A Method for System Performance Analysis of the SuperSPARC Microprocessor

by

Melissa C Kwok

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1994

Author .....
Department of Electrical Engineering and Computer Science
May, 1994

Certified by .....
Peter Ostrin
Texas Instruments Incorporated
Thesis Supervisor

Certified by .....
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Fredric R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Method for System Performance Analysis of the SuperSPARC Microprocessor

by

Melissa C Kwok

Submitted to the Department of Electrical Engineering and Computer Science
on May, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Science

Abstract

Methodologies and apparatus setups which allow the performance analysis of SuperSPARC systems is the subject of this thesis. To achieve precise and comprehensive analysis, a set of thirty-five system performance parameters were defined via the utilization of the SuperSPARC PIPE signals. A hardware buffer and software environment were implemented to monitor systems unobtrusively. The defined methodologies and apparatus setups are modular and portable which allow the analysis of any arbitrary SuperSPARC system with any user workload. This thesis project provides an extremely efficient and effective setting for performance analysis.

Thesis Supervisor: Peter Ostrin
Title: Texas Instruments Incorporated

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank Peter Ostrin and Professor Stephen Ward for guidance and support throughout the duration of this thesis. Their patience, insight, and understanding helped make this all possible. I would like to thank Steven Krueger, Jim Carlo, Lisha Doucet, Larry Moriarty, other members of exas Instruments SuperSPARC Applications Group, and Adam Malamy of Sun Microsystems for providing a great research opportunity, background material, and needed resources.

I would also like to thank all the members of the NuMesh team for their friendship and support. Finally, I would like to give special thanks to my family for their support and encouragement over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The SuperSPARC Microprocessor

The Texas Instruments SuperSPARC ™ Microprocessor is a single-pipelined, three-way, dynamic superscalar microprocessor. It is composed of 3.1 million transistors implemented in 0.8μ triple-layer-metal salacided BiCMOS technology. The die is 15.98 x 15.98 mm^2, with 166 active pins and 127 powerground pins [11] [12]. The SuperSPARC processor is a highly integrated implementation of the SPARC RISC architecture, and is fully compatible with the SPARC Architecture, version 8, from SPARC International.

The processor contains a 32-bit integer pipeline unit (IU), an IEEE-compatible double-precision floating point unit (FU), a SPARC reference memory management unit (MMU), a 20K-Byte instruction cache (Icache), a 16K-Byte data cache (Dcache), and a bus interface unit (BU). The bus interface supports the SPARC standard MBUS [6]. A block diagram of the SuperSPARC processor is shown in Figure 1-1, while its module configurations are illustrated in Figure 1-2.

This microprocessor features internal parallelism for high performance. This parallelism accelerates all system applications while allowing clock rates to remain manageable. The processor's superscalar execution feature allows up to three instructions to be issued in a single clock cycle, and the 8-stage pipelined instruction process further reduces the execution time by increasing computation throughput. The SuperSPARC is configured with embedded cache structures and external cache support, which reduce or eliminate the need for time

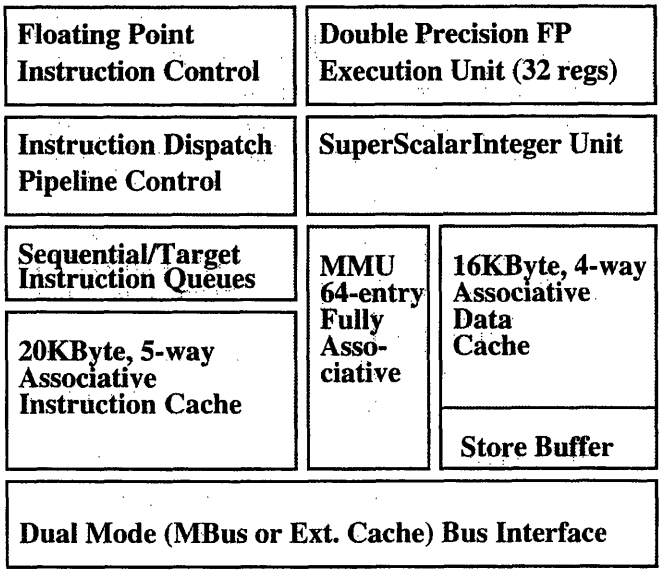


Figure 1-1: SuperSPARC Block Diagram.

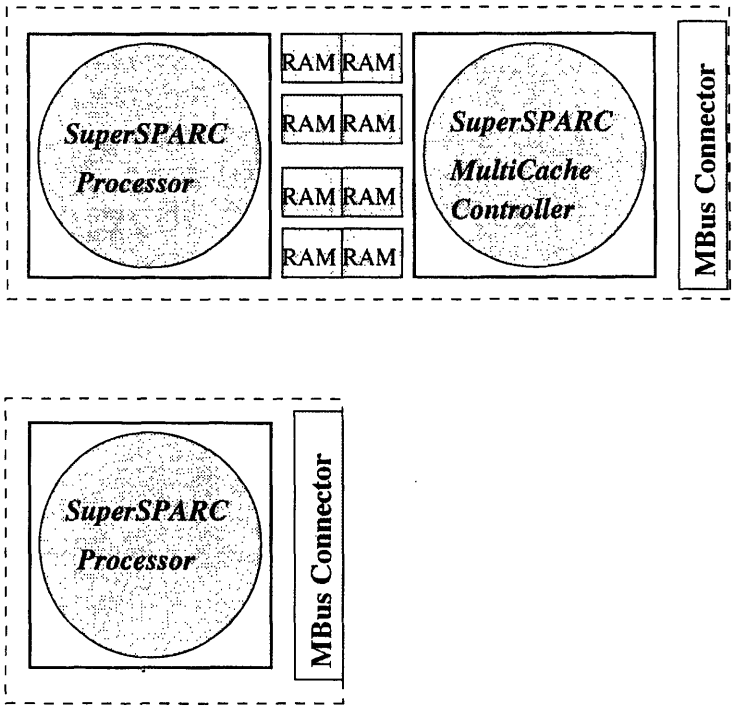


Figure 1-2: SuperSPARC Module Configuration.

consuming external memory references. The SuperSPARC instruction and data caches are fully coherent and physically addressed, thus eliminating flushing overhead. To further boost throughput in floating point applications, the SuperSPARC floating point operations are decoupled from the integer pipeline, and the floating point unit allows one floating point instruction and one floating point load or store to operate in the same cycle [6] [7].

The SuperSPARC enjoys a strong commercial software base. It is installed and operated in a wide variety of system workplaces, including large-scale multiprocessor systems, single-user workstations, and a variety of control applications. SuperSPARC provides an ideal platform for current and future research.

1.2 System Performance Analysis

Since their emergence and rapid success, SuperSPARCs have been installed in a wide variety of systems which differ in size, technology, and cost. Due to this divergence in configurations, system performance of SuperSPARCs with identical workloads could vary enormously across a wide spectrum of performance parameters. It is thus desirable to define a coherent and comprehensive set of performance parameters allowing comparison across various SuperSPARC systems.

Once the SuperSPARC processor is installed and operational in a system workplace, it runs various application workloads. Vendors may wish to apply performance analysis in characterizing the operational bottlenecks for the system. The realized bottleneck parameters may then allow designers to reconfigure the system or alter resource allocations to optimize performance to particular workloads and requirements [9]. It is also useful to have the results of the analysis tabulated, stored, and later used to overcome similar constraints and bottlenecks in future product designs.

The goal of developing an efficient and precise methodology in characterizing system performance of the SuperSPARC processor has motivated this thesis project. The developed methodology should be sufficient to permit performance analysis of any SuperSPARC system with the same procedures and measurement apparatus. Monitoring should be done in an unobtrusive manner, since customer workload may be confidential or proprietary.

Furthermore, the developed methodology should be extensible enough to accommodate the demand for future analysis needs.

1.3 Thesis Overview

This thesis has defined a consistent and coherent set of thirty-five performance parameters for SuperSPARC systems. These parameters are realized by logically combining ten pins on the SuperSPARC microprocessor, called the PIPE pins. The ten PIPE signals are routed from the SuperSPARC internal data path to the external IC contact pins. They provide active information on the processor state without interfering with the execution of user workload. Chapter 2 provides a detailed description of the ten PIPE signals, as well as methods for generating the thirty-five system performance parameters.

A monitoring device must be implemented to capture the thirty-five performance parameters during the execution of a system workload. For the purpose of fast, real-time capturing, a hardware monitor was built. The monitoring device is constructed by connecting a logic analyzer to the ten PIPE pins on the SuperSPARC microprocessor daughter board. The SuperSPARC daughter board is interfaced to a system mother board, which together form the system workplace. A host machine is then connected to the system workplace for program execution, control, and debugging. The entire system is built from subsystems with specified interfaces. Such mechanisms enhance modularity and allow an arbitrary SuperSPARC system to be monitored as long as the interface specifications are met. In this project, the system workplace is composed of a SuperSPARC stand-alone processor daughter board interfacing to a Nimbus NIM6000M mother board. Thus, all measured and reported data in this project are based on this specific hardware system. Chapter 3 describes the monitoring system and details the hardware specifications.

Chapter 4 presents the project's system workload and the logic analyzer programs. A set of four variable-length programs defines the system workload in this project. "test" is a tiny program used solely for the verification of the declared methodology and implemented apparatus setup. "matrix6" and "matrix200" are matrix multiplication programs operating on different size matrices. "dhry" is the standard Dhrystone benchmark program. The

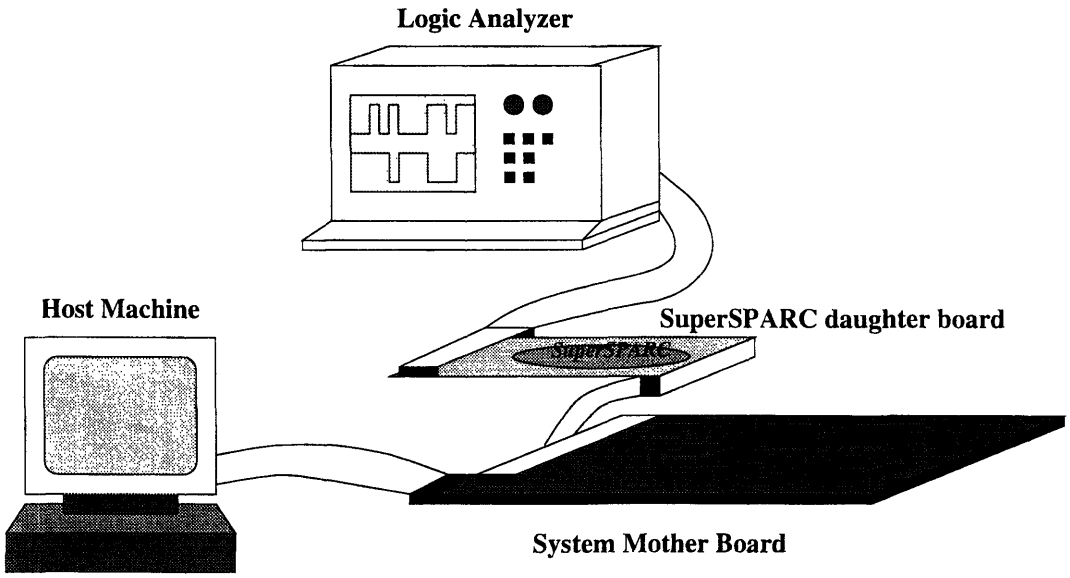


Figure 1-3: Thesis Project System Diagram.

generation of the user workload and all software environment specifications are described in detail. In addition, a set of twelve trigger programs are constructed for the logic analyzer to capture all variable data used to compute system performance parameters.

Chapter 5 derives the set of thirty-five system performance parameters from the raw data. These parameters are compared across the different system workloads. The results are analyzed and the validity of the implemented methodology is discussed. Figure 1-3 illustrates the system diagram for this thesis project.

Chapter 2

SuperSPARC External Monitors in Performance Analysis

This chapter defines the set of thirty-five performance parameters for the analysis of SuperSPARC systems. These parameters are generated from the SuperSPARC external monitors, known as the PIPE pins. In order to fully explain the usage of these monitors in performance analysis, the fundamentals of the SuperSPARC pipeline will first be introduced. Following the pipeline overview, this chapter will present a description of the ten PIPE signals. The definition and the derivation of the thirty-five performance parameters will then be discussed. Finally, a summary of the statistical variables will be presented.

2.1 SuperSPARC Pipeline

This section provides a brief introduction to the operation of the SuperSPARC pipeline. The reader should refer to the Texas Instruments "SuperSPARC User's Guide" for a more detailed description.

Each instruction in the SuperSPARC processor goes through a number of phases in its execution. These phases include fetching, decoding, computing, and result storage. The SuperSPARC's pipeline architecture allows these phases to be overlapped [10] [3]. Thus, a new instruction can be issued at the beginning of each clock cycle rather than waiting for the previous instruction execution to complete all its phases. Multiple executions can be in

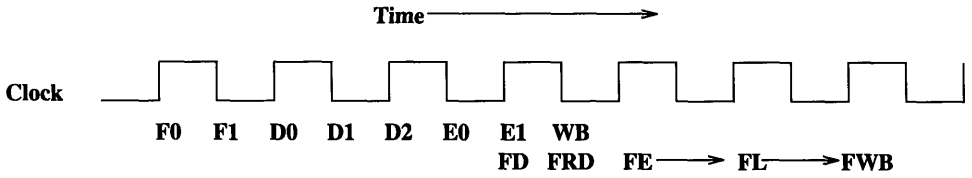


Figure 2-1: SuperSPARC Pipeline Diagram.

progress simultaneously, leading to higher processor throughput.

The SuperSPARC processor's integer pipeline consists of eight stages: two instruction fetching stages, three instruction decoding stages, two execution stages, and one write back stage [6]. Each of the these stages operates in half a clock cycle. Thus the integer pipeline executes in four clock cycles.

The SuperSPARC floating point pipeline, however, consists of five stages: two decode stages, two execution stages, and one write back stage [6]. The decode and write back stages of the floating point pipeline operate in half a clock cycle, while the floating point execution stages operate in one clock cycle. The SuperSPARC floating point pipeline is loosely coupled with the integer pipeline, with a latency of at most three cycles. Figure 2-1 shows the plurality of the integer pipeline stages as well as the floating point pipeline stages.

2.2 SuperSPARC PIPE Pins

The SuperSPARC is configured with large embedded caches. It is possible to execute code continuously from its internal caches with no need for memory references to external devices, making it very difficult to monitor the processor's cycle-by-cycle activities via the indications on the external bus [4] [8].

To monitor the first order behavior of the SuperSPARC processor, ten additional signals have been routed from the integer and floating point datapaths to the external IC contact pins to provide cycle-by-cycle observation for key internal states. These ten external signals are called the PIPE pins (PIPE[9:0]), as they provide information about the pipelined central processor unit (CPU) of the SuperSPARC processor [9]. Information is provided on the following events:

- The number of instructions that complete execution.
- When a branch operation occurs.
- When a branch operation is taken.
- When a memory operation occurs.
- When a floating point operation occurs.
- When the pipeline is held by either the floating point unit or the memory unit.
- When interrupts or exceptions occur.

The following table summarizes these ten PIPE signals and their assertion conditions at each clock cycle.

PIPE[9] Asserted when any valid data memory reference instruction occurred in the execution stage of the previous clock cycle.

PIPE[8] Asserted when any valid floating point operation occurred in the execution stage of the previous clock cycle.

PIPE[7] Asserted when any valid control transfer instruction occurred in the execution stage of the previous clock cycle.

PIPE[6] Asserted when no instructions were available when the instruction group currently at the writeback stage was decoded.

PIPE[5] Asserted when the processor pipeline is stalled by the Data Cache (e.g. data cache miss).

PIPE[4] Asserted when the processor pipeline is stalled by the Floating Point Unit (e.g. floating point memory reference interlock).

PIPE[3] Asserted when the branch in execution stage of the previous cycle was taken.

PIPE[2:1] Indicates the number of instructions in the execution stage of the current cycle: 00=None, 01=1, 10=2, 11=3.

PIPE[0] Asserted when an interrupt or an instruction exception has been incurred.

2.3 PIPE Pins in System Performance Analysis

System performance characteristics can be measured by monitoring the behavior of the PIPE pins. The monitoring takes effect at the execution stage of the pipelined phases, starting at the first instruction in an arbitrary system workload. Tracing stops after the last instruction of the program. This section explores the set of thirty-five performance analyses which can be derived from monitoring these ten external data lines.

2.3.1 Program Execution Time

Program execution time does not involve the PIPE signals. It can be determined by capturing the total number of elapsed clock cycles during the execution of the workload and then multiplying the program cycles by the clock period:

$$ProgramExecutionTime = numProgramCycles * ClockPeriod \tag{2.1}$$

2.3.2 Data Cache Statistics

An external memory reference operation is initiated when the referenced data do not reside in the embedded data cache; i.e., on a data cache miss. Such a memory reference operation stalls the SuperSPARC processor pipeline for a number of cycles until the load operation is complete. Frequent occurrences of data cache misses will degrade system performance [8].

Statistics on the data cache are evolved around the behavior of the PIPE[5] signal. Assertion of PIPE[5] indicates that the processor pipeline is being held by the data cache. By monitoring PIPE[5] and the system clock pin, data cache statistics can be generated.

Capturing the cycles when PIPE[5] is asserted gives rise to the number of pipeline stall cycles that are caused by data cache misses. Dividing the number of data cache miss cycles by the number of program cycles gives the data cache miss cycle rate.

$$numData\$MissCycles = numPIPE[5] \tag{2.2}$$

$$Data\$MissCycleRate = \frac{numData\$MissCycles}{numProgramCycles} \tag{2.3}$$

When a data cache miss occurs, the pipeline is stalled (PIPE[5] is asserted) for many cycles until the referenced data are successfully loaded from the secondary cache or the main memory. At the end of each miss event, PIPE[5] is deasserted. Thus, a data cache miss event is identified by the deassertion of PIPE[5] followed immediately by the assertion of PIPE[5] in the next clock cycle. Counting the number of occurrences of this pattern gives the total number of data cache miss events during the program's execution. The data cache

miss rate can then be determined by dividing the number of miss events by the number of memory operations. Memory operation count, *numMEMop*, is described in section 2.3.5:

$$numData\$MissEvents = num\overline{PIPE[5]}followed\_byPIPE[5] \tag{2.4}$$

$$Data\$MissRate = \frac{numData\$MissEvents}{numMEMop} \tag{2.5}$$

The duration of an arbitrary data cache miss event is indicated by the number of PIPE[5] assertion cycles when the miss event occurs. The average miss event duration throughout the program's execution can be determined by dividing the total number of data cache miss cycles by the total number of data cache miss events in a program execution.

$$Data\$MissDuration = \frac{numData\$MissCycles}{numData\$MissEvents} \tag{2.6}$$

2.3.3 Floating Point Unit Statistics

The SuperSPARC floating point unit may stall the processor pipeline during either a floating point operation or during a floating point memory reference event [5]. The stalling is caused largely by limitations of hardware resources and inefficient software code scheduling. By examining PIPE[4], the behavior of the SuperSPARC floating point unit can be studied.

When the SuperSPARC processor pipeline is stalled by the floating point unit, PIPE[4] is asserted. Thus, the total number of floating point interlock cycles during the program's execution can be determined by capturing all the PIPE[4] assertion cycles. Floating point interlock cycle rate is computed by dividing the floating point interlock cycles by the total program cycles. A floating point interlock event, like a data cache miss event, is identified by the deassertion of PIPE[4] followed immediately by the assertion of PIPE[4] in the subsequent clock cycle. The average floating point interlock duration can then be determined by dividing the floating point interlock cycles by the floating point interlock events:

$$numFPinterlockCycles = numPIPE[4] \tag{2.7}$$

$$FPinterlockCycleRate = \frac{numFPinterlockCycles}{numProgramCycles} \tag{2.8}$$

$$numFPinterlockEvents = num\overline{PIPE[4]}followed\_byPIPE[4] \tag{2.9}$$

$$FPinterlockDuration = \frac{numFPinterlockCycles}{numFPinterlockEvents} \tag{2.10}$$

2.3.4 Stall-Free Cycles

When the SuperSPARC processor pipeline is stalled, instruction streams cease advancing through the pipeline stages. All operations are paused as though the pipeline is frozen. This can either be caused by an external memory reference due to a data cache miss or by the floating point unit interlock. Since both causes may be simultaneously active, a stall-free cycle is a cycle when both causes are not active (i.e. when both PIPE[5] and PIPE[4] are deasserted). A special signal, CYCLE0, is defined to indicate a stall-free cycle. Thus, the number of stall-free cycles during a program's execution is the number of CYCLE0.

$$CYCLE0 = \overline{PIPE[5]} * \overline{PIPE[4]}$$

$$numStallFreeCycles = numCYCLE0 \tag{2.11}$$

Stall-free cycles allow all the pipeline operations to proceed, and all the superscalar processes to progress. The following sections examine the SuperSPARC processor behavior during stall-free cycles.

2.3.5 Instruction Cache Statistics

The SuperSPARC instruction prefetcher continuously fetches instructions to the instruction queue (IQ) [6]. The IQ supplies instruction groups for pipelined execution. The IQ is filled from the instruction cache on an instruction cache hit, and from the main memory on an instruction cache miss. When a cache miss occurs, pipeline bubbles (empty instruction

groups) are fetched from the IQ into the processor pipeline. The pipeline proceeds with the filled bubbles until instructions are available from the IQ. The instruction cache miss phenomenon can be examined by monitoring PIPE[6] and CYCLE0.

PIPE[6] assertion denotes that no instruction resides in the pipeline during the decoding stage. Therefore, a cycle when both PIPE[6] and CYCLE0 (stall-free cycle) are asserted indicates that the processor pipeline is starved with bubble (pipeline still proceeds). Such starvation is a direct result of an instruction cache miss. An instruction cache miss event is realized by the deassertion of PIPE[6] followed immediately by the assertion of PIPE[6] in the next clock cycle when CYCLE0 is asserted in both cycles. The average duration of an instruction cache miss event can be obtained by dividing the number of miss cycles by the number of miss events. The instruction cache miss rate can be measured by dividing the number of instruction miss events by the total number of executed instructions in the program. The method to obtain the total number of executed instructions, *numProgramInstructions*, is described in the next subsection.

$$numI\$MissCycles = num(PIPE[6] * CYCLE0) \tag{2.12}$$

$$I\$MissCycleRate = \frac{numI\$MissCycles}{numStallFreeCycles} \tag{2.13}$$

$$numI\$MissEvents = num(CYCLE0 * \overline{PIPE[6]} followed\_by CYCLE0 * PIPE[6]) \tag{2.14}$$

$$I\$MissRate = \frac{numI\$MissEvents}{numProgramInstructions} \tag{2.15}$$

$$I\$MissDuration = \frac{numI\$MissCycles}{numI\$MissEvent} \tag{2.16}$$

When instruction miss events occur, bubbles are inserted, and no instruction is available in the processor pipeline. Thus, instruction cache misses may be a major cause for having zero-instruction groups in the processor pipeline, and can degrade the superscalar performance of the processor.

2.3.6 Instruction Grouping Statistics

User programs may be expressed in some high level languages, and are compiled into machine code to be executed on the system machine. Dynamic instruction counts of user programs (typically standard benchmark programs) can be used to reflect the machine's instruction-set efficiency, to compare compiler differences, or to compare library routine differences [1] [2]. By monitoring SuperSPARC PIPE[2:1], one can determine the total number of executed instructions in a user program. The superscalar feature of the SuperSPARC processor can be studied, and the dominant causes of pipeline bubbles can also be investigated.

The superscalar design of the SuperSPARC allows the processor to issue up to three instructions per clock cycle to the processor pipeline. However, the actual number of instructions executed in each pipeline stage is determined dynamically by examining the next few available instructions. PIPE[2:1] indicates the number of instructions in the execution stage of the current clock cycle: 00=0, 01=1, 10=2, 11=3. By monitoring these two pins during stall-free cycles (when CYCLE0 is asserted), statistics on the size of the executed instruction groups can be gathered:

$$num0InstrGroup = num(\overline{PIPE[2]} * \overline{PIPE[1]} * CYCLE0) \tag{2.17}$$

$$num1InstrGroup = num(\overline{PIPE[2]} * PIPE[1] * CYCLE0) \tag{2.18}$$

$$num2InstrGroup = num(PIPE[2] * \overline{PIPE[1]} * CYCLE0) \tag{2.19}$$

$$num3InstrGroup = num(PIPE[2] * PIPE[1] * CYCLE0) \tag{2.20}$$

Since each stall-free cycle must contain an instruction group with either zero, one, two, or three instructions, the following equation can be used to verify the instruction group statistics:

$$numCYCLE0 = num0InstrGroup + num1InstrGroup + num2InstrGroup + num3InstrGroup$$

The occurrences of the four instruction groups can be combined to calculate the total number of instructions executed in a user program:

$$numProgramInstructions = (num1InstrGroup * 1) + (num2InstrGroup * 2) + (num3InstrGroup * 3) \quad (2.21)$$

Knowing the number of executed instructions as well as the number of elapsed clock cycles, the average instructions per cycle (IPC) and the average cycles per instruction (CPI) can be calculated:

$$IPC = \frac{numProgramInstructions}{numProgramCycles} \quad (2.22)$$

$$CPI = \frac{numProgramCycles}{numProgramInstructions} \quad (2.23)$$

The fractions of each executed instruction group can be conveniently calculated by dividing the number of executed instruction groups by the total number of stall-free cycles:

$$0InstrGroupFraction = \frac{num0InstrGroup}{numCYCLE0} \quad (2.24)$$

$$1InstrGroupFraction = \frac{num1InstrGroup}{numCYCLE0} \quad (2.25)$$

$$2InstrGroupFraction = \frac{num2InstrGroup}{numCYCLE0} \quad (2.26)$$

$$3InstrGroupFraction = \frac{num3InstrGroup}{numCYCLE0} \quad (2.27)$$

2.3.7 Instruction Type Statistics

Due to hardware resources and restrictions, all SuperSPARC instructions are classified into one of four types: arithmetic logic operations, memory operations, branch operations, and floating point operations [6]. It is often desirable to analyze the composition of the instruction types in an executed program.

The assertion of PIPE[9] indicates when there is a valid memory operation in the execution stage of the processor pipeline. Thus, counting the occurrences of PIPE[9] assertion during all stall-free cycles provides the number of memory operations in the user program:

$$numMEMop = num(PIPE[9] * CYCLE0) \tag{2.28}$$

The assertion of PIPE[8] specifies a valid floating point operation in the execution stage. The number of floating point operations in a given user program can therefore be obtained by counting PIPE[8] assertions during stall-free cycles.

$$numFPop = num(PIPE[8] * CYCLE0) \tag{2.29}$$

Similarly, PIPE[7] assertion defines a valid branch operation:

$$numBRop = num(PIPE[7] * CYCLE0) \tag{2.30}$$

Since the total number of executed instructions is known, the number of arithmetic logic operations can be obtained by subtracting the number of other operations from the total number of executed instructions.

$$numALUop = numProgramInstructions - numMEMop - numFPop - numBRop \tag{2.31}$$

PIPE[3] assertion indicates that the branch instruction in the previous cycle is taken. Thus, by counting the number of PIPE[3] assertions during stall-free cycles, the number of taken branch instructions can be determined.

$$numTakenBRop = numPIPE[3] * CYCLE0 \tag{2.32}$$

The number of untaken branch instructions can then be calculated by subtracting the number of taken branch instructions from the number of branch instructions.

$$numUntakenBRop = numBRop - numTakenBRop \tag{2.33}$$

The fraction of taken and untaken branch instructions can then be easily derived:

$$fractionTakenBRop = \frac{numTakenBRop}{numBRop} \tag{2.34}$$

$$fractionUntakenBRop = \frac{numUntakenBRop}{numBRop} \tag{2.35}$$

2.4 Summary of Statistical Variables

This section summarizes all the statistical variables which are gathered by monitoring SuperSPARC PIPE pins. These variables are used to derive the thirty-five performance parameters specified in this chapter. A total of sixteen statistic variables are defined. The following table provides descriptions for each variable.

Variable	Description
$numProgramCycles$	The total number of clock cycles during program execution.
$numData\$MissCycles$	The total number of cycles when the processor pipeline is being held by the data cache.
$numData\$MissEvents$	The total number of data cache miss events.
$numFPinterlockCycles$	The total number of cycles when the processor pipeline is being held by the floating point unit.
$numFPinterlockEvents$	The total number of floating point interlock events.
$numCYCLE0$	The total number of stall-free cycles.
$numI\$MissCycles$	The total number of cycles where there is an instruction cache miss.
$numI\$MissEvents$	The total number of instruction cache miss events.
$num0InstrGroup$	The total number of zero instruction groups.
$num1InstrGroup$	The total number of one instruction groups.
$num2InstrGroup$	The total number of two instruction groups.
$num3InstrGroup$	The total number of three instruction groups.
$numMEMop$	The total number of memory reference operations.
$numFPop$	The total number of floating point operations.
$numBRop$	The total number of branch operations.
$numTakenBRop$	The total number of taken branch operations.

Table 2.1: Statistical Variables

Chapter 3

System Hardware Construction and Specification

System performance can be measured in various ways. In general, there are three types of mechanisms that monitor a sequence of executed instructions: software monitors, hardware monitors, and firmware monitors [4]. In this project, a hardware monitor was chosen as the dynamic measurement mechanism.

The implemented hardware monitor is extremely fast and accurate. It has the ability to perform real-time tracing. Cycle-by-cycle monitoring of the key internal system state is made possible and the captured analysis data are readily available. In addition, the hardware monitor is independent of the system machine. The monitoring is done unobtrusively, and system performance is not degraded by the monitoring.

This chapter describes the configuration and specification of the implemented hardware buffer, and presents the methodology used to capture the desired statistics data.

3.1 Hardware Apparatus Configuration

The hardware buffer implemented in this project consists of four distinct subsystems: a host machine, a system mother board, a microprocessor daughter board, and a logic analyzer. The system monitor is composed of the logic analyzer and its interface to the PIPE pins on the microprocessor daughter board. The system workplace is defined by the system mother

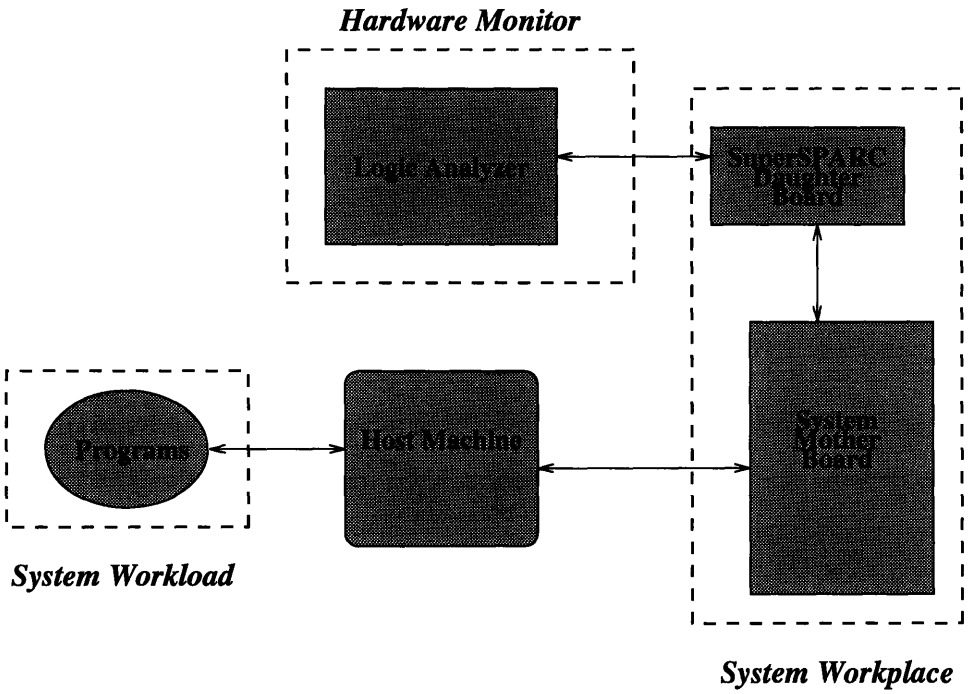


Figure 3-1: Hardware Buffer Block Diagram.

board, the microprocessor daughter board, and the interface between the two boards. The host machine interfaces to the system mother board for the control of program execution and disassembling. The system block diagram for this hardware buffer is illustrated in Figure 3-1.

The following subsections review the design decisions relating to each of the four subsystems. Note that in constructing a hardware buffer, users may replace any of the subsystems with a design of their own, as long as the interface specifications are met and the functionality of the subsystem is implemented.

3.1.1 Host Machine

A Sun SPARCStation 10 is used as the host machine. This host machine acts as an interpretive mechanism to the system machine (the system mother board plus the processor daughter board). It is connected to the system mother board via a standard serial cable. Its functionalities include downloading the executable program to the system machine, controlling program execution, providing terminal emulation through the serial cable to the system machine.

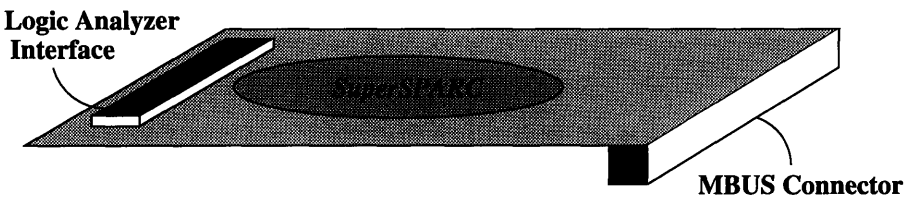


Figure 3-2: SuperSPARC Microprocessor Daughter Board

3.1.2 System Mother Board

The NIMBus NIM6000M is used as the system mother board in this project. It features 96 Mbytes of DRAM on board, a DRAM controller, a serial port for connection with the host machine, and it has the standard form factor of a SUN SPARCStation 2 mother board. Its interface connection allows the horizontal placement of the microprocessor daughter board vertically above the system mother board. The interfacing to the daughter board is realized via a 100-pin MBus interface connector meeting the SPARC International Industry Standard MBus Interface Specifications. This system mother board also features a hardware reset button to allow quick and easy system reset when the host machine is being shared by other resources. During program execution, the system is clocked at 33MHz.

3.1.3 Microprocessor Daughter Board

The SuperSPARC microprocessor daughter board is implemented based on the TMX390Z50-40M module. This TMX390Z50-40M module features the stand-alone SuperSPARC chipset configuration with control circuitry. This module consists of a SuperSPARC microprocessor revision #2.8 mounted on a 3.25 x 5.75 inch printed circuit board. This daughter board consists of two interface connections for interfacing with the system mother board as well as the logic analyzer. Figure 3-2 illustrates the microprocessor daughter board. See appendix D for the schematic of the TMX390Z50-40M module.

Along the 3.25 inch edge of the module is a microstrip connector which allows the interface to the mother board of the system machine. This connector is a one-hundred pin connector referred to as the SPARC International Industry Standard MBus connector. MBus is a synchronous bus architecture used in SPARC systems and is specifically designed for high speed transfer between a SPARC processor and memory.

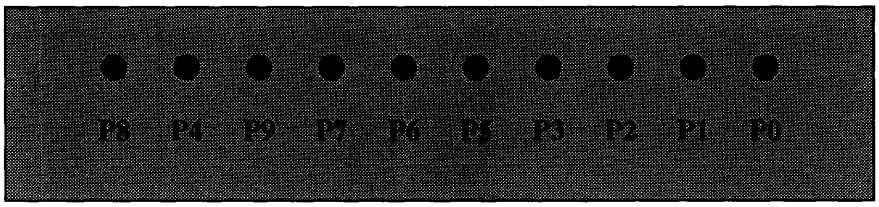


Figure 3-3: SuperSPARC Microprocessor Daughter Board PIPE Pin Interface

The interface to the logic analyzer consists of the ten PIPE signals on the processor as well as forty MBUS signals. A 1 x 3 inch pin connector plate is mounted on top of this microprocessor daughter board for PIPE pin interfacing. This pin connector plate consists of two rows of header pins – ten PIPE IC pins (PIPE[9:0], illustrated in Figure 3-3) and ground pins which form the eleven probing channels shown below. These eleven channels are monitored by the logic analyzer via high-bandwidth channel probes.

In addition to the PIPE pins, thirty-six MBus address/data signals as well as four other MBus control signals are connected to the logic analyzer via the channel probes. This connection is realized by mounting a header pin connector on top of the MBUS microstrip connector. These connections probe the MBus activities and detect the logic analyzer trigger conditions. (See section 3.2)

3.1.4 Logic Analyzer

The logic analyzer constitutes the monitor of the hardware buffer. It is responsible for pin tracing, data storage, real-time monitoring of system behavior, and providing cycle-by-cycle observation of the processor's key internal states. The logic analyzer selected for this project is a Tektronix DAS9200 with 92A96SD acquisition module.

This Tektronix DAS9200 logic analyzer system can handle a wide range of performance analysis. It features a flexible programming environment for system configurations, setups, triggering, and program disassembling. The 92A96SD acquisition module provides 512K bits of memory per channel, with a total of 96 channels. In addition to the memory depth, the module also contains two full speed 32-bit counters to track various events. These two counters are heavily used in gathering data for performance characterization of the SuperSPARC system.

The high-bandwidth channel probes on the logic analyzer are connected to the header pins on the processor daughter board for interfacing. Users should refer to appendix A for the logic analyzer configuration files.

3.2 Hardware Monitor Interface Specification

To monitor the first order behavior of the SuperSPARC processor, the following pins are probed by the hardware monitor via the connections between the logic analyzer channel probes and the processor daughter board header pins:

- PIPE[9:0]
- MCLK0 (connected to the VCLK on SuperSPARC)
- MAD[35:0]
- MAS*, MRDY*, MBB*

PIPE[9:0] are the external monitors on the SuperSPARC, while MCLK0, MAD[35:0], MAS*, MRDY*, and MBB* are signals on the MBus.

The behavior of the ten PIPE signals is used to study the performance of the system machine. These pins are being clocked by the system machine clock, which is the MCLK0 signal on the MBus (or the VCLK on the SuperSPARC).

In addition to the ten PIPE signals and the system clock signal, the thirty-six MBus address lines and three MBus control signals are also being traced to provide information on the MBus activity. The information on the MBus activity is critical when the behavior of the PIPE signals is in doubt. The SPARC Reference MMU Architecture implements 36-bit physical addresses to provide a 64-gigabyte physical address space. Thus, the low order thirty-six bits of the sixty-four bit MBus address/data lines (MAD[35:0]) are monitored to manifest the physical addresses of memory reference operations. The MBus busy signal, MBB*, indicates when the MBus is in use. The MBus address strobe signal, MAS*, indicates when the signal on the MBus is an address instead of data. MRDY*, the MBus data ready signal, is asserted when the signal on the MBus is data.

3.3 Monitoring Methodology

Chapter 2 described the set of thirty-five performance parameters for the analysis of SuperSPARC systems. These performance parameters are derived from sixteen statistic variables. These statistic variables can be captured by monitoring the SuperSPARC PIPE pin patterns in the hardware buffer described in this chapter.

The basic idea in monitoring is to realize that each statistic variable to be captured is recognized as the number of occurrences of certain PIPE pin assertion/deassertion patterns. Each occurrence of a PIPE pin pattern is defined as an event. Thus, the hardware buffer needs to capture the occurrences of all sixteen different events during a program's execution.

The major concern in such a capturing scheme is data storage. The current system workplace operates at 33 MHz. Thus, with the 512K bits of memory per channel on the logic analyzer, it stores data for only a tiny fraction of a second. Most typical benchmark programs, however, run for a least a few minutes.

To compensate for the shortage of storage memory and to avoid adding extra hardware to the monitoring buffer, a different approach is investigated. The thirty-two bit counter on the logic analyzer is used to count the number of occurrences of each event. This is accomplished by specifying a condition for each event to trigger the counter. Counter overflow is incorporated into the final count. Thus, by doing sixteen counts in a program's execution, all statistical variables can be captured without the need to store the PIPE pin patterns for every clock cycle.

A total of fourteen trigger programs are created to specify the sixteen trigger conditions. These programs are described in the next chapter. To capture a particular statistic variable, the corresponding trigger program is run on the logic analyzer during a program's execution. This step is repeated to gather all sixteen variables. The following table summarizes the trigger conditions and the trigger actions for each of the sixteen statistical variables.

Statistic Variables	Trigger Action	Trigger Condition
numProgramCycles	counter value + 1	MCLK0
numData$MissCycles	counter value + 1	PIPE[5]
numData$MissEvents	counter value + 1	$\overline{PIPE[5]}$ followed by PIPE[5]
numFPinterlockCycles	counter value + 1	PIPE[4]
numFPinterlockEvents	counter value + 1	$\overline{PIPE[4]}$ followed by PIPE[4]
numCYCLE0	counter value + 1	$\overline{PIPE[5]}$ * $\overline{PIPE[4]}$
numI$MissCycles	counter value + 1	PIPE[6] * CYCLE0
numI$MissEvents	counter value + 1	$\overline{PIPE[6]}$ followed by PIPE[6] * CYCLE0
num0InstrGroup	counter value + 1	$\overline{PIPE[2]}$ * $\overline{PIPE[1]}$ * CYCLE0
num1InstrGroup	counter value + 1	$\overline{PIPE[2]}$ * PIPE[1] * CYCLE0
num2InstrGroup	counter value + 1	PIPE[2] * $\overline{PIPE[1]}$ * CYCLE0
num3InstrGroup	counter value + 1	PIPE[2] * PIPE[1] * CYCLE0
numMEMop	counter value + 1	PIPE[9] * CYCLE0
numFPop	counter value + 1	PIPE[8] * CYCLE0
numBRop	counter value + 1	PIPE[7] * CYCLE0
numTakenBRop	counter value + 1	PIPE[3] * CYCLE0

Table 3.1: Methodologies for gathering statistical data

Chapter 4

Software Environment Specifications

This chapter describes the software environment required for the course of system performance monitoring. Section 4.1 presents the steps for generating a user program, and describes the procedures for program execution on the system machine. Section 4.2 describes the four programs used to gather performance analysis data. Section 4.3 describes the logic analyzer trigger programs. The presented specification is tailored for the operations on the system machine described in Chapter 3.

4.1 Program Generation and Execution

4.1.1 System Workload and Its Execution

In this project, a workload program is initially written in C and then translated into SPARC assembly code by a SPARC compiler. The assembly language program is augmented with features to trigger the logic analyzer and to ensure a proper operating environment for the SuperSPARC processor. The modified assembly source file is then assembled into an executable object file by using an assembler. Makefiles are created to maintain the object files and to generate executable programs. Appendix C contains the Makefiles, original C source code, and the edited SPARC Assembly source files.

The following shows an example C program, together with its modified assembly code. The process of modifying the assembly file is discussed in the next subsection.

C code:

```
#include<stdio.h>
float random();
main()
{
  int count = 0;
  float number = 1.5;
  while(
.
.
.
}
float random(the_number)
     float the_number;
{
.
.
.
}
```

SPARC assembly code (edited version):

```
#include <machine/asm_linkage.h>
#include <machine/mmu.h>
LL0:
.seg "data"
_romp:  .word   0
.seg "text"
.proc 04
.global __entry
__entry:
        set     0x2000, %o2
        mov     %o2, %sp
        set     _romp, %o1
        st      %o0, [%o1]
lda [%g0]0x04, %g4
set 0x00000700, %g3
or %g4, %g3, %g4
sta %g4, [%g0]0x04 ! Write to MCNTL
set     0x4000, %o0 ! Set Page Map
set     0x4ee,  %o1
call    _SetPageMap, 2
nop
        ldda    [%g0] 0x4c, %g2
        or      %g3, -1, %g3
        stda    %g2,[%g0]0x4c          ! Write to action register
__trigger0:
        set     0x00003000, %g1
        stda    %g0, [%g1] 0x20
_main:
!#PROLOGUE# 0
.
.
.
__trigger1:
        set     0x00003100, %g1
        stda    %g0, [%g1] 0x20
__exit:
        sethi   %hi(_romp),%o0
        ld      [%o0+%lo(_romp)],%o0
        ld      [%o0+0x74],%g1
        call    %g1,0
```

```
        nop
LE31:
ret
restore
        LF31 = -64
LP31 = 64
LST31 = 64
LT31 = 64
.seg "data"
/*
 * Set the page map entry for the virtual address in the first arg (%o0)
 * to the entry specified in the second arg (%o1)
 * Called as SetPageMap(v_addr, pme)
 * ASI_MOD  0x4
 * ASI_MEM  0x20
 */
    ENTRY(SetPageMap)
    andn    %o0, 0x3, %o2          ! align on word
    set     RMMU_CTP_REG, %o5
    lda     [%o5]ASI_MOD, %o0          ! get context table pointer
    set     0xffffc000, %o5     and     %o0, %o5, %o3
    sll     %o3, 0x4, %o3
    set     RMMU_CTX_REG, %o5
    lda     [%o5]ASI_MOD, %o4          ! get context register
    set     0xfffffc00, %o5
    and     %o4, %o5, %o5
    sll     %o5, 2, %o5
    or      %o5, %o3, %o3
    srl     %o0, 0x8, %o0
    srl     %o4, 0xa, %o4
    or      %o4, %o0, %o0
    and     %o0, 0x3f, %o0
    sll     %o0, 0xc, %o0
    or      %o0, %o3, %o3
    lda     [%o3]ASI_MEM, %o0         ! get region pointer
    andn    %o0, 0x1, %o5
    sll     %o5, 0x4, %o3
    set     0xff000000, %o0           ! mask for level 1 offset in vaddr
    and     %o2, %o0, %o0
    srl     %o0, 0x16, %o0
    add     %o0, %o3, %o3             ! get l2 pointer offset in l1 table
    lda     [%o3]ASI_MEM, %o0         ! get segment pointer
    andn    %o0, 0x1, %o5
    sll     %o5, 0x4, %o3
    set     0x00fc0000, %o0           ! mask for level 2 offset in vaddr
    and     %o2, %o0, %o0
    srl     %o0, 0x10, %o0
    add     %o0, %o3, %o3             ! get l3 pointer offset in l2 table
    lda     [%o3]ASI_MEM, %o0         ! get page table pointer
    andn    %o0, 0x1, %o5
    sll     %o5, 0x4, %o3
    set     0x0003f000, %o0           ! mask for l3 offset in vaddr
    and     %o2, %o0, %o0
    srl     %o0, 0x0a, %o0
    add     %o0, %o3, %o3             ! get pte addr in l3 table
    sta     %o1, [%o3]ASI_MEM      ! store entry to physical address
    mov     %o7, %g6               ! save return address
    call    _OnMBus, 0
    nop
    cmp     %o0, 0
    bnz     leave
    nop
    call    _InvalidateEcacheLine, 1
    mov     %o3, %o0
leave:
                                     ! the program
```

```
    retl
    mov     %g6, %o7
leave:
                                        ! the program
    retl
    mov     %g6, %o7
/*
 * Will return 0 in %o0 if the processor is connected to VBus or
 * not zero if the processor is directly on MBus
 */
    ENTRY (OnMBus)
    set     RMMU_CTL_REG, %o5
    lda     [%o5]ASI_MOD, %o4          ! get mmu control reg
    retl
    and     %o4, CPU_VIK_MB, %o0
/*
 * Will invalidate the Ecache line for the address in %o0
 * Assumes MBus module 0. Will not check ANYTHING, just invalidates
 */
    ENTRY (InvalidateEcacheLine)
    set     0xfff00000, %o5
    andn    %o0, %o5, %o4
    set     0xff800000, %o5
    or      %o5, %o4, %o3
    andn    %o3, 0x7, %o3
    ldda    [%o3]ASI_MXCC, %o4
    set     0x2222, %o2
    andn    %o5, %o2, %o5
    retl
    stda    %o4, [%o3]ASI_MXCC
```

When the executable program is ready, it will be run on the system machine as a standalone program. The SPARC Open Boot PROM 2.0 is used on the system machine. Its functionalities include program execution control, system machine interaction, program disassembling and debugging. The Open boot PROM commands are controlled with the Open Boot 2.0 Fourth Monitor. The following lists the commands used in the Open Boot PROM environment during the process of a program execution. Letters following a "!" are comments. All necessary operating environments must be properly setup and stable by the time of the execution.

On the host machine:

```
host_machine% cd _directory_containing_the_executable_program
host_machine% tip _system_machine_name
Type b (boot), c (continue), or n (new command mode)
> n ! to get the ok prompt
ok dlbin ! to download the executable program
(enter: ~C)
> sendbin _executable_program
ok go
 program terminate ! this signals a successively executed program
! i.e. without system error
```

In the case of any programming or environment error, the program will not terminate

properly. Users should refer to the "SPARC Open PROM Toolkit Reference Summary" for the list of commands used in program debugging.

4.1.2 Modifying the Assembly Source File

When programs are written in SPARC assembly language or in some higher level language, they need to be modified to ensure proper operation when executing on the system machine. All the modifications are made to the SPARC assembly files. Readers should refer to "SPARC Assembly Language Reference Manual" for more information on SPARC assembly syntax.

To execute a program in the Open Boot PROM, the stack pointer is set. The specified value stored in register %o0 is loaded into the declared Open Boot variable, _romp, before the program execution. The following four lines of assembly code are inserted before the actual source code:

```
        set     0x2000, %o2
        mov     %o2, %sp
        set     _romp, %o1
        st      %o0, [%o1]
```

The Open Boot PROM variable, _romp, is declared under the data segment:

```
        .seg    "data"
_romp:  .word   0
```

When the program terminates, the utility function stored at an offset 0x74 from the content of _romp is called for the executing program to exit properly. To perform these tasks, the following code needs to be inserted after the source code:

```
__exit:
        sethi   %hi(_romp),%o0
        ld      [%o0+%lo(_romp)],%o0
        ld      [%o0+0x74],%g1
        call    %g1,0
        nop
```

When the user program is written in C and compiled down to assembly code, the above lines will replace the following stack setup code generated by the SPARC compiler:

```
! for stack setup:
!       sethi   %hi(LF27),%g1
!       add     %g1,%lo(LF27),%g1
```

```
!       save    %sp,%g1,%sp
! for stack restore:
!       ret
!       restore
```

To accurately measure performance characterization, the SuperSPARC processor is required to be operating in the desired functional mode [6]. Most importantly, the instruction cache, the data cache, and the store buffer need to be enabled. This is done by setting appropriate values of the SuperSPARC MMU control register (MCNTL). In addition, the SuperSPARC multiple-instruction-per-cycle mode also needs to be enabled by setting the MIX field in the SuperSPARC breakpoint action register. All environment code to perform setups is inserted before the source code as follows:

```
lda     [%g0]0x04, %g4
set     0x00000700, %g3
or      %g4, %g3, %g4
sta     %g4, [%g0]0x04          ! Write to MCNTL
ldda    [%g0] 0x4c, %g2
or      %g3, -1, %g3
stda    %g2,[%g0]0x4c           ! Write to action register
```

Finally, since the Open Boot PROM sets all page maps to read only mode, page maps need to be set to read and write mode before the program execution. This is done by inserting an entry code segment after the actual source code as shown in the example assembly file in the previous section. A call to the set page map entry is placed before the source code:

```
set     0x4000, %o0             ! Set Page Map
set     0x4ee,  %o1
call    _SetPageMap, 2
nop
```

4.1.3 Triggering the Logic Analyzer

It is apparent that the above code editing increases the size of the program. A good triggering mechanism is required for the logic analyzer so that it only performs pin tracing during the actual source code execution.

The logic analyzer starts pin tracing and event counting once it sees a specified physical address on the MBUS address lines, MAD[35:0]. For each instruction or data access, the SuperSPARC instruction unit appends to the 32-bit memory addres an 8-bit address

space identifier, or ASI. To make such an address appear on the MBUS, a double store alternate instruction is issued to store a garbage value into the specified ASI MMU physical pass-through address. This implementation uses the physical address 0x00000300020 for signaling the start of the program, and the physical address 0x00000310020 for signaling the end of the program. The following __trigger0 function is used to mark the start of the program, and is placed after the environment setup code and immediately before the source code:

```
__trigger0:
        set     0x00003000, %g1
        stda    %g0, [%g1] 0x20
```

The __trigger1 function indicates the end of the program. It is placed immediately after the end of the source code and before the __exit function:

```
__trigger1:
        set     0x00003100, %g1
        stda    %g0, [%g1] 0x20
```

4.2 Execution Program Set

Four executable programs are generated for performance analysis [4]. They represent a good mix of general purpose computing. The first program is a very short program used mainly for verifying the hardware and software implementation, as well as the triggering methodology. The second workload in this project is a short matrix program, and the third is a moderate length matrix program. The last program is a standard benchmark program. All programs are included in Appendix C.

test – the original source code is written in C. This is a very small-scale program which contains simple arithmetic operations on an integer variable and a floating point variable. This program is also characterized by repeated loops and function calls. The provided features are sufficient to thoroughly test the system buffer's hardware and software configuration, as well as the algorithms used for gathering statistics.

matrix6 – the source code of this program is written in C. This a floating point intensive program which runs operations on a 6 by 6 matrix. This program contains multiple function calls and structured loops.

matrix200 – this program has the same structure as "matrix6", yet it runs operations on a 200 by 200 matrix. This program is large enough to preclude fitting all executing workset in the embedded caches.

dhry – this is the integer intensive Dhrystone benchmark program. The C version of the Dhrystone program is compiled down to SPARC assembly, which is modified and translated into an executable. 100,000 measurement loops are used. "multiply.o" and "string.o" from the standard UNIX C library are included in the executable program.

All four programs are compiled down to assembly files, modified, then translated into executable code. Analysis results and statistical distributions are presented in the next chapter.

4.3 Logic Analyzer Programming

For the purpose of the analysis, fourteen trigger programs are created for the sixteen statistical variables presented in Chapter 3. These trigger programs are written in a C-like language which contain clauses and statements. The syntax is used to describe state machine structures that control triggering and data sampling actions. The following table lists these trigger programs and their functionalities. All logic analyzer trigger programs are included in Appendix B.

0_instr_gp – this program allows counter #1 to capture each occurrence of a zero_instruction group, or bubble, at the rising edge of each clock cycle. Counter #2 is used to capture the total number of stall-free cycles during program execution.

1_instr_gp – this program allows counter #1 to capture each occurrence of a one_instruction group at the rising edge of each clock cycle. Counter #2 is used to capture the total number of stall-free cycles during program execution.

2_instr_gp – this program allows counter #1 to capture each occurrence of a two_instruction group at the rising edge of each clock cycle. Counter #2 is used to capture the total number of stall-free cycles during program execution.

3_instr_gp – this program allows counter #1 to capture each occurrence of a three_instruction group at the rising edge of each clock cycle. Counter #2 is used to capture the total number

of stall-free cycles during program execution.

BRop_exec – this program allows counter #1 to capture each occurrence of a branch operation at the rising edge of each clock cycle. Counter #2 is used to capture the total number of stall-free cycles during program execution.

FPop_exec – this program allows counter #1 to capture each occurrence of a floating point operation at the rising edge of each clock cycle. Counter #2 is used to capture the total number of stall-free cycles during program execution.

MEMop_exec – this program allows counter #1 to capture each occurrence of a memory reference operation at the rising edge of each clock cycle. Counter #2 is used to capture the total number of stall-free cycles during program execution.

BRop_takn – counter #1 is used to capture the number of occurrences of each branch operations which is taken. Counter #2 is used to capture the total number of branch operations during program execution.

D$_miss_cycle – this program allows counter #1 to count the number of cycles when the processor pipeline is stalled by the data cache. It also allows counter #2 to count the total number of cycles elapsed during program execution.

D$_miss_event – this program allows counter #1 to count the number of data cache miss events. It also allows counter #2 to count the total number of cycles elapsed during program execution.

FP_intrlk_cyc – this program allows counter #1 to count the number of cycles when the processor pipeline is stalled by the floating point unit. It also allows counter #2 to count the total number of cycles elapsed during program execution.

FP_intrlk_evnt – this program allows counter #1 to count the number of floating point interlock events. It also allows counter #2 to count the total number of cycles elapsed during program execution.

I$_miss_cycle – this program allows counter #1 to count the number of cycles when there is an instruction cache miss. It also allows counter #2 to count the number of stall-free cycles during program execution.

I$_miss_event – this program allows counter #1 to count the number of instruction cache miss events. It also allows counter #2 to count the number of stall-free cycles during

program execution.

Chapter 5

Data Analysis

In this chapter, the statistics gathered from the hardware monitoring of the program set are presented. The first section examines the generated raw data and uses the results from "test" to verify the defined methodology and apparatus setup. The second section then analyzes the derived performance parameters and discusses the validity of using the SuperSPARC PIPE pins in performance analysis. Users should keep in mind that the data presented in this chapter are strictly tailored only to the system configuration described in Chapter 3 and Chapter 4.

5.1 Raw Data and Methodology Verification

Four executable programs are created for performance characterization. "test" is a very small program used merely to verify the validity of the implemented buffer. The other three programs are used to gather actual performance parameters – "matrix6" is a small scale program, "matrix200" is like "matrix6" but with a bigger dataset, and "dhry" is a large benchmark program. To gather statistic variables, the set of fourteen logic analyzer trigger programs were run for each of the four executed programs. Tables 5-1 to 5-4 list these trigger programs and their corresponding counter values. Notice that as the program size increases, the least significant digit of the counter values may be inconsistent. However, since this variation constitutes less than 0.001% of the accuracy, it is ignored.

From the raw data above, the desired performance parameters can be derived using

Logic Analyzer Program	Counter 2	Counter 1
0_instr_gp	95	50
1_instr_gp	95	30
2_instr_gp	95	11
3_instr_gp	95	4
BRop_exec	95	10
FPop_exec	95	5
MEMop_exec	95	25
BRop_takn	10	7
D$_miss_cyc	252	154
D$_miss_evnt	252	5
FP_intrlk_cyc	252	5
FP_intrlk_evnt	252	2
I$_miss_cyc	95	49
I$_miss_evnt	95	9

Table 5.1: Logic Analyzer Result – test

the equations described in Chapter 2. The parameters for each executable program are summarized in appendix E.

The data for "test" can be used to verify the validity of the hardware monitoring strategies. First of all, all stall free cycles must contain zero, one, two, or three instructions. The collected data on instruction grouping satisfy the following equation:

$$numCYCLE0 = num0InstrGroups + num1InstrGroup + num2InstrGroup + num3InstrGroup$$

$$95 = 50 + 30 + 11 + 4$$

The disassembled format of "test", "test.dis", further helps in verifying other statistics. The number of arithmetic logic operations, memory operations, branch operations, and floating point operations is verified by counting the executed instructions in "test.dis". These numbers all match the derived parameters. Floating point interlock statistics are also verified by identifying the two floating point code interlock events in the source code:

Logic Analyzer Program	Counter 2	Counter 1
0_instr_gp	1039	286
1_instr_gp	1039	319
2_instr_gp	1039	358
3_instr_gp	1039	76
BRop_exec	1039	81
FPop_exec	1039	180
MEMop_exec	1039	420
BRop_taken	81	55
D$_miss_cyc	1599	240
D$_miss_evnt	1599	10
FP_intrlk_cyc	1599	322
FP_intrlk_evnt	1599	47
I$_miss_cyc	1039	202
I$_miss_evnt	1039	32

Table 5.2: Logic Analyzer Result – matrix6

```
4070  _main+38      ldf     [%i6 - 8], %f0
!-- Pipeline stalls until ldf finishes with %f0
4074  _main+3c      fstod   %f0 , %f2
40d4  _cal_sum+1c      faddd   %f0 , %f0 , %f2
!-- Pipeline stalls until faddd finishes with %f0
40d8  _cal_sum+20      fmovs   %f2 , %f0
```

By comparing the C version, "test.c" and the disassembled version, "test.dis", the number of branch instructions and the number of taken branches can be counted, and statistics can be verified. The commented lines below indicate where the seven taken branch operations and three untaken branch operations take place:

```
⋮
  while(count < 3)
    count++;
  sum = cal_sum(sum);
⋮

4054  _main+1c      bge     4070  _main+38
! 3 untaken BRop when count = 0, 1, 2
! 1 taken BRop when count = 3
4058  _main+20      sethi   0, %g0
405c  _main+24      ld      [%i6 - 4], %o1
4060  _main+28      add     %o1, 1, %o1
4064  _main+2c      st      %o1, [%i6 - 4]
4068  _main+30      ba      404c  _main+14
! 3 taken BRop for branch always
⋮
4088  _main+50      call    40b8  _cal_sum
! 1 taken branch
```

Logic Analyzer Program	Counter 2	Counter 1
0_instr_gp	928,235	80,580
1_instr_gp	928,230	324,051
2_instr_gp	928,245	483,205
3_instr_gp	928,236	40,399
BRop_exec	928,230	81,409
FPop_exec	928,236	161,200
MEMop_exec	928,230	605,015
BRop_taken	81,409	40,809
D$_miss_cyc	1,904,282	693,568
D$_miss_evnt	1,904,275	79,677
FP_intrlk_cyc	1,904,284	282,784
FP_intrlk_evnt	1,904,280	40,399
I$_miss_cyc	928,230	175
I$_miss_evnt	928,236	38

Table 5.3: Logic Analyzer Result – matrix200

```
:
40e0  _cal_sum+28       ba      40e8  _cal_sum+30
! 1 taken branch
:
40e8  _cal_sum+30       jmp     %i7, 8, %g0
:
```

In contrast to the statistics above, data cache misses and instruction cache misses cannot be easily verified. They will be examined in the next section.

5.2 Statistic Analysis

Since "test" is a very small program used to verify hardware monitor strategies and is not representable of actual user code, the captured data are not included in this section for statistical analysis. This section examines the statistics derived from "matrix6", "matrix200", and "dhry".

5.2.1 Data Cache Miss

The data cache miss statistics for the three programs indicated a data cache miss rate range of 2.38% to 49% – 2.38% for "matrix6", 13% for "matrix200", and 49% for "dhry". This pattern reflects the fact that as the size of the program's dataset increases, the miss rate

Logic Analyzer Program	Counter 2	Counter 1
0_instr_gp	1,293,000,944	154,000,898
1_instr_gp	1,293,000,950	719,000,076
2_instr_gp	1,293,000,946	336,999,970
3_instr_gp	1,293,000,945	83,000,000
BRop_exec	1,293,000,945	229,000,001
FPop_exec	1,293,000,945	0
MEMop_exec	1,293,000,945	740,000,005
BRop_taken	229,000,001	152,999,999
D$_miss_cyc	7,124,729,181	5,831,728,245
D$_miss_evnt	7,124,729,181	363,000,010
FP_intrlk_cyc	7,124,729,181	0
FP_intrlk_evnt	7,124,729,181	0
I$_miss_cyc	1,293,000,940	40,000,896
I$_miss_evnt	1,293,000,946	20,000,139

Table 5.4: Logic Analyzer Result – dhry

increases. This phenomenon is expected since as the dataset becomes large, most data will not reside in the on-chip cache, and a significant number of data references to the next higher level memory (in this case, the main memory) is required. The data cache miss cycle rate also goes up as the size of the program increases – from 0.15 to 0.36 to 0.82 for "matrix6", "matrix200", and "dhry", respectively. This pattern matches the pattern of the data cache miss rate.

When users gather data cache statistics for a system configuration, care must be taken while determining the data cache miss rate. Several secondary reasons exist which may also cause a data memory reference. Examples of such reasons are: when the store buffer is full, when there is a synchronous store instruction, and when there is an atomic read-modify-write instruction. All these activities cause data memory reference, causing PIPE[5] to assert. Thus the derived data cache miss rate (based on the assertion of PIPE[5]) may be higher than the actual data cache miss rate.

The average duration of data cache miss events shows a range of 8.7 cycles/event for "matrix200" to 24 cycles/event for "matrix6". This inconsistency is mainly due to two reasons: the lumped effect with instruction cache misses, and the effect of the SuperSPARC data cache block replacement strategy.

Instruction cache prefetch nominally occurs independently of processor pipeline stalls. Both data and instruction cache misses require a memory reference via the external bus. In the above scenario, "matrix6" completes faster due to a smaller dataset. Thus, instruction cache miss constitutes a larger portion of the total misses and therefore contends for a larger portion of the bus cycles. Some of these instruction reference bus cycles occur while the processor pipelined is stalled. They are thus miscounted as data reference cycles, leading to a higher data cache miss event duration.

When a miss event occurs, the pipeline stalls and the SuperSPARC issues a 32-byte coherent read on the MBUS. This is a long process and may take many cycles to complete. However, once the first 8-byte datum is available, the pipeline proceeds again, and then stalls to wait for the next 8-byte datum. Thus, this single miss event might appear as multiple miss events, leading to a lower data cache miss duration. This effect is particularly apparent when the data cache miss rate is high (i.e. "matrix200").

Taking into account all the lumped effects requires some care; the individual effects cannot be easily singled out with moderate effort. Users could, however, monitor the additional 64-bit MBUS address/data lines as well as the MBUS control signals on a cycle by cycle basis to manifest the effect of the overlaped instruction miss cycles and the stalled cycles.

5.2.2 Floating Point Code Interlock

The statistics for floating point code interlock are more as expected. It shows a floating point interlock event number of zero for "dhry" (which has no floating point instructions), 47 events for "matrix6", and 40,399 events for "matrix200". The number of interlock events for the two matrix programs can easily be verified by counting the number of floating point code interlocks inside and outside of the program loops. The average duration of a floating point operation is between 6.85 cycles/event for "matrix6" to 7.00 cycles/event for "matrix200". The consistency of these statistics verifies the validity of the described methodology in generating floating point interlock parameters.

5.2.3 Instruction Cache Miss

As the size of the program increases, more memory references are expected; thus, the number of instruction cache miss events should increase [8]. The instruction cache miss statistics reflect this phenomenon with 32 miss events for "matrix6", 38 miss events for "matrix200", and 20,000,139 events for "dhry". Note that although the number of executed operations in "matrix200" exceeds those in "matrix6" by a large amount, the number of instruction miss events should not differ greatly, since the program sizes are almost identical.

The instruction cache miss rate equals the number of instruction miss events divided by the total number of instructions. The derived instruction cache miss rates support the above assumption. "matrix6" has an instruction cache miss rate of 2.53%, "matrix200" has a miss rate of 2.69e-3%, and "dhry" has a miss rate of 1.22%. Note that the instruction cache miss rate is particularly high for "matrix6" due to the initial program compulsory miss. Such compulsory miss effect is smoothed out when the number of executed instructions increases.

The statistics on the number of instruction cache miss cycles, and thus, the average durations of instruction miss events are inconsistent due to reasons similar to those presented for the data cache miss phenomenon. Instruction cache miss duration ranges from 2.00 cycles/event for "dhry" to 5.44 cycles/event for "matrix6". An instruction cache miss can be hidden while the processor pipeline is stalled. When the processor pipeline advances, the instruction fetch may have either completely or partially completed. In general, this circumstance increases the cycle count for the data cache miss and decreases the cycle count for the instruction cache miss. This effect is apparent on programs with a high portion of instruction cache misses (i.e. "matrix6").

5.2.4 Instruction Groups

The statistics on the following parameters can be derived accurately by using the SuperSPARC PIPE signals:

- $num0InstrGroup$
- $num1InstrGroup$
- $num2InstrGroup$

- $num3InstrGroup$
- $0InstrGroupFraction$
- $1InstrGroupFraction$
- $2InstrGroupFraction$
- $3InstrGroupFraction$
- $numProgramInstructions$
- $numMEMop$
- $numFPop$
- $numBRop$
- $numALUop$
- $numTakenBRop$
- $numUntakenBRop$
- $fractionTakenBRop$
- $fractionUntakenBRop$

These parameters can be verified by estimating the instruction count of the executed program in its assembly format. The parameters on the instruction grouping can also be verified using the following equation:

$$numCYCLE0 = num0InstrGroup + num1InstrGroup$$
$$+num2InstrGroup + num3InstrGroup \quad (5.1)$$
$$(5.2)$$

The statistics collected on the variables can be very useful for both static and dynamic analysis of user system behavior.

5.2.5 IPC and CPI

System performance is sometimes expressed in terms of instructions per cycle (IPC) or cycles per instruction (CPI). Both IPC and CPI are generated accurately for the executable programs.

We expect that as the size of a program increases, the data cache miss rate and/or the instruction cache miss rate will increase, and thus degrade system performance. The derived statistics confirm this assumption. The IPC value ranges from 0.79 instructions/cycle for "matrix6", to 0.74 instructions/cycle for "matrix200", to 0.23 instructions/cycle for "dhry". The CPI values, which are the reciprocal of the IPC values, range from 1.27 cycles/instruction for "matrix6", 1.35 cycles/instruction for "matrix200", to 4.34 cycles/instruction for "dhry".

It is important to note that since all parameters generated here are based on the SuperSPARC stand-alone configuration, the parameters generated with a multicache configuration will show improved performance.

Chapter 6

Conclusion

This thesis has documented the design and implementation of a hardware buffer and software environment which allow the performance analysis of SuperSPARC systems. The defined methodologies and apparatus setups were thoroughly tested, and the validity was determined.

An unconventional, yet extremely efficient and comprehensive method of performance analysis was discovered. The utilization of the SuperSPARC PIPE pins has led to a set of thirty-five system performance parameters. These parameters can be generated using the implemented hardware monitor buffer. This hardware buffer is modular and portable, permitting system performance analysis to be done on different system workplaces and/or different system workloads without making changes to the defined methodology and the apparatus setups. This setup provides an extremely efficient and effective setting for performance analysis.

In this thesis project, the system workplace was constructed using the SuperSPARC microprocessor standalone configuration interfacing to the Nimbus NIM6000M system mother board. All performance parameter data collected are thus particular to this system workplace only. The system workload for this project consisted of a set of four variable-length programs.

The derived data were analyzed to verify the validity of using the SuperSPARC PIPE pins in system performance analysis. The result shows that all but a few performance parameters can be generated accurately with the defined methodology. Some sophisticated

processor interactions, however, have prevented the capturing of accurate data cache and instruction cache statistics.

The scope of this thesis project leaves room for future research in this area. Users may wish to further investigate the causes of the deficiency in data cache and instruction cache statistics. The difficulties presented in this thesis prompt users to implement additional hardware and/or software to generate accurate data cache statistics. Users may also wish to expand the set of thirty-five parameters to capture more analysis statistics (e.g. the frequency and latency between context switches). Since the setup allows the monitoring of processor state on a cycle-by-cycle basis, users may apply this setup for system debugging, reconfiguration and improvement. The derived statistical data may also be useful in the field of compiler optimization and future design improvements.

Appendix A

Logic Analyzer Configuration File

```
 3 Sep 1993 15:29                                        DAS 9200 92A96SD-1 Setup
                                                                          Page 1

CHANNEL
-------

                                                        General Purpose Support

  Group Name   Input                          Probe          Channels
               Radix                                        MSB     LSB
-------------------------------------------------------------------------------

   Address0     Hex          Section  A1        Ch  76543210
                             Section  A0        Ch  76543210

   Address1     Hex          Section  A3        Ch  76543210
                             Section  A2        Ch  76543210

   Address2     Hex          Section  D0        Ch  3210

   Data         Bin          Section  D3        Ch  10
                             Section  D2        Ch  76543210

   PIPEMisc     Off          Section  C0        Ch  0

   MBUSMisc     Bin          Section  C1        Ch  210
```

Figure A-1: Logic Analyzer Channel Setup File

```
 3 Sep 1993 15:30                                        DAS 9200 92A96SD-1 Setup
                                                                          Page 1
CLOCK
-----

                                                        General Purpose Support
Module 92A96 Clock:  External

Sample Clock:
  External_Clocks              External_Qualifiers
 /    Clock_0      AND((                  AND                ) AND              )
```

Figure A-2: Logic Analyzer Clock Setup File

```
 3 Sep 1993 15:29                                        DAS 9200 92A96SD-1 Setup
                                                                          Page 1

CONFIG
------

                                                    General Purpose Support

Module Name:    92A96SD-1
Module Type:    92A96 - 96 channels at 100 Mhz

Software Support:          General Purpose

Acquisition Memory:        131072     Cycles

Latch Mode:                Off


Module Input Signals:                 Module Output Signals:

     None Defined                           Sync Out (Local)


Software Support
    General Purpose:     96 channels/card sync and async to 10.0 ns.
    High Speed Timing:   48 channels/card          async to  5.0 ns and
                         24 channels/card          async at  2.5 ns.
```

Figure A-3: Logic Analyzer Configuration File

```
1 Sep 1993 23:03                                    DAS 9200 92A96SD-1 Setup
                                                                    Page 1

 92A96 SETUP PRINT

 Saved Printer Settings

 Send Output To:    File                      File Name:    MEMop
    Output Format:           PostScript
    Characters per Line:      80
    Lines per Page:          60
    Spaces to Indent:         0


 Output Specification

 Comment in Heading:
 Print Overlays:         No
```

Figure A-4: Logic Analyzer Printer Environment File

Appendix B

Logic Analyzer Trigger Programs

0_instr_gp:

```
 3 Sep 1993 14:58                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                          General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Trigger           and store
               Go To State       0_instr_group
End of State Ck_begin
State    0_instr_group
   If          Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00X00X
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
End of State 0_instr_group
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

1_instr_gp:

```
 3 Sep 1993 15:12                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       1_instr_group
               Trigger          and store
End of State Ck_begin
State    1_instr_group
   If          Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00X01X
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
End of State 1_instr_group
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

2_instr_gp:

```
 3 Sep 1993 15:12                                        DAS 9200 92A96SD-1 Setup
                                                                          Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       2_instr_group
               Trigger          and store
End of State Ck_begin
State    2_instr_group
   If          Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00X10X
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
End of State 2_instr_group
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

3_instr_gp:

```
 3 Sep 1993 15:13                                        DAS 9200 92A96SD-1 Setup
                                                                          Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
--------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       3_instr_group
               Trigger           and store
End of State Ck_begin
State    3_instr_group
   If          Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00X11X
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
End of State 3_instr_group
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

BRop_exec:

```
 3 Sep 1993 15:14                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
-------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       BRop_exec0
               Trigger          and store
End of State Ck_begin
State    BRop_exec0
   If          Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XX1X00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State BRop_exec0
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

BRop_taken:

```
 3 Sep 1993 15:14                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       branch_taken0
               Trigger           and store
End of State Ck_begin
State    branch_taken0
   If          Word              #5    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       end_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XX1X00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX001XXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State branch_taken0
State    end_prog
   If          Anything
   Then        Do Nothing
End of State end_prog
```

D_miss_cyc:

```
 3 Sep 1993 15:15                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                          General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
-------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       Data$_miss_cy
               Trigger          and store
End of State Ck_begin
State    Data$_miss_cy
   If          Anything
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX1XXXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State Data$_miss_cy
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

D_miss_envt:

```
 3 Sep 1993 15:16                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
-------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       D$_miss_evnt0
               Trigger          and store
End of State Ck_begin
State    D$_miss_evnt0
   If          Anything
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX0XXXXX
                                  MBUSMisc  XXX
   Then        Go To State       D$_miss_evnt1
End of State D$_miss_evnt0
State    D$_miss_evnt1
   If          Anything
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX1XXXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
               Go To State       D$_miss_evnt0
End of State D$_miss_evnt1
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

FPintrlk_cy:

```
 3 Sep 1993 15:16                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
--------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       FP_intrlk_cyc
               Trigger           and store
End of State Ck_begin
State    FP_intrlk_cyc
   If          Anything
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXXX1XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State FP_intrlk_cyc
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

FPintrlk_ev:

```
 3 Sep 1993 15:17                                          DAS 9200 92A96SD-1 Setup
                                                                          Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
--------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       FP_intrlk_ev0
               Trigger          and store
End of State Ck_begin
State    FP_intrlk_ev0
   If          Anything
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXXX0XXXX
                                  MBUSMisc  XXX
   Then        Go To State       FP_intrlk_ev1
End of State FP_intrlk_ev0
State    FP_intrlk_ev1
   If          Anything
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXXX1XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
               Go To State       FP_intrlk_ev0
End of State FP_intrlk_ev1
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

FPop_exec:

```
 3 Sep 1993 15:17                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                          General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       FPop_exec0
               Trigger          and store
End of State Ck_begin
State    FPop_exec0
   If          Word              #3    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #4    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      X1XX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State FPop_exec0
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

I_miss_cyc:

```
 3 Sep 1993 15:18                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                          General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       I$_miss_cyc0
               Trigger           and store
End of State Ck_begin
State    I$_miss_cyc0
   If          Word              #4    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       end_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXX100XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State I$_miss_cyc0
State    end_prog
   If          Anything
   Then        Do Nothing
End of State end_prog
```

I_miss_evnt:

```
 3 Sep 1993 15:18                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                          General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
-------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       I$_miss_evnt0
               Trigger           and store
End of State Ck_begin
State    I$_miss_evnt0
   If          Word              #4    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       end_prog
   Or If       Word              #5    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXX0XXXXXX
                                  MBUSMisc  XXX
   Then        Go To State       I$_miss_envt1
End of State I$_miss_evnt0
State    I$_miss_envt1
   If          Word              #4    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       end_prog
   Or If       Word              #5    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXX100XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
               Go To State       I$_miss_evnt0
End of State I$_miss_envt1
State    end_prog
   If          Anything
   Then        Do Nothing
End of State end_prog
```

MEMop_exec:

```
 3 Sep 1993 15:19                                          DAS 9200 92A96SD-1 Setup
                                                                            Page 1
TRIGGER
                                                        General Purpose Support
Trigger Pos:   T--------
Store:   All Cycles
Prompt Visibility:   On
-------------------------------------------------------------------------------
State    Ck_begin
   If          Word              #1    =
                                  Address0  3000
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       MEMop_exec0
               Trigger           and store
End of State Ck_begin
State    MEMop_exec0
   If          Word              #4    =
                                  Address0  3100
                                  Address1  0000
                                  Address2  0
                                  Data      XXXXXXXXXX
                                  MBUSMisc  XX1
   Then        Go To State       End_prog
   Or If       Word              #2    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      XXXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #1
   Or If       Word              #3    =
                                  Address0  XXXX
                                  Address1  XXXX
                                  Address2  X
                                  Data      1XXX00XXXX
                                  MBUSMisc  XXX
   Then        Incr Counter      #2
End of State MEMop_exec0
State    End_prog
   If          Anything
   Then        Do Nothing
End of State End_prog
```

Appendix C

System Workload Source Code

"test" code:

Makefile:

```
TESTFLAGS = -Dss10
ASFLAGS = -P -I../include -I../include2 $(TESTFLAGS)
CC = cc -sun4
OBJECTS = $(SSOURCES:.s=.o) $(CSOURCES:.c=.o)
.c.o:
cc -c -R -O1 -I../include -I../include2 $(TESTFLAGS) $< -o $@
all: test
SMALLOBJECTS = test.o
test: $(SMALLOBJECTS)
/bin/ld -N -T 4000  -e __entry -o test $(SMALLOBJECTS) ../lib/libsmall.a
```

C Version:

```
#include<stdio.h>
float cal_sum();
main()
{
  int count = 0;
  float sum = 1.5;
  while(count < 3)
    count++;
  sum = cal_sum(sum);
}
float cal_sum(the_sum)
     float the_sum;
{
  return (the_sum * 2);
}
```

SPARC Assembly Version (editted):

```
#include <machine/asm_linkage.h>
#include <machine/mmu.h>
LL0:
.seg "data"
_romp:  .word   0
.seg "text"
.proc 04
.global __entry
__entry:
        set     0x2000, %o2
        mov     %o2, %sp
        set     _romp, %o1
        st      %o0, [%o1]
! set 0x10000000, %g1
! sta %g0, [%g0]0x36      ! I$ Flash Clear
!  sta %g0, [%g1]0x36
! sta     %g0, [%g0]0x37 ! D$ Flash Clear
! sta     %g0, [%g1]0x37
lda [%g0]0x04, %g4
set 0x00000700, %g3
or %g4, %g3, %g4
sta %g4, [%g0]0x04 ! Write to MCNTL
! set     0x4000, %o0 ! Set Page Map
! set     0x4ee,  %o1
! call    _SetPageMap, 2
! nop
        lda     [%g0]0x04, %g4
set 0x4200, %g1
        lda     [%g1]0x06, %g5
        ldda    [%g0] 0x4c, %g2
        or      %g3, -1, %g3
        stda    %g2,[%g0]0x4c          ! Write to action register
__trigger0:
        set     0x00003000, %g1
        stda    %g0, [%g1] 0x20
_main:
!#PROLOGUE# 0
! sethi %hi(LF27),%g1
! add %g1,%lo(LF27),%g1
! save %sp,%g1,%sp
        mov     %sp, %fp
!#PROLOGUE# 1
st %g0,[%fp+-0x4]
sethi %hi(L2000000),%o0
ld [%o0+%lo(L2000000)],%f0
st %f0,[%fp+-0x8]
L29:
ld [%fp+-0x4],%o0
cmp %o0,0x3
bge L30
nop
ld [%fp+-0x4],%o1
add %o1,0x1,%o1
st %o1,[%fp+-0x4]
b L29
nop
L30:
ld [%fp+-0x8],%f0
fstod %f0,%f2
st %f2,[%sp+LP27]
ld [%sp+LP27],%o0
```

```
st %f3,[%sp+LP27]
ld [%sp+LP27],%o1
call _cal_sum,2
nop
fdtos %f0,%f3
st %f3,[%fp+-0x8]
LE27:
! ret
! restore
__trigger1:
        set     0x00003100, %g1
        stda    %g0, [%g1] 0x20
__exit:
        sethi   %hi(_romp),%o0
        ld      [%o0+%lo(_romp)],%o0
        ld      [%o0+0x74],%g1
        call    %g1,0
        nop
        LF27 = -112
LP27 = 96
LST27 = 104
LT27 = 104
.seg "data"
.align 4
L2000000: .word 0x3fc00000
.seg "data"
.seg "text"
.proc 07
.global _cal_sum
_cal_sum:
!#PROLOGUE# 0
sethi %hi(LF31),%g1
add %g1,%lo(LF31),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %i1,[%fp+0x48]
ld2 [%fp+0x44],%f0
faddd %f0,%f0,%f2
fmovs %f2,%f0
fmovs %f3,%f1
b LE31
nop
LE31:
ret
restore
        LF31 = -64
LP31 = 64
LST31 = 64
LT31 = 64
.seg "data"
/*
 * Set the page map entry for the virtual address in the first arg (%o0)
 * to the entry specified in the second arg (%o1)
 * Called as SetPageMap(v_addr, pme)
 * ASI_MOD  0x4
 * ASI_MEM  0x20
 */
     ENTRY(SetPageMap)
     andn    %o0, 0x3, %o2          ! align on word
     set     RMMU_CTP_REG, %o5
     lda     [%o5]ASI_MOD, %o0          ! get context table pointer
     set     0xffffc000, %o5
     and     %o0, %o5, %o3
     sll     %o3, 0x4, %o3
     set     RMMU_CTX_REG, %o5
     lda     [%o5]ASI_MOD, %o4          ! get context register
     set     0xffffffc00, %o5
```

```
    and     %o4, %o5, %o5
    sll     %o5, 2, %o5
    or      %o5, %o3, %o3
    srl     %o0, 0x8, %o0
    srl     %o4, 0xa, %o4
    or      %o4, %o0, %o0
    and     %o0, 0x3f, %o0
    sll     %o0, 0xc, %o0
    or      %o0, %o3, %o3
    lda     [%o3]ASI_MEM, %o0         ! get region pointer
    andn    %o0, 0x1, %o5
    sll     %o5, 0x4, %o3
    set     0xff000000, %o0           ! mask for level 1 offset in vaddr
    and     %o2, %o0, %o0
    srl     %o0, 0x16, %o0
    add     %o0, %o3, %o3             ! get l2 pointer offset in l1 table
    lda     [%o3]ASI_MEM, %o0         ! get segment pointer
    andn    %o0, 0x1, %o5
    sll     %o5, 0x4, %o3
    set     0x00fc0000, %o0           ! mask for level 2 offset in vaddr
    and     %o2, %o0, %o0
    srl     %o0, 0x10, %o0
    add     %o0, %o3, %o3             ! get l3 pointer offset in l2 table
    lda     [%o3]ASI_MEM, %o0         ! get page table pointer
    andn    %o0, 0x1, %o5
    sll     %o5, 0x4, %o3
    set     0x0003f000, %o0           ! mask for l3 offset in vaddr
    and     %o2, %o0, %o0
    srl     %o0, 0x0a, %o0
    add     %o0, %o3, %o3             ! get pte addr in l3 table
    sta     %o1, [%o3]ASI_MEM      ! store entry to physical address
    mov     %o7, %g6               ! save return address
    call    _OnMBus, 0
    nop
    cmp     %o0, 0
    bnz     leave
    nop
    call    _InvalidateEcacheLine, 1
    mov     %o3, %o0
leave:
                                      ! the program
    retl
    mov     %g6, %o7

/*
 * Will return 0 in %o0 if the processor is connected to VBus or
 * not zero if the processor is directly on MBus
 */
    ENTRY (OnMBus)
    set     RMMU_CTL_REG, %o5
    lda     [%o5]ASI_MOD, %o4         ! get mmu control reg
    retl
    and     %o4, CPU_VIK_MB, %o0
/*
 * Will invalidate the Ecache line for the address in %o0
 * Assumes MBus module 0. Will not check ANYTHING, just invalidates
 */
    ENTRY (InvalidateEcacheLine)
    set     0xfff00000, %o5
    andn    %o0, %o5, %o4
    set     0xff800000, %o5
    or      %o5, %o4, %o3
    andn    %o3, 0x7, %o3
    ldda    [%o3]ASI_MXCC, %o4
    set     0x2222, %o2
```

```
andn    %o5, %o2, %o5
retl
stda    %o4, [%o3]ASI_MXCC
```

"matrix6" and "matrix200" code:

Makefile:

```
TESTFLAGS = -Dss10
ASFLAGS = -P -I../include -I../include2 $(TESTFLAGS)
CC = cc -sun4
OBJECTS = $(SSOURCES:.s=.o) $(CSOURCES:.c=.o)
.c.o:
cc -c -R -O1 -I../include -I../include2 $(TESTFLAGS) $< -o $@
all: matrix
SMALLOBJECTS = matrix.o
matrix: $(SMALLOBJECTS)
/bin/ld -N -T 4000  -e __entry -o matrix $(SMALLOBJECTS) ../lib/libsmall.a
```

C Version:

```
/* This is a sample Matrix Multiplication program */
#include<stdio.h>
#define NCOL 6  /* Number of columns and rows for this particular matrix */
#define NROW 6
/* function prototypes */
void init_row();
void init_col();
void cal_sum();
void print_result();
  float sum[NROW][NCOL], row[NROW], column[NCOL];
main()
{
  int i;
/* sum is the 2-d array holding the result of multiplying the two 1-d */
/* arrays: row[NROW] and column[NCOL] */
  init_row(&row[0]);  /* initialize the row[] matrix */
  init_col(&column[0]);  /* initialize the column[] matrix */
  cal_sum(&sum[0][0], &row[0], &column[0]);  /* calculate the result of row[] x column[] */
}
/* initialize the row[] matrix */
void init_row(the_row)
     float *the_row;
{
  int i;
  float a;
  for(i = 0, a = 0 ; i < NROW ; i++, a++)
    {
      *(the_row + i) = (a / 2);
    }
}
/* initialize the column[] matrix */
void init_col(the_col)
     float *the_col;
{
  int i;
  for(i = 0 ; i < NCOL ; i++)
    {
      *(the_col + i) = (i * 2 - 5);
    }
}
/* calculate the result of row[] x column[] and stores it in sum[] */
void cal_sum(the_sum, the_row, the_col)
     float *the_sum, *the_row, *the_col;
{
  float temp;
  int i, j;
  for(j = 0 ; j < NROW ; j++)
    {
      for(i = 0 ; i < NCOL ; i++)
{
  temp = (*(the_row + j)) * (*(the_col + i));
  *(the_sum + j * NCOL + i) = temp;
}
    }
}
/*****************************************************************************
* cc matrix.c -o matrix
* NROW = NCOL = 6
* hagar% matrix
|-|
0.000000
 0.500000
 1.000000
 1.500000
```

```
 2.000000
 2.500000
 |-|
[ -5.000000 -3.000000 -1.000000 1.000000 3.000000 5.000000 ]
 -0.000000    -0.000000    -0.000000    0.000000    0.000000    0.000000
 -2.500000    -1.500000    -0.500000    0.500000    1.500000    2.500000
 -5.000000    -3.000000    -1.000000    1.000000    3.000000    5.000000
 -7.500000    -4.500000    -1.500000    1.500000    4.500000    7.500000
 -10.000000    -6.000000    -2.000000    2.000000    6.000000    10.000000
 -12.500000    -7.500000    -2.500000    2.500000    7.500000    12.500000
*****************************************************************************/
```

SPARC Assembly Version:

```
#include <machine/asm_linkage.h>
#include <machine/mmu.h>
LL0:
.seg "data"
.common _i,0x4,"data"
.common _sum,0x27100,"data"
.common _row,0x320,"data"
.common _column,0x320,"data"
_romp:  .word   0
.seg "text"
.proc 04
.global _main
        .global __entry
__entry:
        set     0x2000, %o2
        mov     %o2, %sp
        set     _romp, %o1
        st      %o0, [%o1]
        lda     [%g0]0x04, %g4
        set     0x00000700, %g3
        or      %g4, %g3, %g4
        sta     %g4, [%g0]0x04          ! Write to MCNTL
        lda     [%g0]0x04, %g4
        set     0x4200, %g1
        lda     [%g1]0x06, %g5
        ldda    [%g0] 0x4c, %g2
        or      %g3, -1, %g3
        stda    %g2,[%g0]0x4c           ! Write to action register
__trigger0:
        set     0x00003000, %g1
        stda    %g0, [%g1] 0x20
_main:
!#PROLOGUE# 0
! sethi %hi(LF34),%g1
! add %g1,%lo(LF34),%g1
! save %sp,%g1,%sp
        mov     %sp, %fp
!#PROLOGUE# 1
set _row,%o0
call _init_row,1
nop
set _column,%o0
call _init_col,1
nop
set _sum,%o0
set _row,%o1
set _column,%o2
call _cal_sum,3
nop
LE34:
! ret
! restore
__trigger1:
        set     0x00003100, %g1
        stda    %g0, [%g1] 0x20
__exit:
        sethi   %hi(_romp),%o0
        ld      [%o0+%lo(_romp)],%o0
        ld      [%o0+0x74],%g1
        call    %g1,0
        nop
      LF34 = -96
LP34 = 96
LST34 = 96
```

```
LT34 = 96
.seg "data"
.seg "text"
.proc 020
.global _init_row
_init_row:
!#PROLOGUE# 0
sethi %hi(LF36),%g1
add %g1,%lo(LF36),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %g0,[%fp+-0x4]
sethi %hi(L2000000),%o0
ld [%o0+%lo(L2000000)],%f0
st %f0,[%fp+-0x8]
L40:
ld [%fp+-0x4],%o1
cmp %o1,0xc8
bge L39
nop
sethi %hi(L2000001),%o0
ldd [%o0+%lo(L2000001)],%f0
ld [%fp+-0x8],%f2
fstod %f2,%f4
fdivd %f4,%f0,%f6
fdtos %f6,%f7
ld [%fp+-0x4],%o1
sll %o1,0x2,%o2
ld [%fp+0x44],%o3
st %f7,[%o3+%o2]
L38:
ld [%fp+-0x4],%o4
add %o4,0x1,%o4
st %o4,[%fp+-0x4]
sethi %hi(L2000002),%o5
ld [%o5+%lo(L2000002)],%f8
ld [%fp+-0x8],%f9
fadds %f9,%f8,%f9
st %f9,[%fp+-0x8]
b L40
nop
L39:
LE36:
ret
restore
      LF36 = -72
LP36 = 64
LST36 = 64
LT36 = 64
.seg "data"
.align 4
L2000000: .word 0x0
.align 8
L2000001: .word 0x40000000,0x0
.align 4
L2000002: .word 0x3f800000
.seg "data"
.seg "text"
.proc 020
.global _init_col
_init_col:
!#PROLOGUE# 0
sethi %hi(LF41),%g1
add %g1,%lo(LF41),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %g0,[%fp+-0x4]
```

```
L45:
ld [%fp+-0x4],%o0
cmp %o0,0xc8
bge L44
nop
ld [%fp+-0x4],%o0
sll %o0,0x1,%o1
sub %o1,0x5,%o2
st %o2,[%sp+LP41]
ld [%sp+LP41],%f0
fitos %f0,%f1
ld [%fp+-0x4],%o3
sll %o3,0x2,%o4
ld [%fp+0x44],%o5
st %f1,[%o5+%o4]
L43:
ld [%fp+-0x4],%o7
add %o7,0x1,%o7
st %o7,[%fp+-0x4]
b L45
nop
L44:
LE4i:
ret
restore
        LF41 = -80
LP41 = 64
LST41 = 72
LT41 = 72
.seg "data"
.seg "text"
.proc 020
.global _cal_sum
_cal_sum:
!#PROLOGUE# 0
sethi %hi(LF46),%g1
add %g1,%lo(LF46),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %i1,[%fp+0x48]
st %i2,[%fp+0x4c]
st %g0,[%fp+-0xc]
L50:
ld [%fp+-0xc],%o0
cmp %o0,0xc8
bge L49
nop
st %g0,[%fp+-0x8]
L53:
ld [%fp+-0x8],%o0
cmp %o0,0xc8
bge L52
nop
ld [%fp+-0x8],%o0
sll %o0,0x2,%o1
ld [%fp+0x4c],%o2
ld [%o2+%o1],%f0
fstod %f0,%f2
ld [%fp+-0xc],%o3
sll %o3,0x2,%o4
ld [%fp+0x48],%o5
ld [%o5+%o4],%f4
fstod %f4,%f6
fmuld %f6,%f2,%f8
fdtos %f8,%f9
st %f9,[%fp+-0x4]
```

```
ld [%fp+-0xc],%o7
sll %o7,5,%o7
mov %o7,%l0
sll %l0,3,%l0
add %o7,%l0,%o7
sll %l0,1,%l0
add %o7,%l0,%o7
ld [%fp+0x44],%l1
add %l1,%o7,%l2
ld [%fp+-0x8],%l3
sll %l3,0x2,%l4
ld [%fp+-0x4],%l5
st %l5,[%l2+%l4]
L51:
ld [%fp+-0x8],%l6
add %l6,0x1,%l6
st %l6,[%fp+-0x8]
b L53
nop
L52:
L48:
ld [%fp+-0xc],%l7
add %l7,0x1,%l7
st %l7,[%fp+-0xc]
b L50
nop
L49:
LE46:
ret
restore
        LF46 = -80
LP46 = 64
LST46 = 64
LT46 = 64
.seg "data"
```

"dhry" code:

Makefile:

```
TESTFLAGS = -Dss10
ASFLAGS = -P -I../include -I../include2 $(TESTFLAGS)
CC = cc -sun4
OBJECTS = $(SSOURCES:.s=.o) $(CSOURCES:.c=.o)
.c.o:
cc -c -R -O1 -I../include -I../include2 $(TESTFLAGS) $< -o $@
all: dhry
SMALLOBJECTS = dhry_1.o dhry_2.o multiply.o string.o
dhry: $(SMALLOBJECTS)
/bin/ld -N -T 4000  -e __entry -o dhry $(SMALLOBJECTS) ../lib/libsmall.a
```

C Version:

```
/*
 ****************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_1.c (part 2 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ****************************************************************************
 */
#include "dhry.h"
/* #include "/projects/numesh/sparc/include/fifo.h" */
/* Global Variables: */
Rec_Pointer     Ptr_Glob,
                Next_Ptr_Glob;
int             Int_Glob;
Boolean         Bool_Glob;
char            Ch_1_Glob,
                Ch_2_Glob;
int             Arr_1_Glob [50];
int             Arr_2_Glob [50] [50];
#ifdef MALLOC
extern char     *malloc ();
#endif
Enumeration     Func_1 ();
  /* forward declaration necessary since Enumeration may not simply be int */
#ifndef REG
        Boolean Reg = false;
#define REG
        /* REG becomes defined as empty */
        /* i.e. no register variables   */
#else
        Boolean Reg = true;
#endif
/* variables for time measurement: */
#ifdef TIMES
struct tms      time_info;
/*  don't define times() at all... int is default, and sometimes broken
    (as on sun os 4's, where it's clock_t */
/* extern  int     times ();
                /* see library function "times" */
#define Too_Small_Time (2*HZ)
                /* Measurements should last at least about 2 seconds */
#endif
#ifdef TIME
extern long     time();
                /* see library function "time"  */
#define Too_Small_Time 2
                /* Measurements should last at least 2 seconds */
#endif
#ifdef MSC_CLOCK
extern clock_t clock();
#define Too_Small_Time (2*HZ)
#endif
long            Begin_Time,
                End_Time,
                User_Time;
float           Microseconds,
                Dhrystones_Per_Second;
/* end of variables for time measurement */
        One_Fifty       Int_1_Loc;
  REG   One_Fifty       Int_2_Loc;
        One_Fifty       Int_3_Loc;
  REG   char            Ch_Index;
```

```
        Enumeration     Enum_Loc;
        Str_30          Str_1_Loc;
        Str_30          Str_2_Loc;
  REG   int             Run_Index;
  REG   int             Number_Of_Runs;
main ()
/*****/
  /* main program, corresponds to procedures        */
  /* Main and Proc_0 in the Ada version             */
{
  /* Initializations */
#ifdef MALLOC
  Next_Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
  Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
#else
  static Rec_Type _Next_Glob, _Glob;
  Next_Ptr_Glob = &_Next_Glob;
  Ptr_Glob = &_Glob;
#endif
  Ptr_Glob->Ptr_Comp                    = Next_Ptr_Glob;
  Ptr_Glob->Discr                       = Ident_1;
  Ptr_Glob->variant.var_1.Enum_Comp     = Ident_3;
  Ptr_Glob->variant.var_1.Int_Comp      = 40;
  strcpy (Ptr_Glob->variant.var_1.Str_Comp,
          "DHRYSTONE PROGRAM, SOME STRING");
  strcpy (Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");
  Arr_2_Glob [8][7] = 10;
        /* Was missing in published program. Without this statement,    */
        /* Arr_2_Glob [8][7] would have an undefined value.             */
        /* Warning: With 16-Bit processors and Number_Of_Runs > 32000,  */
        /* overflow may occur for this array element.                   */
/*
  printf ("\n");
  printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
  printf ("\n");
  if (Reg)
  {
    printf ("Program compiled with 'register' attribute\n");
    printf ("\n");
  }
  else
  {
    printf ("Program compiled without 'register' attribute\n");
    printf ("\n");
  }
*/
#ifndef NUMRUNS
#define NUMRUNS 1000000
#endif
#ifdef NUMRUNS
  Number_Of_Runs = NUMRUNS;
#else
  printf ("Please give the number of runs through the benchmark: ");
  {
    int n;
    scanf ("%d", &n);
    Number_Of_Runs = n;
  }
  printf ("\n");
#endif
/*  printf ("Execution starts, %d runs through Dhrystone\n", Number_Of_Runs);*/
/* write_fifo(Number_Of_Runs); */
  /***************/
  /* Start timer */
  /***************/
#ifdef TIMES
  times (&time_info);
```

```
  Begin_Time = (long) time_info.tms_utime;
#endif
#ifdef TIME
  Begin_Time = time ( (long *) 0);
#endif
#ifdef MSC_CLOCK
  Begin_Time = clock();
#endif
  for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
  {
    Proc_5();
    Proc_4();
      /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
    Int_1_Loc = 2;
    Int_2_Loc = 3;
    strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
    Enum_Loc = Ident_2;
    Bool_Glob = ! Func_2 (Str_1_Loc, Str_2_Loc);
      /* Bool_Glob == 1 */
    while (Int_1_Loc < Int_2_Loc)  /* loop body executed once */
    {
      Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
        /* Int_3_Loc == 7 */
      Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
        /* Int_3_Loc == 7 */
      Int_1_Loc += 1;
    } /* while */
      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Proc_8 (Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
      /* Int_Glob == 5 */
    Proc_1 (Ptr_Glob);
    for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
                             /* loop body executed twice */
    {
      if (Enum_Loc == Func_1 (Ch_Index, 'C'))
          /* then, not executed */
        {
        Proc_6 (Ident_1, &Enum_Loc);
        strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
        Int_2_Loc = Run_Index;
        Int_Glob = Run_Index;
        }
    }
      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Int_2_Loc = Int_2_Loc * Int_1_Loc;
    Int_1_Loc = Int_2_Loc / Int_3_Loc;
    Int_2_Loc = 7 * (Int_2_Loc - Int_3_Loc) - Int_1_Loc;
      /* Int_1_Loc == 1, Int_2_Loc == 13, Int_3_Loc == 7 */
    Proc_2 (&Int_1_Loc);
      /* Int_1_Loc == 5 */
  } /* loop "for Run_Index" */
  /**************/
  /* Stop timer */
  /**************/
#ifdef TIMES
  times (&time_info);
  End_Time = (long) time_info.tms_utime;
#endif
#ifdef TIME
  End_Time = time ( (long *) 0);
#endif
#ifdef MSC_CLOCK
  End_Time = clock();
#endif
/*
  printf ("Execution ends\n");
  printf ("\n");
```

```
  printf ("Final values of the variables used in the benchmark:\n");
  printf ("\n");
  printf ("Int_Glob:            %d\n", Int_Glob);
  printf ("        should be:   %d\n", 5);
  printf ("Bool_Glob:           %d\n", Bool_Glob);
  printf ("        should be:   %d\n", 1);
  printf ("Ch_1_Glob:           %c\n", Ch_1_Glob);
  printf ("        should be:   %c\n", 'A');
  printf ("Ch_2_Glob:           %c\n", Ch_2_Glob);
  printf ("        should be:   %c\n", 'B');
  printf ("Arr_1_Glob[8]:       %d\n", Arr_1_Glob[8]);
  printf ("        should be:   %d\n", 7);
  printf ("Arr_2_Glob[8][7]:    %d\n", Arr_2_Glob[8][7]);
  printf ("        should be:   Number_Of_Runs + 10\n");
  printf ("Ptr_Glob->\n");
  printf ("  Ptr_Comp:          %d\n", (int) Ptr_Glob->Ptr_Comp);
  printf ("        should be:   (implementation-dependent)\n");
  printf ("  Discr:             %d\n", Ptr_Glob->Discr);
  printf ("        should be:   %d\n", 0);
  printf ("  Enum_Comp:         %d\n", Ptr_Glob->variant.var_1.Enum_Comp);
  printf ("        should be:   %d\n", 2);
  printf ("  Int_Comp:          %d\n", Ptr_Glob->variant.var_1.Int_Comp);
  printf ("        should be:   %d\n", 17);
  printf ("  Str_Comp:          %s\n", Ptr_Glob->variant.var_1.Str_Comp);
  printf ("        should be:   DHRYSTONE PROGRAM, SOME STRING\n");
  printf ("Next_Ptr_Glob->\n");
  printf ("  Ptr_Comp:          %d\n", (int) Next_Ptr_Glob->Ptr_Comp);
  printf ("        should be:   (implementation-dependent), same as above\n");
  printf ("  Discr:             %d\n", Next_Ptr_Glob->Discr);
  printf ("        should be:   %d\n", 0);
  printf ("  Enum_Comp:         %d\n", Next_Ptr_Glob->variant.var_1.Enum_Comp);
  printf ("        should be:   %d\n", 1);
  printf ("  Int_Comp:          %d\n", Next_Ptr_Glob->variant.var_1.Int_Comp);
  printf ("        should be:   %d\n", 18);
  printf ("  Str_Comp:          %s\n",
                                Next_Ptr_Glob->variant.var_1.Str_Comp);
  printf ("        should be:   DHRYSTONE PROGRAM, SOME STRING\n");
  printf ("Int_1_Loc:           %d\n", Int_1_Loc);
  printf ("        should be:   %d\n", 5);
  printf ("Int_2_Loc:           %d\n", Int_2_Loc);
  printf ("        should be:   %d\n", 13);
  printf ("Int_3_Loc:           %d\n", Int_3_Loc);
  printf ("        should be:   %d\n", 7);
  printf ("Enum_Loc:            %d\n", Enum_Loc);
  printf ("        should be:   %d\n", 1);
  printf ("Str_1_Loc:           %s\n", Str_1_Loc);
  printf ("        should be:   DHRYSTONE PROGRAM, 1'ST STRING\n");
  printf ("Str_2_Loc:           %s\n", Str_2_Loc);
  printf ("        should be:   DHRYSTONE PROGRAM, 2'ND STRING\n");
  printf ("\n");
*/
/*  write_fifo(-1); */
  User_Time = End_Time - Begin_Time;
#if defined(TIME) || defined(TIMES) || defined(MSC_TIME)
  if (User_Time < Too_Small_Time)
  {
    printf ("Measured time too small to obtain meaningful results\n");
    printf ("Please increase number of runs\n");
    printf ("\n");
  }
  else
```

```
  {
#ifdef TIME
    Microseconds = (float) User_Time * Mic_secs_Per_Second
                        / (float) Number_Of_Runs;
    Dhrystones_Per_Second = (float) Number_Of_Runs / (float) User_Time;
#else
    Microseconds = (float) User_Time * Mic_secs_Per_Second
                        / ((float) HZ * ((float) Number_Of_Runs));
    Dhrystones_Per_Second = ((float) HZ * (float) Number_Of_Runs)
                        / (float) User_Time;
#endif
    printf ("Microseconds for one run through Dhrystone: ");
    printf ("%6.1f \n", Microseconds);
    printf ("Dhrystones per Second:                      ");
    printf ("%6.1f \n", Dhrystones_Per_Second);
    printf ("\n");
  }
#endif
/*    return(0);  */
}
Proc_1 (Ptr_Val_Par)
/******************/
REG Rec_Pointer Ptr_Val_Par;
    /* executed once */
{
  REG Rec_Pointer Next_Record = Ptr_Val_Par->Ptr_Comp;
                                        /* == Ptr_Glob_Next */
  /* Local variable, initialized with Ptr_Val_Par->Ptr_Comp,    */
  /* corresponds to "rename" in Ada, "with" in Pascal           */
  structassign (*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
  Ptr_Val_Par->variant.var_1.Int_Comp = 5;
  Next_Record->variant.var_1.Int_Comp
        = Ptr_Val_Par->variant.var_1.Int_Comp;
  Next_Record->Ptr_Comp = Ptr_Val_Par->Ptr_Comp;
  Proc_3 (&Next_Record->Ptr_Comp);
    /* Ptr_Val_Par->Ptr_Comp->Ptr_Comp
                        == Ptr_Glob->Ptr_Comp */
  if (Next_Record->Discr == Ident_1)
    /* then, executed */
  {
    Next_Record->variant.var_1.Int_Comp = 6;
    Proc_6 (Ptr_Val_Par->variant.var_1.Enum_Comp,
           &Next_Record->variant.var_1.Enum_Comp);
    Next_Record->Ptr_Comp = Ptr_Glob->Ptr_Comp;
    Proc_7 (Next_Record->variant.var_1.Int_Comp, 10,
           &Next_Record->variant.var_1.Int_Comp);
  }
  else /* not executed */
    structassign (*Ptr_Val_Par, *Ptr_Val_Par->Ptr_Comp);
} /* Proc_1 */
Proc_2 (Int_Par_Ref)
/******************/
    /* executed once */
    /* *Int_Par_Ref == 1, becomes 4 */
One_Fifty   *Int_Par_Ref;
{
  One_Fifty  Int_Loc;
  Enumeration   Enum_Loc;
  Int_Loc = *Int_Par_Ref + 10;
  do /* executed once */
    if (Ch_1_Glob == 'A')
      /* then, executed */
    {
      Int_Loc -= 1;
      *Int_Par_Ref = Int_Loc - Int_Glob;
```

```
      Enum_Loc = Ident_1;
    } /* if */
  while (Enum_Loc != Ident_1); /* true */
} /* Proc_2 */
Proc_3 (Ptr_Ref_Par)
/******************/
    /* executed once */
    /* Ptr_Ref_Par becomes Ptr_Glob */
Rec_Pointer *Ptr_Ref_Par;
{
  if (Ptr_Glob != Null)
    /* then, executed */
    *Ptr_Ref_Par = Ptr_Glob->Ptr_Comp;
  Proc_7 (10, Int_Glob, &Ptr_Glob->variant.var_1.Int_Comp);
} /* Proc_3 */
Proc_4 () /* without parameters */
/*******/
    /* executed once */
{
  Boolean Bool_Loc;
  Bool_Loc = Ch_1_Glob == 'A';
  Bool_Glob = Bool_Loc | Bool_Glob;
  Ch_2_Glob = 'B';
} /* Proc_4 */
Proc_5 () /* without parameters */
/*******/
    /* executed once */
{
  Ch_1_Glob = 'A';
  Bool_Glob = false;
} /* Proc_5 */
        /* Procedure for the assignment of structures,          */
        /* if the C compiler doesn't support this feature       */
#ifdef  NOSTRUCTASSIGN
memcpy (d, s, l)
register char   *d;
register char   *s;
register int    l;
{
        while (l--) *d++ = *s++;
}
#endif
/*
 ************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_2.c (part 3 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ************************************************************************
 */
#include "dhry.h"
#ifndef REG
#define REG
        /* REG becomes defined as empty */
        /* i.e. no register variables   */
#endif
extern  int     Int_Glob;
extern  char    Ch_1_Glob;
Proc_6 (Enum_Val_Par, Enum_Ref_Par)
/*********************************/
    /* executed once */
```

```
    /* Enum_Val_Par == Ident_3, Enum_Ref_Par becomes Ident_2 */
Enumeration  Enum_Val_Par;
Enumeration *Enum_Ref_Par;
{
  *Enum_Ref_Par = Enum_Val_Par;
  if (! Func_3 (Enum_Val_Par))
    /* then, not executed */
    *Enum_Ref_Par = Ident_4;
  switch (Enum_Val_Par)
  {
    case Ident_1:
      *Enum_Ref_Par = Ident_1;
      break;
    case Ident_2:
      if (Int_Glob > 100)
        /* then */
      *Enum_Ref_Par = Ident_1;
      else *Enum_Ref_Par = Ident_4;
      break;
    case Ident_3: /* executed */
      *Enum_Ref_Par = Ident_2;
      break;
    case Ident_4: break;
    case Ident_5:
      *Enum_Ref_Par = Ident_3;
      break;
  } /* switch */
} /* Proc_6 */
Proc_7 (Int_1_Par_Val, Int_2_Par_Val, Int_Par_Ref)
/*********************************************/
    /* executed three times                                      */
    /* first call:      Int_1_Par_Val == 2, Int_2_Par_Val == 3,  */
    /*                  Int_Par_Ref becomes 7                    */
    /* second call:     Int_1_Par_Val == 10, Int_2_Par_Val == 5, */
    /*                  Int_Par_Ref becomes 17                   */
    /* third call:      Int_1_Par_Val == 6, Int_2_Par_Val == 10, */
    /*                  Int_Par_Ref becomes 18                   */
One_Fifty       Int_1_Par_Val;
One_Fifty       Int_2_Par_Val;
One_Fifty      *Int_Par_Ref;
{
  One_Fifty Int_Loc;
  Int_Loc = Int_1_Par_Val + 2;
  *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
} /* Proc_7 */
Proc_8 (Arr_1_Par_Ref, Arr_2_Par_Ref, Int_1_Par_Val, Int_2_Par_Val)
/*********************************************************************/
    /* executed once      */
    /* Int_Par_Val_1 == 3 */
    /* Int_Par_Val_2 == 7 */
Arr_1_Dim       Arr_1_Par_Ref;
Arr_2_Dim       Arr_2_Par_Ref;
int             Int_1_Par_Val;
int             Int_2_Par_Val;
{
  REG One_Fifty Int_Index;
  REG One_Fifty Int_Loc;
  Int_Loc = Int_1_Par_Val + 5;
  Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
  Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
  Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
  for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)
    Arr_2_Par_Ref [Int_Loc] [Int_Index] = Int_Loc;
  Arr_2_Par_Ref [Int_Loc] [Int_Loc-1] += 1;
  Arr_2_Par_Ref [Int_Loc+20] [Int_Loc] = Arr_1_Par_Ref [Int_Loc];
  Int_Glob = 5;
} /* Proc_8 */
Enumeration Func_1 (Ch_1_Par_Val, Ch_2_Par_Val)
```

```
/*************************************************/
    /* executed three times                                         */
    /* first call:      Ch_1_Par_Val == 'H', Ch_2_Par_Val == 'R'    */
    /* second call:     Ch_1_Par_Val == 'A', Ch_2_Par_Val == 'C'    */
    /* third call:      Ch_1_Par_Val == 'B', Ch_2_Par_Val == 'C'    */
Capital_Letter   Ch_1_Par_Val;
Capital_Letter   Ch_2_Par_Val;
{
  Capital_Letter        Ch_1_Loc;
  Capital_Letter        Ch_2_Loc;
  Ch_1_Loc = Ch_1_Par_Val;
  Ch_2_Loc = Ch_1_Loc;
  if (Ch_2_Loc != Ch_2_Par_Val)
    /* then, executed */
    return (Ident_1);
  else  /* not executed */
  {
    Ch_1_Glob = Ch_1_Loc;
    return (Ident_2);
   }
} /* Func_1 */
Boolean Func_2 (Str_1_Par_Ref, Str_2_Par_Ref)
/*************************************************/
    /* executed once */
    /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
    /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */
Str_30  Str_1_Par_Ref;
Str_30  Str_2_Par_Ref;
{
  REG One_Thirty        Int_Loc;
      Capital_Letter    Ch_Loc;
  Int_Loc = 2;
  while (Int_Loc <= 2) /* loop body executed once */
    if (Func_1 (Str_1_Par_Ref[Int_Loc],
                Str_2_Par_Ref[Int_Loc+1]) == Ident_1)
      /* then, executed */
    {
      Ch_Loc = 'A';
      Int_Loc += 1;
    } /* if, while */
  if (Ch_Loc >= 'W' && Ch_Loc < 'Z')
    /* then, not executed */
    Int_Loc = 7;
  if (Ch_Loc == 'R')
    /* then, not executed */
    return (true);
  else /* executed */
  {
    if (strcmp (Str_1_Par_Ref, Str_2_Par_Ref) > 0)
      /* then, not executed */
    {
      Int_Loc += 7;
      Int_Glob = Int_Loc;
      return (true);
    }
    else /* executed */
      return (false);
  } /* if Ch_Loc */
} /* Func_2 */
Boolean Func_3 (Enum_Par_Val)
/***************************/
    /* executed once        */
    /* Enum_Par_Val == Ident_3 */
Enumeration Enum_Par_Val;
{
  Enumeration Enum_Loc;
  Enum_Loc = Enum_Par_Val;
```

```
  if (Enum_Loc == Ident_3)
    /* then, executed */
    return (true);
  else /* not executed */
    return (false);
} /* Func_3 */
```

SPARC Assembly Version (editted):

```
#include <machine/asm_linkage.h>
#include <machine/mmu.h>
LL0:
.seg "data"
_romp:  .word   0
.common _Ptr_Glob,0x4,"data"
.common _Next_Ptr_Glob,0x4,"data"
.common _Int_Glob,0x4,"data"
.common _Bool_Glob,0x4,"data"
.common _Ch_1_Glob,0x1,"data"
.common _Ch_2_Glob,0x1,"data"
.common _Arr_1_Glob,0xc8,"data"
.common _Arr_2_Glob,0x2710,"data"
.align 4
.global _Reg
_Reg:
.word 0x0
.common _Begin_Time,0x4,"data"
.common _End_Time,0x4,"data"
.common _User_Time,0x4,"data"
.common _Microseconds,0x4,"data"
.common _Dhrystones_Per_Second,0x4,"data"
.common _Int_1_Loc,0x4,"data"
.common _Int_2_Loc,0x4,"data"
.common _Int_3_Loc,0x4,"data"
.common _Ch_Index,0x1,"data"
.common _Enum_Loc,0x4,"data"
.common _Str_1_Loc,0x1f,"data"
.common _Str_2_Loc,0x1f,"data"
.common _Run_Index,0x4,"data"
.common _Number_Of_Runs,0x4,"data"
.seg "text"
.proc 04
.global _main
        .global __entry
__entry:
        set     0x20000, %o2
        mov     %o2, %sp
        set     _romp, %o1
        st      %o0, [%o1]
        lda     [%g0]0x04, %g4
        set     0x00000700, %g3
        or      %g4, %g3, %g4
        sta     %g4, [%g0]0x04          ! Write to MCNTL
        lda     [%g0]0x04, %g4
        set     0x4200, %g1
        lda     [%g1]0x06, %g5
        ldda    [%g0] 0x4c, %g2
        or      %g3, -1, %g3
        stda    %g2,[%g0]0x4c           ! Write to action register
_main:
!#PROLOGUE# 0
! sethi %hi(LF50),%g1
! add %g1,%lo(LF50),%g1
! save %sp,%g1,%sp
        mov     %sp, %fp
!#PROLOGUE# 1
.seg "bss"
.align 4
L52: .skip 48
.seg "text"
.seg "bss"
.align 4
L53: .skip 48
```

```
.seg "text"
set L52,%o0
sethi %hi(_Next_Ptr_Glob),%o1
st %o0,[%o1+%lo(_Next_Ptr_Glob)]
set L53,%o2
sethi %hi(_Ptr_Glob),%o3
st %o2,[%o3+%lo(_Ptr_Glob)]
sethi %hi(_Ptr_Glob),%o4
ld [%o4+%lo(_Ptr_Glob)],%o4
sethi %hi(_Next_Ptr_Glob),%o5
ld [%o5+%lo(_Next_Ptr_Glob)],%o5
st %o5,[%o4]
sethi %hi(_Ptr_Glob),%o7
ld [%o7+%lo(_Ptr_Glob)],%o7
st %g0,[%o7+0x4]
sethi %hi(_Ptr_Glob),%l0
ld [%l0+%lo(_Ptr_Glob)],%l0
mov 0x2,%l1
st %l1,[%l0+0x8]
sethi %hi(_Ptr_Glob),%l2
ld [%l2+%lo(_Ptr_Glob)],%l2
mov 0x28,%l3
st %l3,[%l2+0xc]
.seg "data1"
L55:
.ascii "DHRYSTONE PROGRAM, SOME STRING\0"
.seg "text"
sethi %hi(_Ptr_Glob),%o0
ld [%o0+%lo(_Ptr_Glob)],%o0
add %o0,0x10,%o0
set L55,%o1
call _strcpy,2
nop
.seg "data1"
L56:
.ascii "DHRYSTONE PROGRAM, 1'ST STRING\0"
.seg "text"
set _Str_1_Loc,%o0
set L56,%o1
call _strcpy,2
nop
mov 0xa,%l4
sethi %hi(_Arr_2_Glob+0x65c),%l5
st %l4,[%l5+%lo(_Arr_2_Glob+0x65c)]
__trigger0:
        set     0x00003000, %g1
        stda    %g0, [%g1] 0x20
set 0xf4240,%l6
sethi %hi(_Number_Of_Runs),%l7
st %l6,[%l7+%lo(_Number_Of_Runs)]
mov 0x1,%i0
sethi %hi(_Run_Index),%i1
st %i0,[%i1+%lo(_Run_Index)]
sethi %hi(_Run_Index),%i2
ld [%i2+%lo(_Run_Index)],%i2
sethi %hi(_Number_Of_Runs),%i3
ld [%i3+%lo(_Number_Of_Runs)],%i3
cmp %i2,%i3
bg L58
nop
L59:
call _Proc_5,0
nop
call _Proc_4,0
nop
mov 0x2,%o0
```

```
sethi %hi(_Int_1_Loc),%o1
st %o0,[%o1+%lo(_Int_1_Loc)]
mov 0x3,%o2
sethi %hi(_Int_2_Loc),%o3
st %o2,[%o3+%lo(_Int_2_Loc)]
.seg "data1"
L62:
.ascii "DHRYSTONE PROGRAM, 2'ND STRING\0"
.seg "text"
set _Str_2_Loc,%o0
set L62,%o1
call _strcpy,2
nop
mov 0x1,%o4
sethi %hi(_Enum_Loc),%o5
st %o4,[%o5+%lo(_Enum_Loc)]
set _Str_1_Loc,%o0
set _Str_2_Loc,%o1
call _Func_2,2
nop
tst %o0
bne L2000000
nop
mov 1,%o7
b L2000001
nop
L2000000:
mov 0,%o7
L2000001:
sethi %hi(_Bool_Glob),%l0
st %o7,[%l0+%lo(_Bool_Glob)]
sethi %hi(_Int_1_Loc),%l1
ld [%l1+%lo(_Int_1_Loc)],%l1
sethi %hi(_Int_2_Loc),%l2
ld [%l2+%lo(_Int_2_Loc)],%l2
cmp %l1,%l2
bge L65
nop
L66:
sethi %hi(_Int_1_Loc),%o0
ld [%o0+%lo(_Int_1_Loc)],%o0
mov %o0,%o1
sll %o1,2,%o1
add %o0,%o1,%o0
sethi %hi(_Int_2_Loc),%o2
ld [%o2+%lo(_Int_2_Loc)],%o2
sub %o0,%o2,%o3
sethi %hi(_Int_3_Loc),%o4
st %o3,[%o4+%lo(_Int_3_Loc)]
sethi %hi(_Int_1_Loc),%o0
ld [%o0+%lo(_Int_1_Loc)],%o0
sethi %hi(_Int_2_Loc),%o1
ld [%o1+%lo(_Int_2_Loc)],%o1
set _Int_3_Loc,%o2
call _Proc_7,3
nop
sethi %hi(_Int_1_Loc),%o5
ld [%o5+%lo(_Int_1_Loc)],%o5
add %o5,0x1,%o5
sethi %hi(_Int_1_Loc),%o7
st %o5,[%o7+%lo(_Int_1_Loc)]
L64:
sethi %hi(_Int_1_Loc),%l0
ld [%l0+%lo(_Int_1_Loc)],%l0
sethi %hi(_Int_2_Loc),%l1
ld [%l1+%lo(_Int_2_Loc)],%l1
cmp %l0,%l1
bl L66
nop
```

```
L65:
set _Arr_1_Glob,%o0
set _Arr_2_Glob,%o1
sethi %hi(_Int_1_Loc),%o2
ld [%o2+%lo(_Int_1_Loc)],%o2
sethi %hi(_Int_3_Loc),%o3
ld [%o3+%lo(_Int_3_Loc)],%o3
call _Proc_8,4
nop
sethi %hi(_Ptr_Glob),%o0
ld [%o0+%lo(_Ptr_Glob)],%o0
call _Proc_1,1
nop
mov 0x41,%l2
sethi %hi(_Ch_Index),%l3
stb %l2,[%l3+%lo(_Ch_Index)]
sethi %hi(_Ch_Index),%l4
ldsb [%l4+%lo(_Ch_Index)],%l4
sethi %hi(_Ch_2_Glob),%l5
ldsb [%l5+%lo(_Ch_2_Glob)],%l5
cmp %l4,%l5
bg L71
nop
L72:
sethi %hi(_Ch_Index),%o0
ldsb [%o0+%lo(_Ch_Index)],%o0
mov 0x43,%o1
call _Func_1,2
nop
sethi %hi(_Enum_Loc),%o1
ld [%o1+%lo(_Enum_Loc)],%o1
cmp %o1,%o0
bne L73
nop
mov 0,%o0
set _Enum_Loc,%o1
call _Proc_6,2
nop
.seg "data1"
L75:
.ascii "DHRYSTONE PROGRAM, 3'RD STRING\0"
.seg "text"
set _Str_2_Loc,%o0
set L75,%o1
call _strcpy,2
nop
sethi %hi(_Run_Index),%o0
ld [%o0+%lo(_Run_Index)],%o0
sethi %hi(_Int_2_Loc),%o1
st %o0,[%o1+%lo(_Int_2_Loc)]
sethi %hi(_Run_Index),%o2
ld [%o2+%lo(_Run_Index)],%o2
sethi %hi(_Int_Glob),%o3
st %o2,[%o3+%lo(_Int_Glob)]
L73:
L70:
sethi %hi(_Ch_Index),%o4
ldsb [%o4+%lo(_Ch_Index)],%o4
add %o4,0x1,%o4
sll %o4,24,%o4
sra %o4,24,%o4
sethi %hi(_Ch_Index),%o5
stb %o4,[%o5+%lo(_Ch_Index)]
sethi %hi(_Ch_Index),%o7
ldsb [%o7+%lo(_Ch_Index)],%o7
sethi %hi(_Ch_2_Glob),%l0
ldsb [%l0+%lo(_Ch_2_Glob)],%l0
cmp %o7,%l0
```

```
ble L72
nop
L71:
sethi %hi(_Int_2_Loc),%o0
ld [%o0+%lo(_Int_2_Loc)],%o0
sethi %hi(_Int_1_Loc),%o1
ld [%o1+%lo(_Int_1_Loc)],%o1
call .mul,2
nop
sethi %hi(_Int_2_Loc),%l1
st %o0,[%l1+%lo(_Int_2_Loc)]
sethi %hi(_Int_2_Loc),%o0
ld [%o0+%lo(_Int_2_Loc)],%o0
sethi %hi(_Int_3_Loc),%o1
ld [%o1+%lo(_Int_3_Loc)],%o1
call .div,2
nop
sethi %hi(_Int_1_Loc),%l2
st %o0,[%l2+%lo(_Int_1_Loc)]
sethi %hi(_Int_2_Loc),%l3
ld [%l3+%lo(_Int_2_Loc)],%l3
sethi %hi(_Int_3_Loc),%l4
ld [%l4+%lo(_Int_3_Loc)],%l4
sub %l3,%l4,%l5
mov %l5,%l6
sll %l6,3,%l6
sub %l6,%l5,%l5
sethi %hi(_Int_1_Loc),%l7
ld [%l7+%lo(_Int_1_Loc)],%l7
sub %l5,%l7,%i0
sethi %hi(_Int_2_Loc),%i1
st %i0,[%i1+%lo(_Int_2_Loc)]
set _Int_1_Loc,%o0
call _Proc_2,1
nop
L57:
sethi %hi(_Run_Index),%i2
ld [%i2+%lo(_Run_Index)],%i2
add %i2,0x1,%i2
sethi %hi(_Run_Index),%i3
st %i2,[%i3+%lo(_Run_Index)]
sethi %hi(_Run_Index),%i4
ld [%i4+%lo(_Run_Index)],%i4
sethi %hi(_Number_Of_Runs),%i5
ld [%i5+%lo(_Number_Of_Runs)],%i5
cmp %i4,%i5
ble L59
nop
__trigger1:
        set     0x00003100, %g1
        stda    %g0, [%g1] 0x20
L58:
sethi %hi(_End_Time),%o0
ld [%o0+%lo(_End_Time)],%o0
sethi %hi(_Begin_Time),%o1
ld [%o1+%lo(_Begin_Time)],%o1
sub %o0,%o1,%o2
sethi %hi(_User_Time),%o3
st %o2,[%o3+%lo(_User_Time)]
LE50:
! ret
! restore
__exit:
        sethi   %hi(_romp),%o0
        ld      [%o0+%lo(_romp)],%o0
        ld      [%o0+0x74],%g1
        call    %g1,0
```

```
        nop
       LF50 = -96
LP50 = 96
LST50 = 96
LT50 = 96
.seg "data"
.seg "text"
.proc 04
.global _Proc_1
_Proc_1:
!#PROLOGUE# 0
sethi %hi(LF77),%g1
add %g1,%lo(LF77),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
ld [%fp+0x44],%o0
ld [%o0],%o1
st %o1,[%fp+-0x4]
ld [%fp+0x44],%o0
ld [%o0],%o1
sethi %hi(_Ptr_Glob),%o2
ld [%o2+%lo(_Ptr_Glob)],%o2
mov 48,%l0
L2000002:
subcc %l0,4,%l0
ld [%o2+%l0],%o4
bne L2000002
st %o4,[%o1+%l0]
ld [%fp+0x44],%l1
mov 0x5,%l2
st %l2,[%l1+0xc]
ld [%fp+-0x4],%l3
ld [%fp+0x44],%l4
ld [%l4+0xc],%l5
st %l5,[%l3+0xc]
ld [%fp+-0x4],%l6
ld [%fp+0x44],%l7
ld [%l7],%i0
st %i0,[%l6]
ld [%fp+-0x4],%o0
call _Proc_3,1
nop
ld [%fp+-0x4],%i1
ld [%i1+0x4],%i2
tst %i2
bne L80
nop
ld [%fp+-0x4],%o0
mov 0x6,%o1
st %o1,[%o0+0xc]
ld [%fp+0x44],%o2
ld [%o2+0x8],%o0
ld [%fp+-0x4],%o1
add %o1,0x8,%o1
call _Proc_6,2
nop
ld [%fp+-0x4],%o3
sethi %hi(_Ptr_Glob),%o4
ld [%o4+%lo(_Ptr_Glob)],%o4
ld [%o4],%o5
st %o5,[%o3]
ld [%fp+-0x4],%o7
ld [%o7+0xc],%o0
ld [%fp+-0x4],%o2
add %o2,0xc,%o2
```

```
mov 0xa,%o1
call _Proc_7,3
nop
b L81
nop
L80:
ld [%fp+0x44],%l0
ld [%l0],%l1
ld [%fp+0x44],%l2
mov 48,%l6
L2000003:
subcc %l6,4,%l6
ld [%l1+%l6],%l4
bne L2000003
st %l4,[%l2+%l6]
L81:
LE77:
ret
restore
        LF77 = -104
LP77 = 96
LST77 = 96
LT77 = 96
.seg "data"
.seg "text"
.proc 04
.global _Proc_2
_Proc_2:
!#PROLOGUE# 0
sethi %hi(LF82),%g1
add %g1,%lo(LF82),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
ld [%fp+0x44],%o0
ld [%o0],%o1
add %o1,0xa,%o2
st %o2,[%fp+-0x4]
L86:
sethi %hi(_Ch_1_Glob),%o3
ldsb [%o3+%lo(_Ch_1_Glob)],%o3
cmp %o3,0x41
bne L87
nop
ld [%fp+-0x4],%o0
sub %o0,0x1,%o0
st %o0,[%fp+-0x4]
ld [%fp+-0x4],%o1
sethi %hi(_Int_Glob),%o2
ld [%o2+%lo(_Int_Glob)],%o2
sub %o1,%o2,%o3
ld [%fp+0x44],%o4
st %o3,[%o4]
st %g0,[%fp+-0x8]
L87:
L85:
ld [%fp+-0x8],%o5
tst %o5
bne L86
nop
L84:
LE82:
ret
restore
        LF82 = -72
LP82 = 64
LST82 = 64
LT82 = 64
.seg "data"
.seg "text"
.proc 04
.global _Proc_3
```

```
_Proc_3:
!#PROLOGUE# 0
sethi %hi(LF88),%g1
add %g1,%lo(LF88),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
sethi %hi(_Ptr_Glob),%o0
ld [%o0+%lo(_Ptr_Glob)],%o0
tst %o0
be L90
nop
ld [%fp+0x44],%o1
sethi %hi(_Ptr_Glob),%o2
ld [%o2+%lo(_Ptr_Glob)],%o2
ld [%o2],%o3
st %o3,[%o1]
L90:
sethi %hi(_Ptr_Glob),%o2
ld [%o2+%lo(_Ptr_Glob)],%o2
add %o2,0xc,%o2
sethi %hi(_Int_Glob),%o1
ld [%o1+%lo(_Int_Glob)],%o1
mov 0xa,%o0
call _Proc_7,3
nop
LE88:
ret
restore
        LF88 = -96
LP88 = 96
LST88 = 96
LT88 = 96
.seg "data"
.seg "text"
.proc 04
.global _Proc_4
_Proc_4:
!#PROLOGUE# 0
sethi %hi(LF91),%g1
add %g1,%lo(LF91),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
sethi %hi(_Ch_1_Glob),%o0
ldsb [%o0+%lo(_Ch_1_Glob)],%o0
cmp %o0,0x41
bne L2000004
nop
mov 1,%o1
b L2000005
nop
L2000004:
mov 0,%o1
L2000005:
st %o1,[%fp+-0x4]
ld [%fp+-0x4],%o2
sethi %hi(_Bool_Glob),%o3
ld [%o3+%lo(_Bool_Glob)],%o3
or %o3,%o2,%o3
sethi %hi(_Bool_Glob),%o4
st %o3,[%o4+%lo(_Bool_Glob)]
mov 0x42,%o5
sethi %hi(_Ch_2_Glob),%o7
stb %o5,[%o7+%lo(_Ch_2_Glob)]
LE91:
ret
restore
        LF91 = -72
LP91 = 64
LST91 = 64
LT91 = 64
```

```
.seg "data"
.seg "text"
.proc 04
.global _Proc_5
_Proc_5:
!#PROLOGUE# 0
sethi %hi(LF93),%g1
add %g1,%lo(LF93),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
mov 0x41,%o0
sethi %hi(_Ch_1_Glob),%o1
stb %o0,[%o1+%lo(_Ch_1_Glob)]
sethi %hi(_Bool_Glob),%o2
st %g0,[%o2+%lo(_Bool_Glob)]
LE93:
ret
restore
        LF93 = -64
LP93 = 64
LST93 = 64
LT93 = 64
.seg "data"
LL0:
.seg "data"
.seg "text"
.proc 04
.global _Proc_6
_Proc_6:
!#PROLOGUE# 0
sethi %hi(LF28),%g1
add %g1,%lo(LF28),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %i1,[%fp+0x48]
ld [%fp+0x48],%o0
ld [%fp+0x44],%o1
st %o1,[%o0]
ld [%fp+0x44],%o0
call _Func_3,1
nop
tst %o0
bne L31
nop
ld [%fp+0x48],%o2
mov 0x3,%o3
st %o3,[%o2]
L31:
b L33
nop
L34:
ld [%fp+0x48],%o0
st %g0,[%o0]
b L32
nop
L35:
sethi %hi(_Int_Glob),%o1
ld [%o1+%lo(_Int_Glob)],%o1
cmp %o1,0x64
ble L36
nop
ld [%fp+0x48],%o2
st %g0,[%o2]
b L37
nop
L36:
ld [%fp+0x48],%o3
mov 0x3,%o4
st %o4,[%o3]
```

```
L37:
b L32
nop
L38:
ld [%fp+0x48],%o5
mov 0x1,%o7
st %o7,[%o5]
b L32
nop
L39:
b L32
nop
L40:
ld [%fp+0x48],%l0
mov 0x2,%l1
st %l1,[%l0]
b L32
nop
L33:
ld [%fp+0x44],%o0
cmp %o0,4
bgu L2000000
sll %o0,2,%o0
set L2000001,%o1
ld [%o0+%o1],%o0
jmp %o0
nop
L2000001:
.word L34
.word L35
.word L38
.word L39
.word L40
L2000000:
L32:
LE28:
ret
restore
        LF28 = -96
LP28 = 96
LST28 = 96
LT28 = 96
.seg "data"
.seg "text"
.proc 04
.global _Proc_7
_Proc_7:
!#PROLOGUE# 0
sethi %hi(LF42),%g1
add %g1,%lo(LF42),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %i1,[%fp+0x48]
st %i2,[%fp+0x4c]
ld [%fp+0x44],%o0
add %o0,0x2,%o1
st %o1,[%fp+-0x4]
ld [%fp+0x48],%o2
ld [%fp+-0x4],%o3
add %o2,%o3,%o4
ld [%fp+0x4c],%o5
st %o4,[%o5]
LE42:
ret
restore
        LF42 = -72
LP42 = 64
LST42 = 64
LT42 = 64
.seg "data"
.seg "text"
.proc 04
```

```
.global _Proc_8
_Proc_8:
!#PROLOGUE# 0
sethi %hi(LF45),%g1
add %g1,%lo(LF45),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %i1,[%fp+0x48]
st %i2,[%fp+0x4c]
st %i3,[%fp+0x50]
ld [%fp+0x4c],%o0
add %o0,0x5,%o1
st %o1,[%fp+-0x8]
ld [%fp+-0x8],%o2
sll %o2,0x2,%o3
ld [%fp+0x44],%o4
ld [%fp+0x50],%o5
st %o5,[%o4+%o3]
ld [%fp+-0x8],%o7
sll %o7,0x2,%l0
ld [%fp+0x44],%l1
add %l1,%l0,%l2
ld [%fp+-0x8],%l3
sll %l3,0x2,%l4
ld [%fp+0x44],%l5
ld [%l5+%l4],%l6
st %l6,[%l2+0x4]
ld [%fp+-0x8],%l7
sll %l7,0x2,%i0
ld [%fp+0x44],%i1
add %i1,%i0,%i2
ld [%fp+-0x8],%i3
st %i3,[%i2+0x78]
ld [%fp+-0x8],%i4
st %i4,[%fp+-0x4]
L49:
ld [%fp+-0x8],%i5
add %i5,0x1,%o0
ld [%fp+-0x4],%o1
cmp %o1,%o0
bg L48
nop
ld [%fp+-0x8],%o2
sll %o2,3,%o2
mov %o2,%o3
sll %o3,3,%o3
add %o2,%o3,%o2
sll %o3,1,%o3
add %o2,%o3,%o2
ld [%fp+0x48],%o4
add %o4,%o2,%o5
ld [%fp+-0x4],%o7
sll %o7,0x2,%l0
ld [%fp+-0x8],%l1
st %l1,[%o5+%l0]
L47:
ld [%fp+-0x4],%l2
add %l2,0x1,%l2
st %l2,[%fp+-0x4]
b L49
nop
L48:
ld [%fp+-0x8],%l3
```

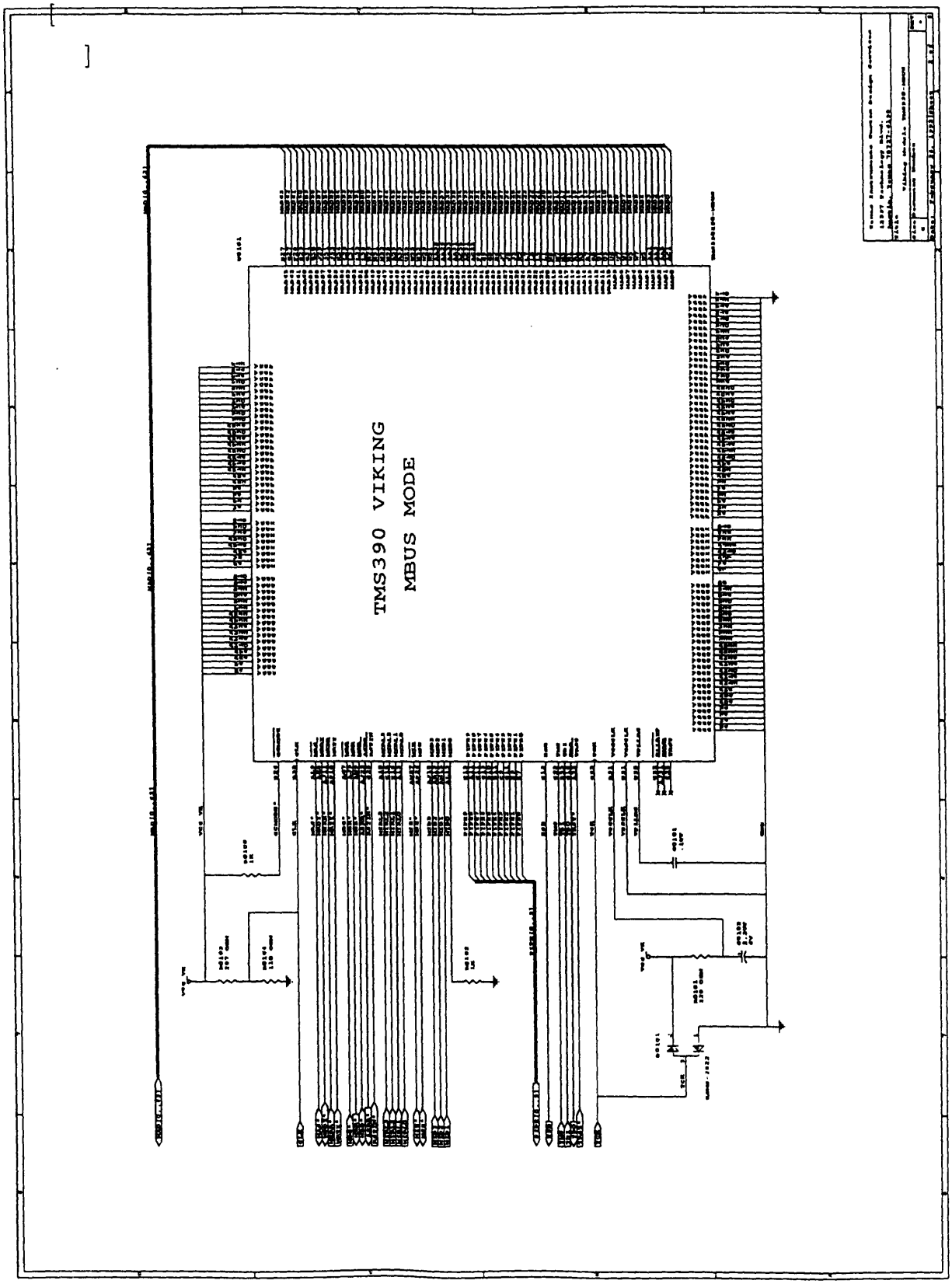
```
sll %l3,3,%l3
mov %l3,%l4
sll %l4,3,%l4
add %l3,%l4,%l3
sll %l4,1,%l4
add %l3,%l4,%l3
ld [%fp+0x48],%l5
add %l5,%l3,%l6
ld [%fp+-0x8],%l7
sll %l7,0x2,%i0
sub %i0,0x4,%i1
ld [%l6+%i1],%i2
add %i2,0x1,%i2
st %i2,[%l6+%i1]
ld [%fp+-0x8],%i3
sll %i3,3,%i3
mov %i3,%i4
sll %i4,3,%i4
add %i3,%i4,%i3
sll %i4,1,%i4
add %i3,%i4,%i3
ld [%fp+0x48],%i5
add %i5,%i3,%o0
add %o0,0xfa0,%o1
ld [%fp+-0x8],%o2
sll %o2,0x2,%o3
ld [%fp+-0x8],%o4
sll %o4,0x2,%o5
ld [%fp+0x44],%o7
ld [%o7+%o5],%l0
st %l0,[%o1+%o3]
mov 0x5,%l1
sethi %hi(_Int_Glob),%l2
st %l1,[%l2+%lo(_Int_Glob)]
LE45:
ret
restore
        LF45 = -72
LP45 = 64
LST45 = 64
LT45 = 64
.seg "data"
.seg "text"
.proc 012
.global _Func_1
_Func_1:
!#PROLOGUE# 0
sethi %hi(LF51),%g1
add %g1,%lo(LF51),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
stb %i0,[%fp+0x47]
stb %i1,[%fp+0x4b]
ldsb [%fp+0x47],%o0
stb %o0,[%fp+-0x1]
ldsb [%fp+-0x1],%o1
stb %o1,[%fp+-0x2]
ldsb [%fp+-0x2],%o2
ldsb [%fp+0x4b],%o3
cmp %o2,%o3
be L53
nop
mov 0,%o0
b LE51
nop
L53:
ldsb [%fp+-0x1],%o0
```

```
sethi %hi(_Ch_1_Glob),%o1
stb %o0,[%o1+%lo(_Ch_1_Glob)]
mov 0x1,%o0
b LE51
nop
LE51:
mov %o0,%i0
ret
restore
        LF51 = -72
LP51 = 64
LST51 = 64
LT51 = 64
.seg "data"
.seg "text"
.proc 04
.global _Func_2
_Func_2:
!#PROLOGUE# 0
sethi %hi(LF55),%g1
add %g1,%lo(LF55),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
st %i1,[%fp+0x48]
mov 0x2,%o0
st %o0,[%fp+-0x4]
L57:
ld [%fp+-0x4],%o1
cmp %o1,0x2
bg L58
nop
ld [%fp+0x44],%o2
ld [%fp+-0x4],%o3
ldsb [%o2+%o3],%o0
ld [%fp+0x48],%o4
ld [%fp+-0x4],%o5
add %o4,%o5,%o7
ldsb [%o7+0x1],%o1
call _Func_1,2
nop
tst %o0
bne L59
nop
mov 0x41,%o0
stb %o0,[%fp+-0x5]
ld [%fp+-0x4],%o1
add %o1,0x1,%o1
st %o1,[%fp+-0x4]
L59:
b L57
nop
L58:
ldsb [%fp+-0x5],%o2
cmp %o2,0x57
bl L60
nop
ldsb [%fp+-0x5],%o3
cmp %o3,0x5a
bge L60
nop
mov 0x7,%o4
st %o4,[%fp+-0x4]
L60:
ldsb [%fp+-0x5],%o5
cmp %o5,0x52
bne L61
nop
mov 0x1,%o0
b LE55
```

```
nop
L61:
ld [%fp+0x44],%o0
ld [%fp+0x48],%o1
call _strcmp,2
nop
tst %o0
ble L63
nop
ld [%fp+-0x4],%o0
add %o0,0x7,%o0
st %o0,[%fp+-0x4]
ld [%fp+-0x4],%o1
sethi %hi(_Int_Glob),%o2
st %o1,[%o2+%lo(_Int_Glob)]
mov 0x1,%o0
b LE55
nop
L63:
mov 0,%o0
b LE55
nop
LE55:
mov %o0,%i0
ret
restore
        LF55 = -104
LP55 = 96
LST55 = 96
LT55 = 96
.seg "data"
.seg "text"
.proc 04
.global _Func_3
_Func_3:
!#PROLOGUE# 0
sethi %hi(LF64),%g1
add %g1,%lo(LF64),%g1
save %sp,%g1,%sp
!#PROLOGUE# 1
st %i0,[%fp+0x44]
ld [%fp+0x44],%o0
st %o0,[%fp+-0x4]
ld [%fp+-0x4],%o1
cmp %o1,0x2
bne L66
nop
mov 0x1,%o0
b LE64
nop
L66:
mov 0,%o0
b LE64
nop
LE64:
mov %o0,%i0
ret
restore
        LF64 = -72
LP64 = 64
LST64 = 64
LT64 = 64
.seg "data"
```

Appendix D

TMX390z50-40M Schematics


TMS390 VIKING
MBUS MODE

Appendix E

System Performance Parameters

"test":

```
Program Execution Time  =  252 cycles / 33MHz  =  7.64e-6 seconds
numData$MissCycles  =  154 cycles
Data$MissCycleRate  =  154 cycles / 252 cycles  =  0.61
numData$MissEvents  =  5 events
Data$MissRate  =  5 events / 25 MEMops  =  .20
Data$MissDuration  =  54 cycles / 5 events  =  10.80 cycles/event
numFPinterlockCycles  =  5 cycles
FPinterlockCycleRate  =  5 cycles / 252 cycles  =  1.98e-2
numFPinterlockEvents  =  2 events
FPinterlockDuration  =  5 cycles / 2 events  =  2.5 cycles/event
numStallFreeCycles  =  95 cycles
numI$MissCycles  =  49 cycles
I$MissCycleRate  =  49 cycles / 95 cycles  =  0.52
numI$MissEvents  =  9 events
I$MissRate  = 9 / 64  =  0.14
I$MissDuration  =  49 cycles / 9 events  = 5.44 cycles/event
num0InstrGroup  =  50
num1InstrGroup  =  30
num2InstrGroup  =  11
num3InstrGroup  =  4
0InstrGroupFraction  =  50 / 95  =  0.53
1InstrGroupFraction  =  30 / 95  =  0.32
2InstrGroupFraction  =  11 / 95  =  0.12
3InstrGroupFraction  =  4 / 95  =  4.21e-2
numProgramInstructions  =  30 + 11 * 2 + 4 * 3  =  64
IPC  =  64 instructions / 252 cycles  =  0.25 instructions/cycle
CPI  =  252 cycles / 64 instructions  =  3.94 cycles/instruction
numMEMop  =  25
numFPop  =  5
numBRop  =  10
numALUop  =  64 - 25 - 5 - 10  =  24
numTakenBRop  =  7
numUntakenBRop  =  10 - 7  =  3
fractionTakenBRop  =  7 / 10  =  0.7
fractionUntakenBRop  =  3 / 10  =  0.3
```

"matrix6":

```
Program Execution Time  =  1599 cycles / 33MHz  =  4.85e-5 seconds
numData$MissCycles  =  240 cycles
Data$MissCycleRate  =  240 cycles / 1599 cycles  =  0.15
numData$MissEvents  =  10 events
Data$MissRate  =  10 events / 420 MEMops  =  2.38e-2
Data$MissDuration  =  240 cycles / 10 events  = 24 cycles/event
numFPinterlockCycles  =  322 cycles
FPinterlockCycleRate  =  322 cycles / 1599 cycles  =  0.20
numFPinterlockEvents  =  47 events
FPinterlockDuration  =  322 cycles / 47 events  =  6.85 cycles/event
numStallFreeCycles  =  1039 cycles
numI$MissCycles  =  202 cycles
I$MissCycleRate  =  202 cycles / 1039 cycles  =  0.19
numI$MissEvents  =  32 events
I$MissRate  =  32 / 1263  =  2.53e-2
I$MissDuration  =  202 cycles / 32 events  = 5.44 cycles/event
num0InstrGroup  =  286
num1InstrGroup  =  319
num2InstrGroup  =  358
num3InstrGroup  =  76
0InstrGroupFraction  =  286 / 1039  =  0.28
1InstrGroupFraction  =  319 / 1039  =  0.31
2InstrGroupFraction  =  358 / 1039  =  0.34
3InstrGroupFraction  =  76 / 1039  =  7.31e-2
numProgramInstructions  =  319 + 358 * 2 + 76 * 3  =  1263
IPC  =  1263 instructions / 1599 cycles  =  0.79 instructions/cycle
CPI  =  1599 cycles / 1263 instructions  =  1.27 cycles/instruction
numMEMop  =  420
numFPop  =  180
numBRop  =  81
numALUop  =  1263 - 420 - 180 - 81  =  582
numTakenBRop  =  55
numUntakenBRop  =  81 - 55  =  26
fractionTakenBRop  =  55 / 81  =  0.68
fractionUntakenBRop  =  26 / 81  =  0.32
```

"matrix200":

```
Program Execution Time  =  1,904,282 cycles / 33MHz  =  5.77e-2 seconds
numData$MissCycles  =  693,568 cycles
Data$MissCycleRate  =  693,568 cycles / 1,904,282 cycles  =  0.36
numData$MissEvents  =  79,677 events
Data$MissRate  =  79,677 events / 605,015 MEMops  =  0.13
Data$MissDuration  =  693,568 cycles / 79,677 events  =  8.70 cycles/event
numFPinterlockCycles  =  282,784 cycles
FPinterlockCycleRate  =  282,784 cycles / 693,568 cycles  =  0.41
numFPinterlockEvents  =  40,399 events
FPinterlockDuration  =  282,784 cycles / 40,399 events  =  7.00 cycles/event
numStallFreeCycles  =  928,236 cycles
numI$MissCycles  =  175 cycles
I$MissCycleRate  =  175 cycles / 928,236 cycles  =  1.89e-4
numI$MissEvents  =  38 events
I$MissRate  =  38 / 1,411,658  =  2.69e-5
I$MissDuration  =  175 cycles / 38 events  = 4.61 cycles/event
num0InstrGroup  =  80,580
num1InstrGroup  =  324,051
num2InstrGroup  =  483,205
num3InstrGroup  =  40,399
0InstrGroupFraction  =  80,580 / 928,236  =  8.68e-2
1InstrGroupFraction  =  324,051 / 928,236  =  0.35
2InstrGroupFraction  =  483,205 / 928,236  =  0.52
3InstrGroupFraction  =  40,399 / 928,236  =  4.35e-2
numProgramInstructions  =  324,051 + 483,205 * 2 + 40,399 * 3  =  1,411,658
IPC  =  1,411,658 / 1,904,282  =  0.74
CPI  =  1,904,282 / 1,411,658  =  1.35
numMEMop  =  605,015
numFPop  =  161,200
numBRop  =  81,409
numALUop  =  1,411,658 - 605,015 - 161,200 - 81,409  =  564,034
numTakenBRop  =  40,809
numUntakenBRop  =  81,409 - 40,809  =  40,600
fractionTakenBRop  =  40,809 / 81,409  =  0.50
fractionUntakenBRop  =  40,600 / 81,409  =  0.50
```

"dhry":

```
Program Execution Time  =  7,124,729,181 cycles / 33MHz  =  215.90 seconds
                        3.6 minutes
numData$MissCycles  =  5,831,728,245 cycles
Data$MissCycleRate  =  5,831,728,245 cycles / 7,124,729,181 cycles  =  0.82
numData$MissEvents  =  363,000,010 events
Data$MissRate  =  363,000,010 events / 740,000,005 MEMops  =  0.49
Data$MissDuration  =  5,831,728,245 cycles / 363,000,010 events  =
16.07 cycles/event
numFPinterlockCycles  =  0 cycles
FPinterlockCycleRate  = 0
numFPinterlockEvents  =  0 event
FPinterlockDuration  =  0 cycle/event
numStallFreeCycles  =  1,293,000,945 cycles
numI$MissCycles  =  40,000,896 cycles
I$MissCycleRate  =  40,000,896 cycles / 1,293,000,945 cycles  =  3.09e-2
numI$MissEvents  =  20,000,139 events
I$MissRate  =  20,000,139 / 1,642,000,016  =  1.22e-2
I$MissDuration  =  40,000,896 cycles / 20,000,139 events  = 2.00 cycles/event
num0InstrGroup  =  154,000,898
num1InstrGroup  =  719,000,076
num2InstrGroup  =  336,999,970
num3InstrGroup  =  83,000,000
0InstrGroupFraction  =  154,000,898 / 1,293,000,945  =  0.12
1InstrGroupFraction  =  719,000,076 / 1,293,000,945  =  0.56
2InstrGroupFraction  =  336,999,970 / 1,293,000,945  =  0.26
3InstrGroupFraction  =  83,000,000 / 1,293,000,945  =  6.42e-2
numProgramInstructions  =  719,000,076 + 336,999,970 * 2 + 83,000,000 * 3
=  1,642,000,016
IPC  =  1,642,000,016 instructions / 7,124,729,181 cycles  =  0.23
CPI  =  7,124,729,181 cycles / 1,642,000,016 instructions  =  4.34
numMEMop  =  740,000,005
numFPop  =  0
numBRop  =  229,000,001
numALUop  =  1,642,000,016 - 740,000,005 - 229,000,001
=  673,000,010
numTakenBRop  =  152,999,999
numUntakenBRop  =  229,000,001 - 152,999,999  =  76,000,002
fractionTakenBRop  =  152,999,999 / 229,000,001  =  0.67
fractionUntakenBRop  =  76,000,002 / 229,000,001  =  0.33
```

Bibliography

[1] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. "An Analysis of SPARC and MIPS Instruction Set Utlization on the SPEC Benchmarks." *Architectural Support for Programming Languages and Operating Systems*, pages 290-302, 1991.

[2] Micheal D. Smith, Mark Horowits, and Monica S. Lam. "Efficient Superscalar Performance Through Boosting." *Architectural Support for Programming Languages and Operating Systems*, pages 248-261, 1992.

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

[4] Jerome C. Huck and Michael J. Flynn. *Analyzing Computer Architectures.* IEEE Computer Society Press, Washington, D.C. 20036, 1989.

[5] Roland L. Lee, Alex Y. Kwok, and Faye A Briggs. "The Floating Point Performance of a Superscalar SPARC Processor." *Architectural Support for Programming Languages and Operating Systems*, pages 28-39, 1991.

[6] Texas Instruments Incorporated. *SuperSPARC User's Guide*, 1992.

[7] SPARC International, Version 8. *The SPARC Architecture Manual.* Prentice Hall, 1992.

[8] Joseph Torrellas, Anoop Gupta, and John Hennessy. "Characterizing the Caching and Sysnchronization Performance of a Multiprocessor Operating System." *Architectural Support for a Programming Languagues and Operating Systems*, pages 162-174, 1992.

[9] Richard D. Trauben. *Methods and Apparatus for Unobtrusively Monitoring Processor States and Characterizing Bottlenecks in a Pipelined Processor Executing Grouped Instructions*, 1992.

[10] Stephen A. Ward and Robert H. Halstead. *Computation Structures.* MIT Press, Cambridge, Massachusetts, 1990.

[11] Greg Blanck and Steve Krueger. "The SuperSPARC ™ Microprocessor." *IEEE Compcon Proceeding*, pages 136-141, 1992.

[12] Fuad Abu-Nofal et al. "A Three-Million-Transistor Microprocessor." *IEEE International Solid-State Circuits Conference*, pages 108-109, 257, 1992.