0.2	2970	21
11 windows	0.3	0.0	0.2	0.2	2410	17
12 windows	0.3	0.0	0.1	0.1	2130	15
13 windows	0.2	0.0	0.1	0.1	1850	13
14 windows	0.2	0.0	0.1	0.1	1570	11
15 windows	0.2	0.0	0.1	0.1	1290	9
16 windows	0.1	0.0	0.1	0.1	1010	7
total	–	–	–	–	–	–

CACHE CYCLES: SS1	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
I-read miss	8.7	1.1	58.2	31.7	67428	5619
D-read miss	1.8	0.2	11.9	6.5	13790	1150
D-write miss	1.0	0.8	6.7	21.9	7750	3875
write buffer stalls	3.5	1.4	23.2	40.0	26866	7082
total	14.9	3.6	100.0	100.0	115834	17726

CACHE CYCLES: SS2	overall (%)		category (%)		raw	
	cycles	inst.	cycles	count	cycles	count
I-read miss	12.0	0.8	69.5	40.2	93389	3820
D-read miss	2.8	0.2	16.1	9.3	21653	887
D-write miss	2.0	0.7	11.8	35.1	15807	3336
write buffer stalls	0.5	0.3	2.7	15.5	3594	1470
total	17.3	1.9	100.0	100.0	134443	9513

```
****************************************************************
*                                                              *
* WARNING: More than 1% of instructions are floating point instructions.  *
*      Spanner does not simulate the floating point pipeline.   *
*                                                              *
****************************************************************
```

LIST OF REFERENCES

[AAD90] D. Alpert, A. Averbuch, and O. Danieli, "Performance Comparison of Load/Store and Symmetric Instruction Set Architectures", *Proc. of The 17th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, pp.172-181, May 1990.

[ABMP91] A. Agrawal, E. W. Brown, D. Murata, and J. Pelatino, "Bipolar ECL Implementation of SPARC", *The SPARC Technical Papers,* Springer-Verlag, USA, 1991.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry, "The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy", *Proc. of the 18th International Symposium on Computer Architecture, Computer Architecture News,* Vol. 19, No. 3, pp.330-339, May 1991.

[Dei90] Harvey M. Deitel, "Hardware, Software, Firmware", *Operating Systems, 2nd edition,* Addison-Wesley Publishing Company, USA, 1990.

[DW90]   Jack W. Davidson and David B. Whalley, "Reducing the Cost Branches by Using Registers", *Proc. of The 17th Annual International Symposium on Computer Architecture,* pp.182-191, IEEE Computer Society, Los Almitos, California, May 1990.

[FM87]   Borivoje Furht and Veljko Milutinovic, "A Survey of Microprocessor Architectures for Memory Management", *COMPUTER,* March 1987.

[Gar91] Robert B. Garner, "The Scalable Processor Architecture (SPARC)", *The SPARC Technical Papers,* Springer-Verlag, USA, 1991.

[GM87]   Charles E. Gimarc and Veljko M. Milutinovic, "A Survey of RISC Processors and Computers of the Mid-1980s", *COMPUTER,* Vol. 24, No. 9, pp.59-68, September 1987.

[Gro90] Evan O. Grossman, "Intel's RISC Chip: Better Late than Never", *PC WEEK/SPECIAL EDITION/RISC,* April 2, 1990.

[HP90]   John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[Ibb90]   Roland N. Ibbett, *The Architecture of High Performance Computers*, Springer-Verlag, Inc., New York, 1990.

[Jou90]   Norman P. Jouppi, "Improving Direct Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", *Proc. of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[Kan87]   Gerry Kane, *MIPS R200 RISC Architecture*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[Kro81]   David Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization", *Proc. 8th International Symposium on Computer Architecture*, pp.81-85, June 1981.

[LFK90]   Gideon Langholz, Joan Francioni, and Abraham Kandel, *Elements of Computer Organization*, Prentice Hall, Inc. Englewood Cliff, New Jersey, 1990.

[Met90]   Dave Methvin, "RISC-Based Systems Find Strength in Simplicity", *PC WEEK/SPECIAL EDITION/RISC*, April 2, 1990.

[NA91]   Masood Namjoo and Anant Agrawal, "Implementing SPARC: A High Performance 32-Bit RISC Microprocessor", *The SPARC Technical Papers*, Springer-Verlag, USA, 1991.

[Por89]   Allan Kenned Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications", PhD Dissertation, UMI Dissertation Service Ann Harbor, Michigan, 1989.

[RT88]   A RISC Tutorial, Sun Microsystems, Inc., Mountain View, California, 1988.

[SC91]   Harold S. Stone and John Cocke, "Computer Architecture in the 1990s", *COMPUTER*, Vol. 20, No. 9, pp.30-37, September 1991.

[SD91]   C. Scheurich and M. Dubois, "Lockup-free Caches in High-Performance Multiprocessors", *Journal of Parallel and Distributed Computing*, pp.25-36, Academic Press, Inc., 1991.

[SPA88]   *The SPARC Techinical Manual*, Sun Microsystems, Inc., Mountain View, California, 1988.

[TT91]   Bill Tuthill and Richard Tuck, "A RISC Tutorial", *The SPARC Techinical Papers*, Springer-Verlag, USA, 1991.

126

INITIAL DISTRIBUTION LIST

Defense Technical Information Center   2
Cameron Station
Alexandria, VA   221314

Dudley Knox Library   2
Code 0142
Naval Postgraduate School
Monterey, CA   93943

Director of Research Administration   1
Code 012
Naval Postgraduate School
Monterey, CA   93943

Chairman, Code CS   2
Computer Science Department
Naval Postgraduate School
Monterey, CA   93943

Dr. Amr Zaky   2
Computer Science Department (CS/Za)
Naval Postgraduate School
Monterey, CA   93943

Dr. Michael L. Nelson   1
Computer Science Department (CS/Ne)
Naval Postgraduate School
Monterey, CA   93943

Captain Leonard Tharpe   3
717 Powell St.
Paris, TN   38242