FIRST EDITION

OpenSPARC™ Internals

OpenSPARC T1/T2
CHIP MULTITHREADED THROUGHPUT COMPUTING

Sun microsystems

David L. Weaver, Editor

www.OpenSPARC.net

OpenSPARC™ Internals

OpenSPARC T1/T2 CMT Throughput Computing

David L. Weaver, Editor

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A. 650-960-1300

ISBN 978-0-557-01974-8
First printing, October 2008

Contents

Preface

Open-source software? Sure, everyone has heard of that. We all take advantage of it as we navigate through the Internet (OpenSolaris™, Java™, Linux, Apache, Perl, etc.) and even when we sit down to relax with TiVo (Linux) or watch a Blu-Ray disc (Java).

But open-source *hardware* … eh? What is open-source *hardware*?! Small amounts of computer hardware Intellectual Property (IP) have been available for many years in open-source form, typically as circuit descriptions written in an RTL (Register Transfer Level) language such as Verilog or VHDL. However, until now, few large hardware designs have been available in open-source form. One of the most complex designs imaginable is for a complete microprocessor; with the notable exception of the LEON 32-bit SPARC® processor, *none* have been available in open-source form until recently.

In March 2006, the complete design of Sun Microsystems' UltraSPARC™ T1 microprocessor was released—in open-source form, it was named OpenSPARC™ T1. In early 2008, its successor, OpenSPARC™ T2, was also released in open-source form. These were the first (and still only) 64-bit microprocessors ever open-sourced. They were also the first (and still only) CMT (chip multithreaded) microprocessors ever open-sourced. Both designs are freely available from the OpenSPARC™ website, `http://www.OpenSPARC.net`, to anyone. These downloads include not only the processor design source code but also simulation tools, design verification suites, Hypervisor source code, and other helpful tools. Variants that easily synthesize for FPGA targets are also available.

Organization of Book

This book is intended as a sort of "tour guide" for those who have downloaded the OpenSPARC T1 or OpenSPARC T2 design, or might be considering doing so. •

- Chapter 1, *Introducing Chip Multithreaded (CMT) Processors*, addresses the question "why build a multithreaded processor?"
- Chapter 2, *OpenSPARC Designs*, describes some example uses for these designs.
- Chapter 3, *Architecture Overview*, backs up a step and describes the architecture on which OpenSPARC processors are based.
- Chapter 4, *OpenSPARC T1 and T2 Processor Implementations*, dives into the microarchitecture of both OpenSPARC T1 and OpenSPARC T2.
- Chapter 5, *OpenSPARC T2 Memory Subsystem — A Deeper Look*, describes the memory system.
- Chapter 6, *OpenSPARC Processor Configuration*, explains how to configure a synthesized design from the actual RTL code and provides a couple of examples of useful modifications to the design.
- The design verification methodology used by Sun to verify the processors on which OpenSPARC is based is explained in Chapter 7, *OpenSPARC Design Verification Methodology*.
- Chapter 8, *Operating Systems for OpenSPARC T1*, lists operating systems that already run on OpenSPARC and gives an overview of how you can port your own operating system to run on top of Hypervisor on an OpenSPARC implementation.
- In Chapter 9, *Tools for Developers*, the emphasis shifts to software, as the chapter describes the tools that are available for developing high-performance software for OpenSPARC.
- Software is again the focus in Chapter 10, *System Simulation, Bringup, and Verification*, which discusses OpenSPARC system simulation, RTL, and co-simulation.
- Chapter 11, *OpenSPARC Extension and Modification—Case Study*, presents a real-world example of getting started with the OpenSPARC RTL and tools.

- Appendix A, *Overview: OpenSPARC T1/ T2 Source Code and Environment Setup*, gives an overview of the source code trees for OpenSPARC T1 and OpenSPARC T2 and describes how to set up a functional development environment around them.
- Appendix B and Appendix C provide "deep dives" into the OpenSPARC T1 and OpenSPARC T2 designs, respectively.
- Use of the Verification suites for OpenSPARC T1 and OpenSPARC T2 is found, respectively, in Appendix D and Appendix E.
- Appendix F, *OpenSPARC Resources*, lists URLs for OpenSPARC resources available on the World Wide Web.
- Lastly, Appendix G provides a glossary of terminology used in this book.

Notational Conventions

Fonts are used as follows:

- *Italic* font is used for emphasis, book titles, and the first instance of a word that is defined.
- *Italic* font is also used for terms where substitution is expected, for example, "`fcc`*n*", "virtual processor *n*", or "*reg_plus_imm*".
- *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."
- lowercase arial font is used for register field names (named bits) and instruction field names, for example: "The rs1 field contains...."
- UPPERCASE ARIAL font is used for register names; for example, FSR.
- `TYPEWRITER` (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: `%f0`, `ASI_PRIMARY`, `execute_state`.
- When a register field is shown along with its containing register name, they are separated by a period ('.'), for example, "FSR.cexc".

Case, underscores, and hyphens are used as follows.

- UPPERCASE words are acronyms or instruction names. Some common acronyms appear in the glossary in Appendix G, *OpenSPARC Terminology*. **Note:** Names of some instructions contain both upper- and lower-case letters.

- An underscore character joins words in register, register field, exception, and trap names. **Note:** Such words may be split across lines at the underbar without an intervening hyphen. For example: “This is true whenever the integer_condition_
  code field....”
- A hyphen joins multiple words in a variable name; for example, “*ioram-instance-name*”.

The following notational conventions are used:

- The left arrow symbol ($\leftarrow$) is the assignment operator. For example, “PC $\leftarrow$ PC + 1” means that the Program Counter (PC) is incremented by 1.
- Square brackets ([ ]) are used in two different ways, distinguishable by the context in which they are used:
  - Square brackets indicate indexing into an array. For example, TT[TL] means the element of the Trap Type (TT) array, as indexed by the contents of the Trap Level (TL) register.
  - Square brackets are also used to indicate optional additions/extensions to symbol names. For example, “ST[D|Q]F” expands to all three of “STF”, “STDF”, and “STQF”. Similarly, `ASI_PRIMARY[_LITTLE]` indicates two related address space identifiers, `ASI_PRIMARY` and `ASI_PRIMARY_LITTLE`. (Contrast with the use of angle brackets, below)
- Angle brackets (< >) indicate mandatory additions/extensions to symbol names. For example, “ST<D|Q>F” expands to mean “STDF” and “STQF”. (Contrast with the second use of square brackets, above.)
- Curly braces ({ }) indicate a bit field within a register or instruction. For example, CCR{4} refers to bit 4 in the Condition Code register.
- A consecutive set of values is indicated by specifying the upper and lower limit of the set separated by a colon (:), for example, CCR{3:0} refers to the set of four least significant bits of register CCR.

Notation for numbers is as follows.

- Numbers are decimal (base-10) unless otherwise indicated. Numbers in other bases are followed by a numeric subscript indicating their base (for example, 1001_2, FFFF 0000_{16}).

  Long binary and hexadecimal numbers within the text have spaces inserted every four characters to improve readability.
- Within C language or assembly language examples, numbers may be preceded by “0x” to indicate base-16 (hexadecimal) notation (for example, 0xFFFF0000).

Acknowledgments

OpenSPARC Internals is the result of collaboration among many contributors. I would particularly like to acknowledge the following people for their key contributions:

- **Lawrence Spracklen**, for contributing the text of Chapter 1
- **Sreenivas Vadlapatla**, for coordinating the content of Chapter 5
- **Paul Jordan**, for contributing portions of Chapter 5
- **Tom Thatcher**, for contributing the bulk of Chapter 6 and Appendix A
- **Jared Smolens**, for contributing the “cookbook” examples at the end of Chapter 6
- **Jai Kumar**, for contributing the text of Chapter 7
- **Kevin Clague**, for contributions to Chapter 7
- **Gopal Reddy**, for contributing the text of Chapter 8
- **Darryl Gove** (author of the *Solaris Application Programming* book), for contributing Chapter 9
- **Alex Tsirfa**, for contributing the text of Chapter 10
- **Fabrizio Fazzino**, for contributing the text of Chapter 11
- **Durgam Vahia**, for extracting the text of Appendix B through Appendix E from the sources listed below

The following people contributed to the quality of this book by providing meticulous reviews of its drafts and answers to critical questions: **Paul Jordan**, **Jared Smolens**, **Jeff Brooks**, and **Aaron Wynn**.

Last, but *far* from least, thanks to **Mary Lou Nohr**, for pulling it all together and making everyone else look good (yet *again*)

Sources

I would also like to acknowledge the sources, listed in the table below, from which Appendixes B through E were derived.

App.	Source	Part # / Date / Rev
B	*OpenSPARC T1 Microarchitecture Specification*	819-6650-10 Aug. 2006, Rev. A
C	*OpenSPARC T2 Microarchitecture Specification* *OpenSPARC T2 System-On-Chip (SoC) Micrarchitecture Specification*	820-2545-10 July 2007, Rev. 5 820-2620-05 July 2007, Rev. 5
D	*OpenSPARC T1 Design and Verification User's Guide* (Chapter 3)	819-5019-12, Mar 2007, Rev. A
E	*OpenSPARC T2 Design and Verification User's Guide* (Chapter 3)	820-2729-10, Dec 2007, Rev. A

Editor's Note

We hope you find *OpenSPARC Internals* accurate, readable, and stimulating.

—***David Weaver***
Editor, *OpenSPARC Internals*

Corrections and other comments regarding this book can be emailed to:
`OpenSPARC-book-editor@sun.com`

CHAPTER 1

Introducing Chip Multithreaded (CMT) Processors

Much of the material in this chapter was leveraged from L. Spracklen and S. G. Abraham, "Chip Multithreading: Opportunities and Challenges," in 11th International Symposium on High-Performance Computer Architecture, 2005.

Over the last few decades microprocessor performance has increased exponentially, with processor architects successfully achieving significant gains in single-thread performance from one processor generation to the next. Semiconductor technology has been the main driver for this increase, with faster transistors allowing rapid increases in clock speed to today's multi-GHz frequencies. In addition to these frequency increases, each new technology generation has essentially doubled the number of available transistors. As a result, architects have been able to aggressively chase increased single-threaded performance by using a range of expensive microarchitectural techniques, such as~~,~~ superscalar issue, out-of-order issue, on-chip caching, and deep pipelines supported by sophisticated branch predictors.

However, process technology challenges, including power constraints, the memory wall, and ever-increasing difficulties in extracting further instruction-level parallelism (ILP), are all conspiring to limit the performance of individual processors in the future. While recent attempts at improving single-thread performance through even deeper pipelines have led to impressive clock frequencies, these clock frequencies have not translated into significantly better performance in comparison with less aggressive designs. As a result, microprocessor frequency, which used to increase exponentially, has now leveled off, with most processors operating in the 2–4 GHz range.

This combination of the limited realizable ILP, practical limits to pipelining, and a “power ceiling” imposed by cost-effective cooling considerations have conspired to limit future performance increases within conventional processor cores. Accordingly, processor designers are searching for new ways to effectively utilize their ever-increasing transistor budgets.

The techniques being embraced across the microprocessor industry are chip multiprocessors (CMPs) and chip multithreaded (CMT) processors. CMP, as the name implies, is simply a group of processors integrated onto the same chip. The individual processors typically have comparable performance to their single-core brethren, but for workloads with sufficient thread-level parallelism (TLP), the aggregate performance delivered by the processor can be many times that delivered by a single-core processor. Most current processors adopt this approach and simply involve the replication of existing single-processor processor cores on a single die.

Moving beyond these simple CMP processors, chip multithreaded (CMT) processors go one step further and support many simultaneous hardware strands (or threads) of execution per core by simultaneous multithreading (SMT) techniques. SMT effectively combats increasing latencies by enabling multiple strands to share many of the resources within the core, including the execution resources. With each strand spending a significant portion of time stalled waiting for off-chip misses to complete, each strand’s utilization of the core’s execution resources is extremely low. SMT improves the utilization of key resources and reduces the sensitivity of an application to off-chip misses. Similarly, as with CMP, multiple cores can share chip resources such as the memory controller, off-chip bandwidth, and the level-2/level-3 cache, improving the utilization of these resources.

The benefits of CMT processors are apparent in a wide variety for application spaces. For instance, in the commercial space, server workloads are broadly characterized by high levels of TLP, low ILP, and large working sets. The potential for further improvements in overall single-thread performance is limited; on-chip cycles per instruction (CPI) cannot be improved significantly because of low ILP, and off-chip CPI is large and growing because of relative increases in memory latency. However, typical server applications concurrently serve a large number of users or clients; for instance, a database server may have hundreds of active processes, each associated with a different client. Furthermore, these processes are currently multithreaded to hide disk access latencies. This structure leads to high levels of TLP. Thus, it is extremely attractive to couple the high TLP in the application domain with support for multiple threads of execution on a processor chip.

Though the arguments for CMT processors are often made in the context of overlapping memory latencies, memory bandwidth considerations also play a significant role. New memory technologies, such as fully buffered DIMMs (FBDs), have higher bandwidths (for example, 60 GB/s/chip), as well as higher latencies (for example, 130 ns), pushing up their bandwidth-delay product to 60 GB/s × 130 ns = 7800 bytes. The processor chip's pins represent an expensive resource, and to keep these pins fully utilized (assuming a cache line size of 64 bytes), the processor chip must sustain 7800/64 or over 100 parallel requests. To put this in perspective, a single strand on an aggressive out-of-order processor core generates less than two parallel requests on typical server workloads: therefore, a large number of strands are required to sustain a high utilization of the memory ports.

Finally, power considerations also favor CMT processors. Given the almost cubic dependence between core frequency and power consumption, the latter drops dramatically with reductions in frequency. As a result, for workloads with adequate TLP, doubling the number of cores and halving the frequency delivers roughly equivalent performance while reducing power consumption by a factor of four.

Evolution of CMTs

Given the exponential growth in transistors per chip over time, a rule of thumb is that a board design becomes a chip design in ten years or less. Thus, most industry observers expected that chip-level multiprocessing would eventually become a dominant design trend. The case for a single-chip multiprocessor was presented as early as 1996 by Kunle Olukotun's team at Stanford University. Their Stanford Hydra CMP processor design called for the integration of four MIPS-based processors on a single chip. A DEC/ Compaq research team proposed the incorporation of eight simple Alpha cores and a two-level cache hierarchy on a single chip (code-named Piranha) and estimated a simulated performance of three times that of a single-core, next-generation Alpha processor for on-line transaction processing workloads.

As early as the mid-1990s, Sun recognized the problems that would soon face processor designers as a result of the rapidly increasing clock frequencies required to improve single-thread performance. In response, Sun defined the MAJC architecture to target thread-level parallelism. Providing well-defined support for both CMP and SMT processors, MAJC architecture was industry's first step toward general-purpose CMT processors. Shortly after publishing the MAJC architecture, Sun announced its first MAJC-compliant processor (MAJC-5200), a dual-core CMT processor with cores sharing an L1 data cache.

Subsequently, Sun moved its SPARC processor family toward the CMP design point. In 2003, Sun announced two CMP SPARC processors: Gemini, a dual-core UltraSPARC II derivative; and UltraSPARC IV. These first-generation CMP processors were derived from earlier uniprocessor designs, and the two cores did not share any resources other than off-chip datapaths. In most CMP designs, it is preferable to share the outermost caches, because doing so localizes coherency traffic between the strands and optimizes inter-strand communication in the chip—allowing very fine-grained thread interaction (microparallelism). In 2003, Sun also announced its second-generation CMP processor, UltraSPARC IV+, a follow-on to the UltraSPARC IV processor, in which the on-chip L2 and off-chip L3 caches are shared between the two cores.

In 2006, Sun introduced a 32-way CMT SPARC processor, called UltraSPARC T1, for which the entire design, including the cores, is optimized for a CMT design point. UltraSPARC T1 has eight cores; each core is a four-way SMT with its own private L1 caches. All eight cores share a 3-Mbyte, 12-way level-2 cache~~,~~. Since UltraSPARC T1 is targeted at commercial server workloads with high TLP, low ILP, and large working sets, the ability to support many strands and therefore many concurrent off-chip misses is key to overall performance. Thus, to accommodate eight cores, each core supports single issue and has a fairly short pipeline.

Sun's most recent CMT processor is the UltraSPARC T2 processor. The UltraSPARC T2 processor provides double the threads of the UltraSPARC T1 processor (eight threads per core), as well as improved single-thread performance, additional level-2 cache resources (increased size and associativity), and improved support for floating-point operations.

Sun's move toward the CMT design has been mirrored throughout industry. In 2001, IBM introduced the dual-core POWER-4 processor and recently released second-generation CMT processors, the POWER-5 and POWER-6 processors, in which each core supports 2-way SMT. While this fundamental shift in processor design was initially confined to the high-end server processors, where the target workloads are the most thread-rich, this change has recently begun to spread to desktop processors. AMD and Intel have also subsequently released multicore CMP processors, starting with dual-core CMPs and more recently quad-core CMP processors. Further, Intel has announced that its next-generation quad-core processors will support 2-way SMT, providing a total of eight threads per chip.

CMT is emerging as the dominant trend in general-purpose processor design, with manufacturers discussing their multicore plans beyond their initial quad-core offerings. Similar to the CISC-to-RISC shift that enabled an entire processor to fit on a single chip and internalized all communication between pipeline stages to within a chip, the move to CMT represents a fundamental shift in processor design that internalizes much of the communication between processors to within a chip.

Future CMT Designs

An attractive proposition for future CMT design is to just double the number of cores per chip every generation since a new process technology essentially doubles the transistor budget. Little design effort is expended on the cores, and performance is almost doubled every process generation on workloads with sufficient TLP. Though reusing existing core designs is an attractive option, this approach may not scale well beyond a couple of process generations. Processor designs are already pushing the limits of power dissipation. For the total power consumption to be restrained, the power dissipation of each core must be halved in each generation. In the past, supply voltage scaling delivered most of the required power reduction, but indications are that voltage scaling will not be sufficient by itself. Though well-known techniques, such as clock gating and frequency scaling, may be quite effective in the short term, more research is needed to develop low-power, high-performance cores for future CMT designs.

Further, given the significant area cost associated with high-performance cores, for a fixed area and power budget, the CMP design choice is between a small number of high-performance (high frequency, aggressive out-of-order, large issue width) cores or multiple simple (low frequency, in-order, limited issue width) cores. For workloads with sufficient TLP, the simpler core solution may deliver superior chipwide performance at a fraction of the power. However, for applications with limited TLP, unless speculative parallelism can be exploited, CMT performance will be poor. One possible solution is to support heterogeneous cores, potentially providing multiple simple cores for thread-rich workloads and a single more complex core to provide robust performance for single-threaded applications.

Another interesting opportunity for CMT processors is support for on-chip hardware accelerators. Hardware accelerators improve performance on certain specialized tasks and off-load work from the general-purpose processor. Additionally, on-chip hardware accelerators may be an order of magnitude more power efficient than the general-purpose processor and may be significantly more efficient than off-chip accelerators (for example, eliminating the off-chip traffic required to communicate to an off-chip accelerator). Although high cost and low utilization typically make on-chip hardware accelerators unattractive for traditional processors, the cost of an accelerator can be amortized over many strands, thanks to the high degree of resource sharing associated with CMTs. While a wide variety of hardware accelerators can be envisaged, emerging trends make an extremely compelling case for supporting on-chip network off-load engines and cryptographic accelerators. The future processors will afford opportunities for accelerating other functionality. For instance, with the increasing usage of XML-formatted data, it may become attractive to have hardware support XML parsing and processing.

Finally, for the same amount of off-chip bandwidth to be maintained per core, the total off-chip bandwidth for the processor chip must also double every process generation. Processor designers can meet the bandwidth need by adding more pins or increasing the bandwidth per pin. However, the maximum number of pins per package is only increasing at a rate of 10 percent per generation. Further packaging costs per pin are barely going down with each new generation and increase significantly with pin count. As a result, efforts have recently focused on increasing the per-pin bandwidth by innovations in the processor chip to DRAM memory interconnect through technologies such as double data rate and fully buffered DIMMs. Additional benefits can be obtained by doing more with the available bandwidth; for instance, by compressing off-chip traffic or exploiting silentness to minimize the bandwidth required to perform write-back operations. Compression of the on-chip caches themselves can also improve performance, but the (significant) additional latency that is introduced as a result of the decompression overhead must be carefully balanced against the benefits of the reduced miss rate, favoring adaptive compression strategies.

As a result, going forward we are likely to see an ever-increasing proportion of CMT processors designed from the ground-up in order to deliver ever-increasing performance while satisfying these power and bandwidth constraints.

CHAPTER **2**

OpenSPARC Designs

Sun Microsystems began shipping the UltraSPARC T1 chip multithreaded (CMT) processor in December 2005. Sun surprised the industry by announcing that it would not only ship the processor but also *open-source* that processor—a first in the industry. By March 2006, UltraSPARC T1 had been open-sourced in a distribution called OpenSPARC T1, available on `http://OpenSPARC.net`.

In 2007, Sun began shipping its newer, more advanced UltraSPARC T2 processor, and open-sourced the bulk of that design as OpenSPARC T2.

The "source code" for both designs offered on OpenSPARC.net is comprehensive, including not just millions of lines of the hardware description language (Verilog, a form of "register transfer logic"—RTL) for these microprocessors, but also scripts to compile ("synthesize") that source code into hardware implementations, source code of processor and full-system simulators, prepackaged operating system images to boot on the simulators, source code to the Hypervisor software layer, a large suite of verification software, and thousands of pages of architecture and implementation specification documents.

This book is intended as a "getting started" companion to both OpenSPARC T1 and OpenSPARC T2. In this chapter, we begin that association by addressing this question: *Now that Sun has open-sourced OpenSPARC T1 and T2, what can they be used for?*

One thing is certain: the real-world uses to which OpenSPARC will be put will be infinitely more diverse and interesting than anything that could be suggested in this book! Nonetheless, this short chapter offers a few ideas, in the hope that they will stimulate even more creative thinking …

2.1 Academic Uses for OpenSPARC

The utility of OpenSPARC in academia is limited only by students' imaginations.

The most common academic use of OpenSPARC to date is as a complete example processor architecture and/or implementation. It can be used in coursework areas such as computer architecture, VLSI design, compiler code generation/optimization, and general computer engineering.

In university lab courses, OpenSPARC provides a design that can be used as a known-good starting point for assigned projects.

OpenSPARC can be used as a basis for compiler research, such as for code generation/optimization for highly threaded target processors or for experimenting with instruction set changes and additions.

OpenSPARC is already in use in multiple FPGA-based projects at universities. For more information, visit:

`http://www.opensparc.net/fpga/index.html`

For more information on programs supporting academic use of OpenSPARC, including availability of the Xilinx OpenSPARC FPGA Board, visit web page:

`http://www.OpenSPARC.net/edu/university-program.html`

Specific questions about university programs can be posted on the OpenSPARC general forum at:

`http://forums.sun.com/forum.jspa?forumID=837`

or emailed to OpenSPARC-UniversityProgram@sun.com.

Many of the commercial applications of OpenSPARC, mentioned in the following section, suggest corresponding academic uses.

2.2 Commercial Uses for OpenSPARC

OpenSPARC provides a springboard for design of commercial processors. By starting from a complete, known-good design—including a full verification suite—the time-to-market for a new custom processor can be drastically slashed.

Derivative processors ranging from a simple single-core, single-thread design all the way up through an 8-core, 64-thread design can rapidly be synthesized from OpenSPARC T1 or T2.

2.2.1 FPGA Implementation

An OpenSPARC design can be synthesized and loaded into a field-programmable gate array (FPGA) device. This can be used in several ways:

- An FPGA version of the processor can be used for product prototyping, allowing rapid design iteration
- An FPGA can be used to provide a high-speed simulation engine for a processor under development
- For extreme time-to-market needs where production cost per processor isn't critical, a processor could even be shipped in FPGA form. This could also be useful if the processor itself needs to be field-upgradable via a software download.

2.2.2 Design Minimization

Portions of a standard OpenSPARC design that are not needed for the target application can be stripped out, to make the resulting processor smaller, cheaper, faster, and/or with higher yield rates. For example, for a network routing application, perhaps hardware floating-point operations are superfluous—in which case, the FPU(s) can be removed, saving die area and reducing verification effort.

2.2.3 Coprocessors

Specialized coprocessors can be incorporated into a processor based on OpenSPARC. OpenSPARC T2, for example, comes with a coprocessor containing two 10 Gbit/second Ethernet transceivers (the network interface unit or "NIU"). Coprocessors can be added for any conceivable purpose, including (but hardly limited to) the following:

- Network routing
- Floating-point acceleration
- Cryptographic processing
- I/O compression/decompression engines
- Audio compression/decompression (codecs)
- Video codecs
- I/O interface units for embedded devices such as displays or input sensors

2.2.4 OpenSPARC as Test Input to CAD/ EDA Tools

The OpenSPARC source code (Verilog RTL) provides a large, real-world input dataset for CAD/EDA tools. It can be used to test the robustness of CAD tools and simulators. Many major commercial CAD/EDA tool vendors are already using OpenSPARC this way!

CHAPTER 3

Architecture Overview

OpenSPARC processors are based on a processor architecture named the UltraSPARC Architecture. The OpenSPARC T1 design is based on the UltraSPARC Architecture 2005, and OpenSPARC T2 is based on the UltraSPARC Architecture 2007. This chapter is intended as an overview of the architecture; more details can be found in the *UltraSPARC Architecture 2005 Specification* and the *UltraSPARC Architecture 2007 Specification*.

The UltraSPARC Architecture is descended from the SPARC V9 architecture and complies fully with the “Level 1” (nonprivileged) SPARC V9 specification.

The UltraSPARC Architecture supports 32-bit and 64-bit integer and 32-bit, 64-bit, and 128-bit floating-point as its principal data types. The 32-bit and 64-bit floating-point types conform to IEEE Std 754-1985. The 128-bit floating-point type conforms to IEEE Std 1596.5-1992. The architecture defines general-purpose integer, floating-point, and special state/status register instructions, all encoded in 32-bit-wide instruction formats. The load/store instructions address a linear, 2^{64}-byte virtual address space.

As used here, the word *architecture* refers to the processor features that are visible to an assembly language programmer or to a compiler code generator. It does not include details of the implementation that are not visible or easily observable by software, nor those that only affect timing (performance).

The chapter contains these sections:

- *The UltraSPARC Architecture* on page 12
- *Processor Architecture* on page 15
- *Instructions* on page 17
- *Traps* on page 23
- *Chip-Level Multithreading (CMT)* on page 23

3.1 The UltraSPARC Architecture

This section briefly describes features, attributes, and components of the UltraSPARC Architecture and, further, describes correct implementation of the architecture specification and SPARC V9-compliance levels.

3.1.1 Features

The UltraSPARC Architecture, like its ancestor SPARC V9, includes the following principal features:

- **A linear 64-bit address space** with 64-bit addressing.
- **32-bit wide instructions** — These are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.
- **Few addressing modes** — A memory address is given as either "register + register" or "register + immediate".
- **Triadic register addresses** — Most computational instructions operate on two register operands or one register and a constant and place the result in a third register.
- **A large windowed register file** — At any one instant, a program sees 8 global integer registers plus a 24-register window of a larger register file. The windowed registers can be used as a cache of procedure arguments, local values, and return addresses.
- **Floating point** — The architecture provides an IEEE 754-compatible floating-point instruction set, operating on a separate register file that provides 32 single-precision (32-bit), 32 double-precision (64-bit), and 16 quad-precision (128-bit) overlayed registers.
- **Fast trap handlers** — Traps are vectored through a table.
- **Multiprocessor synchronization instructions** — Multiple variations of atomic load-store memory operations are supported.
- **Predicted branches** — The branch with prediction instructions allows the compiler or assembly language programmer to give the hardware a hint about whether a branch will be taken.
- **Branch elimination instructions** — Several instructions can be used to eliminate branches altogether (for example, Move on Condition). Eliminating branches increases performance in superscalar and superpipelined implementations.

- **Hardware trap stack** — A hardware trap stack is provided to allow nested traps. It contains all of the machine state necessary to return to the previous trap level. The trap stack makes the handling of faults and error conditions simpler, faster, and safer.

In addition, UltraSPARC Architecture includes the following features that were not present in the SPARC V9 specification:

- **Hyperprivileged mode**— This mode simplifies porting of operating systems, supports far greater portability of operating system (privileged) software, supports the ability to run multiple simultaneous guest operating systems, and provides more robust handling of error conditions. Hyperprivileged mode is described in detail in the Hyperprivileged version of the *UltraSPARC Architecture 2005 Specification* or the *UltraSPARC Architecture 2007 Specification*.
- **Multiple levels of global registers** — Instead of the two 8-register sets of global registers specified in the SPARC V9 architecture, the UltraSPARC Architecture provides multiple sets; typically, one set is used at each trap level.
- **Extended instruction set** — The UltraSPARC Architecture provides many instruction set extensions, including the VIS instruction set for "vector" (SIMD) data operations.
- **More detailed, specific instruction descriptions** — UltraSPARC Architecture specifications provide many more details regarding what exceptions can be generated by each instruction, and the specific conditions under which those exceptions can occur, than did SPARC V9. Also, detailed lists of valid ASIs are provided for each load/store instruction from/to alternate space.
- **Detailed MMU architecture** — Although some details of the UltraSPARC MMU architecture are necessarily implementation-specific, UltraSPARC Architecture specifications provide a blueprint for the UltraSPARC MMU, including software view (TTEs and TSBs) and MMU hardware control registers.
- **Chip-level multithreading (CMT)** — The UltraSPARC Architecture provides a control architecture for highly threaded processor implementations.

3.1.2 Attributes

The UltraSPARC Architecture is a processor *instruction set architecture* (ISA) derived from SPARC V8 and SPARC V9, which in turn come from a reduced instruction set computer (RISC) lineage. As an architecture, the UltraSPARC

Architecture allows for a spectrum of processor and system *implementations* at a variety of price/performance points for a range of applications, including scientific or engineering, programming, real-time, and commercial applications. OpenSPARC further extends the possible breadth of design possibilities by opening up key implementations to be studied, enhanced, or redesigned by anyone in the community.

3.1.2.1 Design Goals

The UltraSPARC Architecture is designed to be a target for optimizing compilers and high-performance hardware implementations. The *UltraSPARC Architecture 2005* and *UltraSPARC Architecture 2007 Specification* documents provide design specs against which an implementation can be verified, using appropriate verification software.

3.1.2.2 Register Windows

The UltraSPARC Architecture architecture is derived from the SPARC architecture, which was formulated at Sun Microsystems in 1984 through 1987. The SPARC architecture is, in turn, based on the RISC I and II designs engineered at the University of California at Berkeley from 1980 through 1982. The SPARC "register window" architecture, pioneered in the UC Berkeley designs, allows for straightforward, high-performance compilers and a reduction in memory load/store instructions.

Note that privileged software, not user programs, manages the register windows. Privileged software can save a minimum number of registers (approximately 24) during a context switch, thereby optimizing context-switch latency.

3.1.3 System Components

The UltraSPARC Architecture allows for a spectrum of subarchitectures, such as cache system, I/O, and memory management unit (MMU).

3.1.3.1 Binary Compatibility

An important mandate for the UltraSPARC Architecture is compatibility across implementations of the architecture for application (nonprivileged) software, down to the binary level. Binaries executed in nonprivileged mode should behave identically on all UltraSPARC Architecture systems when those systems are running an operating system known to provide a standard execution environment. One example of such a standard environment is the SPARC V9 Application Binary Interface (ABI).

Although different UltraSPARC Architecture systems can execute nonprivileged programs at different rates, they will generate the same results as long as they are run under the same memory model. See Chapter 9, *Memory*, in an UltraSPARC Architecture specification for more information.

Additionally, UltraSPARC Architecture 2005 and UltraSPARC Architecture 2007 are are upward-compatible from SPARC V9 for applications running in nonprivileged mode that conform to the SPARC V9 ABI and upward-compatible from SPARC V8 for applications running in nonprivileged mode that conform to the SPARC V8 ABI.

An OpenSPARC implementation may or may not maintain the same binary compatibility, depending on how the implementation has been modified and what software execution environment is run on it.

3.1.3.2 UltraSPARC Architecture MMU

UltraSPARC Architecture defines a common MMU architecture (see Chapter 14, *Memory Management*, in any UltraSPARC Architecture specification for details). Some specifics are left implementation-dependent.

3.1.3.3 Privileged Software

UltraSPARC Architecture does not assume that all implementations must execute identical privileged software (operating systems) or hyperprivileged software (hypervisors). Thus, certain traits that are visible to privileged software may be tailored to the requirements of the system.

3.2 Processor Architecture

An UltraSPARC Architecture processor—therefore an OpenSPARC processor—logically consists of an integer unit (IU) and a floating-point unit (FPU), each with its own registers. This organization allows for implementations with concurrent integer and floating-point instruction execution. Integer registers are 64 bits wide; floating-point registers are 32, 64, or 128 bits wide. Instruction operands are single registers, register pairs, register quadruples, or immediate constants.

A *virtual processor* (synonym: *strand*) is the hardware containing the state for execution of a software thread. A *physical core* is the hardware required to execute instructions from one or more software threads, including resources shared among strands. A complete *processor* comprises one or more physical cores and is the physical module that plugs into a system.

An OpenSPARC virtual processor can run in *nonprivileged* mode, *privileged* mode, or *hyperprivileged* mode. In hyperprivileged mode, the processor can execute any instruction, including privileged instructions. In privileged mode, the processor can execute nonprivileged and privileged instructions. In nonprivileged mode, the processor can only execute nonprivileged instructions. In nonprivileged or privileged mode, an attempt to execute an instruction requiring greater privilege than the current mode causes a trap to hyperprivileged software.

3.2.1 Integer Unit (IU)

An OpenSPARC implementation's integer unit contains the general-purpose registers and controls the overall operation of the virtual processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU.

An UltraSPARC Architecture implementation may contain from 72 to 640 general-purpose 64-bit R registers. This corresponds to a grouping of the registers into a number of sets of global R registers plus a circular stack of *N_REG_WINDOWS* sets of 16 registers each, known as register windows. The number of register windows present (*N_REG_WINDOWS*) is implementation dependent, within the range of 3 to 32 (inclusive). In an unmodified OpenSPARC T1 or T2 implementation, *N_REG_WINDOWS* = 8.

3.2.2 Floating-Point Unit (FPU)

An OpenSPARC FPU has thirty-two 32-bit (single-precision) floating-point registers, thirty-two 64-bit (double-precision) floating-point registers, and sixteen 128-bit (quad-precision) floating-point registers, some of which overlap (as described in detail in UltraSPARC Architecture specifications).

If no FPU is present, then it appears to software as if the FPU is permanently disabled.

If the FPU is not enabled, then an attempt to execute a floating-point instruction generates an *fp_disabled* trap and the *fp_disabled* trap handler software must either

- Enable the FPU (if present) and reexecute the trapping instruction, or
- Emulate the trapping instruction in software.

3.3 Instructions

Instructions fall into the following basic categories:

- Memory access
- Integer arithmetic / logical / shift
- Control transfer
- State register access
- Floating-point operate
- Conditional move
- Register window management
- SIMD (single instruction, multiple data) instructions

These classes are discussed in the following subsections.

3.3.1 Memory Access

Load, store, load-store, and PREFETCH instructions are the only instructions that access memory. They use two R registers or an R register and a signed 13-bit immediate value to calculate a 64-bit, byte-aligned memory address. The integer unit appends an ASI to this address.

The destination field of the load/store instruction specifies either one or two R registers or one, two, or four F registers that supply the data for a store or that receive the data from a load.

Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and extended-word (64-bit) accesses. There are versions of integer load instructions that perform either sign-extension or zero-extension on 8-bit, 16-bit, and 32-bit values as they are loaded into a 64-bit destination register. Floating-point load and store instructions support word, doubleword, and quadword[1] memory accesses.

[1]. OpenSPARC T1 and T2 processors do not implement the LDQF instruction in hardware; it generates an exception and is emulated in hyperprivileged software.

CASA, CASXA, and LDSTUB are special atomic memory access instructions that concurrent processes use for synchronization and memory updates.

Note The SWAP instruction is also specified, but it is deprecated and should not be used in newly developed software.

The (nonportable) LDTXA instruction supplies an atomic 128-bit (16-byte) load that is important in certain system software applications.

3.3.1.1 Memory Alignment Restrictions

A memory access on an OpenSPARC virtual processor must typically be aligned on an address boundary greater than or equal to the size of the datum being accessed. An iproperly aligned address in a load, store, or load-store instruction may trigger an exception and cause a subsequent trap. For details, see the *Memory Alignment Restrictions* section in an UltraSPARC Architecture specification.

3.3.1.2 Addressing Conventions

An unmodified OpenSPARC processor uses big-endian byte order by default: the address of a quadword, doubleword, word, or halfword is the address of its most significant byte. Increasing the address means decreasing the significance of the unit being accessed. All instruction accesses are performed using big-endian byte order.

An unmodified OpenSPARC processor also supports little-endian byte order for data accesses only: the address of a quadword, doubleword, word, or halfword is the address of its least significant byte. Increasing the address means increasing the significance of the data unit being accessed.

3.3.1.3 Addressing Range

An OpenSPARC implementation supports a 64-bit virtual address space. The supported range of virtual addresses is restricted to two equal-sized ranges at the extreme upper and lower ends of 64-bit addresses; that is, for n-bit virtual addresses, the valid address ranges are 0 to $2^{n-1} - 1$ and $2^{64} - 2^{n-1}$ to $2^{64} - 1$. See the *OpenSPARC T1 Implementation Supplement* or *OpenSPARC T2 Implementation Supplement* for details.

3.3.1.4 Load/Store Alternate

Versions of load/store instructions, the *load/store alternate* instructions, can specify an arbitrary 8-bit address space identifier for the load/store data access.

Access to alternate spaces 00_{16}–$2F_{16}$ is restricted to privileged and hyperprivileged software, access to alternate spaces 30_{16}–$7F_{16}$ is restricted to hyperprivileged software, and access to alternate spaces 80_{16}–FF_{16} is unrestricted. Some of the ASIs are available for implementation-dependent uses. Privileged and hyperprivileged software can use the implementation-dependent ASIs to access special protected registers, such as MMU control registers, cache control registers, virtual processor state registers, and other processor-dependent or system-dependent values. See the *Address Space Identifiers (ASIs)* chapter in an UltraSPARC Architecture specification for more information.

Alternate space addressing is also provided for the atomic memory access instructions LDSTUBA, CASA, and CASXA.

3.3.1.5 Separate Instruction and Data Memories

The interpretation of addresses in an unmodified OpenSPARC process is “split”; instruction references use one caching and translation mechanism and data references use another, although the same underlying main memory is shared.

In such split-memory systems, the coherency mechanism may be split, so a write[1] into data memory is not immediately reflected in instruction memory. For this reason, programs that modify their own instruction stream (self-modifying code[2]) and that wish to be portable across all UltraSPARC Architecture (and SPARC V9) processors must issue FLUSH instructions, or a system call with a similar effect, to bring the instruction and data caches into a consistent state.

An UltraSPARC Architecture virtual processor may or may not have coherent instruction and data caches. Even if an implementation does have coherent instruction and data caches, a FLUSH instruction is required for self-modifying code—not for cache coherency, but to flush pipeline instruction buffers that contain unmodified instructions which may have been subsequently modified.

1. This includes use of store instructions (executed on the same or another virtual processor) that write to instruction memory, or any other means of writing into instruction memory (for example, DMA).

2. This is practiced, for example, by software such as debuggers and dynamic linkers.

3.3.1.6 Input/Output (I/O)

The UltraSPARC Architecture assumes that input/output registers are accessed through load/store alternate instructions, normal load/store instructions, or read/write Ancillary State register instructions (RDasr, WRasr).

3.3.1.7 Memory Synchronization

Two instructions are used for synchronization of memory operations: FLUSH and MEMBAR. Their operation is explained in *Flush Instruction Memory* and *Memory Barrier* sections, respectively, of UltraSPARC Architecture specifications.

3.3.2 Integer Arithmetic / Logical / Shift Instructions

The arithmetic/logical/shift instructions perform arithmetic, tagged arithmetic, logical, and shift operations. With one exception, these instructions compute a result that is a function of two source operands; the result is either written into a destination register or discarded. The exception, SETHI, can be used in combination with other arithmetic and/or logical instructions to create a constant in an R register.

Shift instructions shift the contents of an R register left or right by a given number of bits ("shift count"). The shift distance is specified by a constant in the instruction or by the contents of an R register.

3.3.3 Control Transfer

Control-transfer instructions (CTIs) include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are delayed; that is, the instruction immediately following a control-transfer instruction in logical sequence is dispatched before the control transfer to the target address is completed. Note that the next instruction in logical sequence may not be the instruction following the control-transfer instruction in memory.

The instruction following a delayed control-transfer instruction is called a *delay* instruction. Setting the *annul bit* in a conditional delayed control-transfer instruction causes the delay instruction to be annulled (that is, to have no effect) if and only if the branch is not taken. Setting the annul bit in an *un*conditional delayed control-transfer instruction ("branch always") causes the delay instruction to be always annulled.

Branch and CALL instructions use PC-relative displacements. The jump and link (JMPL) and return (RETURN) instructions use a register-indirect target address. They compute their target addresses either as the sum of two R registers or as the sum of an R register and a 13-bit signed immediate value. The "branch on condition codes without prediction" instruction provides a displacement of ±8 Mbytes; the "branch on condition codes with prediction" instruction provides a displacement of ±1 Mbyte; the "branch on register contents" instruction provides a displacement of ±128 Kbytes; and the CALL instruction's 30-bit word displacement allows a control transfer to any address within ± 2 gigabytes (± 2^{31} bytes).

Note | The return from privileged trap instructions (DONE and RETRY) get their target address from the appropriate TPC or TNPC register.

3.3.4 State Register Access

This section describes the following state registers:

- Ancillary state registers
- Read and write privileged state registers
- Read and writer hyperprivileged state registers

3.3.4.1 Ancillary State Registers

The read and write ancillary state register instructions read and write the contents of ancillary state registers visible to nonprivileged software (Y, CCR, ASI, PC, TICK, and FPRS) and some registers visible only to privileged and hyperprivileged software (PCR, SOFTINT, TICK_CMPR, and STICK_CMPR).

3.3.4.2 PR State Registers

The read and write privileged register instructions (RDPR and WRPR) read and write the contents of state registers visible only to privileged and hyperprivileged software (TPC, TNPC, TSTATE, TT, TICK, TBA, PSTATE, TL, PIL, CWP, CANSAVE, CANRESTORE, CLEANWIN, OTHERWIN, and WSTATE).

3.3.4.3 HPR State Registers

The read and write hyperprivileged register instructions (RDHPR and WRHPR) read and write the contents of state registers visible only to hyperprivileged software (HPSTATE, HTSTATE, HINTP, HVER, and HSTICK_CMPR).

3.3.5 Floating-Point Operate

Floating-point operate (FPop) instructions perform all floating-point calculations; they are register-to-register instructions that operate on the floating-point registers. FPops compute a result that is a function of one, two, or three source operands. The groups of instructions that are considered FPops are listed in the *Floating-Point Operate (FPop) Instructions* section of UltraSPARC Architecture specifications.

3.3.6 Conditional Move

Conditional move instructions conditionally copy a value from a source register to a destination register, depending on an integer or floating-point condition code or on the contents of an integer register. These instructions can be used to reduce the number of branches in software.

3.3.7 Register Window Management

Register window instructions manage the register windows. SAVE and RESTORE are nonprivileged and cause a register window to be pushed or popped. FLUSHW is nonprivileged and causes all of the windows except the current one to be flushed to memory. SAVED and RESTORED are used by privileged software to end a window spill or fill trap handler.

3.3.8 SIMD

An unmodified OpenSPARC processor includes SIMD (single instruction, multiple data) instructions, also known as “vector” instructions, which allow a single instruction to perform the same operation on multiple data items, totaling 64 bits, such as eight 8-bit, four 16-bit, or two 32-bit data items. These operations are part of the “VIS” instruction set extensions.

3.4 Traps

A *trap* is a vectored transfer of control to privileged or hyperprivileged software through a trap table that may contain the first 8 instructions (32 for some frequently used traps) of each trap handler. The base address of the table is established by software in a state register (the Trap Base Address register, TBA, or the Hyperprivileged Trap Base register, HTBA). The displacement within the table is encoded in the type number of each trap and the level of the trap. Part of the trap table is reserved for hardware traps, and part of it is reserved for software traps generated by trap (Tcc) instructions.

A trap causes the current PC and NPC to be saved in the TPC and TNPC registers. It also causes the CCR, ASI, PSTATE, and CWP registers to be saved in TSTATE. TPC, TNPC, and TSTATE are entries in a hardware trap stack, where the number of entries in the trap stack is equal to the number of supported trap levels. A trap causes hyperprivileged state to be saved in the HTSTATE trap stack. A trap also sets bits in the PSTATE (and, in some cases, HPSTATE) register and typically increments the GL register. Normally, the CWP is not changed by a trap; on a window spill or fill trap, however, the CWP is changed to point to the register window to be saved or restored.

A trap can be caused by a Tcc instruction, an asynchronous exception, an instruction-induced exception, or an interrupt request not directly related to a particular instruction. Before executing each instruction, a virtual processor determines if there are any pending exceptions or interrupt requests. If any are pending, the virtual processor selects the highest-priority exception or interrupt request and causes a trap.

See the *Traps* chapter in an UltraSPARC Architecture specification for a complete description of traps.

3.5 Chip-Level Multithreading (CMT)

An OpenSPARC implementation may include multiple virtual processor cores within the processor (“chip”) to provide a dense, high-throughput system. This may be achieved by having a combination of multiple physical processor cores and/or multiple strands (threads) per physical processor core, referred to as chip-level multithreaded (CMT) processors. CMT-specific hyperprivileged registers are used for identification and configuration of CMT processors.

The CMT programming model describes a common interface between hardware (CMT registers) and software

The common CMT registers and the CMT programming model are described in the *Chip-Level Multithreading (CMT)* chapter in UltraSPARC Architecture specifications.

CHAPTER **4**

OpenSPARC T1 and T2 Processor Implementations

This chapter introduces the OpenSPARC T1 and OpenSPARC T2 chip-level multithreaded (CMT) processors in the following sections:

4.1 General Background

OpenSPARC T1 is the first chip multiprocessor that fully implements Sun's Throughput Computing initiative. OpenSPARC T2 is the follow-on chip multi-threaded (CMT) processor to the OpenSPARC T1 processor. Throughput Computing is a technique that takes advantage of the thread-level parallelism that is present in most commercial workloads. Unlike desktop workloads, which often have a small number of threads concurrently running, most commercial workloads achieve their scalability by employing large pools of concurrent threads.

Historically, microprocessors have been designed to target desktop workloads, and as a result have focused on running a single thread as quickly as possible. Single-thread performance is achieved in these microprocessors by a combination of extremely deep pipelines (over 20 stages in Pentium 4) and by execution of multiple instructions in parallel (referred to as instruction-level parallelism, or ILP). The basic tenet behind Throughput Computing is that exploiting ILP and deep pipelining has reached the point of diminishing returns and as a result, current microprocessors do not utilize their underlying hardware very efficiently.

For many commercial workloads, the physical processor core will be idle most of the time waiting on memory, and even when it is executing, it will often be able to utilize only a small fraction of its wide execution width. So rather than building a large and complex ILP processor that sits idle most of the time, build a number of small, single-issue physical processor cores that employ multithreading built in the same chip area. Combining multiple physical processors cores on a single chip with multiple hardware-supported threads (strands) per physical processor core allows very high performance for highly threaded commercial applications. This approach is called thread-level parallelism (TLP). The difference between TLP and ILP is shown in FIGURE 4-1.

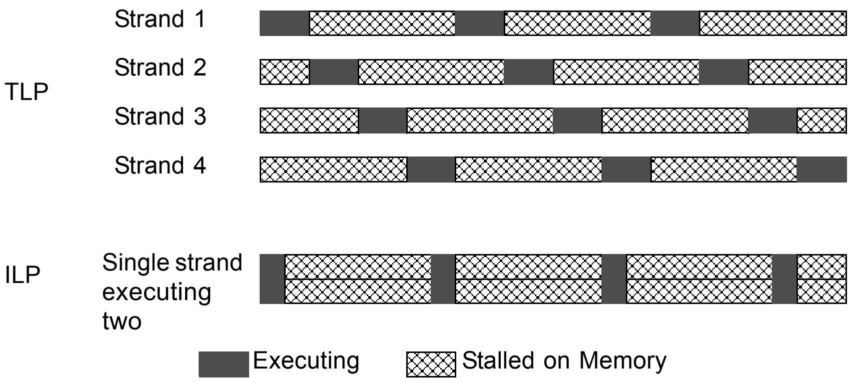


FIGURE 4-1 Differences Between TLP and ILP

The memory stall time of one strand can often be overlapped with execution of other strands on the same physical processor core, and multiple physical processor cores run their strands in parallel. In the ideal case, shown in FIGURE 4-1, memory latency can be completely overlapped with execution of other strands. In contrast, instruction-level parallelism simply shortens the time to execute instructions, and does not help much in overlapping execution with memory latency.[1]

[1]. Processors that employ out-of-order ILP can overlap some memory latency with execution. However, this overlap is typically limited to shorter memory latency events such as L1 cache misses that hit in the L2 cache. Longer memory latency events such as main memory accesses are rarely overlapped to a significant degree with execution by an out-of-order processor.

Given this ability to overlap execution with memory latency, why don't more processors utilize TLP? The answer is that designing processors is a mostly evolutionary process, and the ubiquitous deeply pipelined, wide ILP physical processor cores of today are the evolutionary outgrowth from a time when the CPU was the bottleneck in delivering good performance.

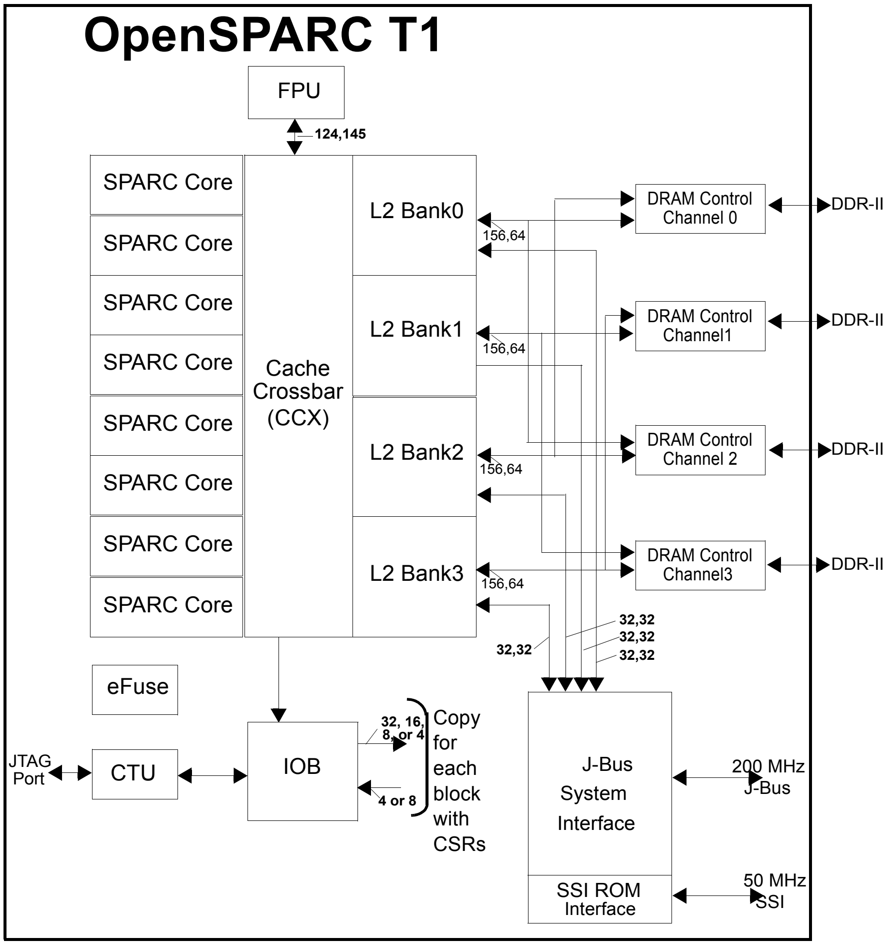
With physical processor cores capable of multiple-GHz clocking, the performance bottleneck has shifted to the memory and I/O subsystems and TLP has an obvious advantage over ILP for tolerating the large I/O and memory latency prevalent in commercial applications. Of course, every architectural technique has its advantages and disadvantages. The one disadvantage of employing TLP over ILP is that execution of a single strand may be slower on a TLP processor than on an ILP processor. With physical processor cores running at frequencies well over 1 GHz, a strand capable of executing only a single instruction per cycle is fully capable of completing tasks in the time required by the application, making this disadvantage a nonissue for nearly all commercial applications.

4.2 OpenSPARC T1 Overview

OpenSPARC T1 is a single-chip multiprocessor. OpenSPARC T1 contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for four virtual processors (or "strands"). These four strands run simultaneously, with the instructions from each of the four strands executed round-robin by the single-issue pipeline. When a strand encounters a long-latency event, such as a cache miss, it is marked unavailable and instructions are not issued from that strand until the long-latency event is resolved. Round-robin execution of the remaining available strands continues while the long-latency event of the first strand is resolved.

Each OpenSPARC T1 physical core has a 16-Kbyte, 4-way associative instruction cache (32-byte lines), 8-Kbyte, 4-way associative data cache (16-byte lines), 64-entry fully associative instruction Translation Lookaside Buffer (TLB), and 64-entry fully associative data TLB that are shared by the four strands. The eight SPARC physical cores are connected through a crossbar to an on-chip unified 3-Mbyte, 12-way associative L2 cache (with 64-byte lines). The L2 cache is banked four ways to provide sufficient bandwidth for the eight OpenSPARC T1 physical cores. The L2 cache connects to four on-chip DRAM controllers, which directly interface to DDR2-SDRAM. In addition, an on-chip J-Bus controller and several on-chip I/O-mapped control registers are accessible to the SPARC physical cores. Traffic from the J-Bus coherently interacts with the L2 cache.

A block diagram of the OpenSPARC T1 chip is shown in FIGURE 4-2.



Notes:
(1) Blocks not scaled to physical size.
(2) Bus widths are labeled as in#,out#, where “in” is into CCX or L2.

FIGURE 4-2 OpenSPARC T1 Chip Block Diagram

4.3 OpenSPARC T1 Components

This section describes each component in OpenSPARC T1 in these subsections.

- *SPARC Physical Core* on this page
- *Floating-Point Unit (FPU)* on page 30
- *L2 Cache* on page 31
- *DRAM Controller* on page 31
- *I/O Bridge (IOB) Unit* on page 31
- *J-Bus Interface (JBI)* on page 32
- *SSI ROM Interface* on page 32
- *Clock and Test Unit (CTU)* on page 32
- *EFuse* on page 33

4.3.1 OpenSPARC T1 Physical Core

Each OpenSPARC T1 physical core has hardware support for four strands. This support consists of a full register file (with eight register windows) per strand, with most of the ASI, ASR, and privileged registers replicated per strand. The four strands share the instruction and data caches and TLBs. An autodemap[1] feature is included with the TLBs to allow the multiple strands to update the TLB without locking.

The core pipeline consists of six stages: Fetch, Switch, Decode, Execute, Memory, and Writeback. As shown in FIGURE 4-3, the Switch stage contains a strand instruction register for each strand. One of the strands is picked by the strand scheduler and the current instruction for that strand is issued to the pipe. While this is done, the hardware fetches the next instruction for that strand and updates the strand instruction register.

The scheduled instruction proceeds down the rest of the stages of the pipe, similar to instruction execution in a single-strand RISC machine. It is decoded in the Decode stage. The register file access also happens at this time. In the Execute stage, all arithmetic and logical operations take place. The memory address is calculated in this stage. The data cache is accessed in the Memory stage and the instruction is committed in the Writeback stage. All traps are signaled in this stage.

1. Autodemap causes an existing TLB entry to be automatically removed when a new entry is installed with the same virtual page number (VPN) and same page size.

Instructions are classified as either short or long latency instructions. Upon encountering a long latency instruction or other stall condition in a certain strand, the strand scheduler stops scheduling that strand for further execution. Scheduling commences again when the long latency instruction completes or the stall condition clears.

FIGURE 4-3 illustrates the OpenSPARC T1 physical core.

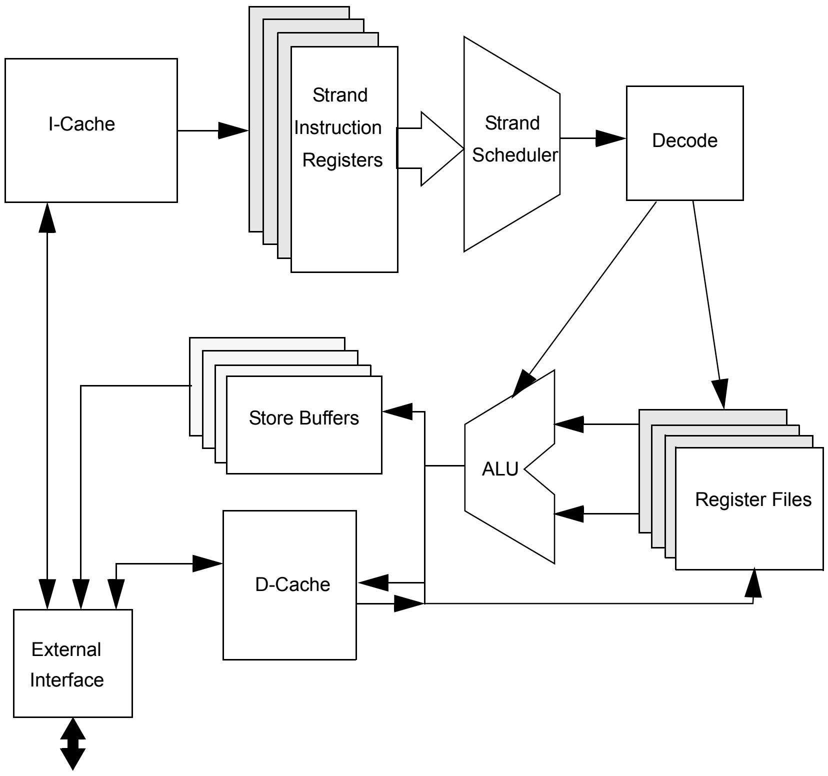


FIGURE 4-3 OpenSPARC T1 Core Block Diagram

4.3.2 Floating-Point Unit (FPU)

A single floating-point unit is shared by all eight OpenSPARC T1 physical cores. The shared floating-point unit is sufficient for most commercial applications, in which fewer than 1% of instructions typically involve floating-point operations.

4.3.3 L2 Cache

The L2 cache is banked four ways, with the bank selection based on physical address bits 7:6. The cache is 3-Mbyte, 12-way set associative with pseudo-LRU replacement (replacement is based on a used-bit scheme), and has a line size of 64 bytes. Unloaded access time is 23 cycles for an L1 data cache miss and 22 cycles for an L1 instruction cache miss.

4.3.4 DRAM Controller

OpenSPARC T1's DRAM Controller is banked four ways[1], with each L2 bank interacting with exactly one DRAM Controller bank. The DRAM Controller is interleaved based on physical address bits 7:6, so each DRAM Controller bank must have the same amount of memory installed and enabled.

OpenSPARC T1 uses DDR2 DIMMs and can support one or two ranks of stacked or unstacked DIMMs. Each DRAM bank/port is two DIMMs wide (128-bit + 16-bit ECC). All installed DIMMs on an individual bank/port must be identical, and the same total amount of memory (number of bytes) must be installed on each DRAM Controller port. The DRAM controller frequency is an exact ratio of the CMP core frequency, where the CMP core frequency must be at least 4× the DRAM controller frequency. The DDR (double data rate) data buses, of course, transfer data at twice the frequency of the DRAM Controller frequency.

The DRAM Controller also supports a small memory configuration mode, using only two DRAM ports. In this mode, L2 banks 0 and 2 are serviced by DRAM port 0, and L2 banks 1 and 3 are serviced by DRAM port 1. The installed memory on each of these ports is still two DIMMs wide.

4.3.5 I/O Bridge (IOB) Unit

The IOB performs an address decode on I/O-addressable transactions and directs them to the appropriate internal block or to the appropriate external interface (J-Bus or SSI). In addition, the IOB maintains the register status for external interrupts.

1. A two-bank option is available for cost-constrained minimal memory configurations.

4.3.6 J-Bus Interface (JBI)

J-Bus is the interconnect between OpenSPARC T1 and the I/O subsystem. It is a 200 MHz, 128-bit-wide, multiplexed address/data bus, used predominantly for DMA traffic, plus the PIO traffic to control it.

The JBI is the block that interfaces to J-Bus, receiving and responding to DMA requests, routing them to the appropriate L2 banks, and also issuing PIO transactions on behalf of the strands and forwarding responses back.

4.3.7 SSI ROM Interface

OpenSPARC T1 has a 50 Mbit/s serial interface (SSI) which connects to an external FPGA which interfaces to the BOOT ROM. In addition, the SSI interface supports PIO accesses across the SSI, thus supporting optional CSRs or other interfaces within the FPGA.

4.3.8 Clock and Test Unit (CTU)

The CTU contains the clock generation, reset, and JTAG circuitry.

OpenSPARC T1 has a single PLL, which takes the J-Bus clock as its input reference, where the PLL output is divided down to generate the CMP core clocks (for OpenSPARC T1 and caches), the DRAM clock (for the DRAM controller and external DIMMs), and internal J-Bus clock (for IOB and JBI). Thus, all OpenSPARC T1 clocks are ratioed. Sync pulses are generated to control transmission of signals and data across clock domain boundaries.

The CTU has the state machines for internal reset sequencing, which includes logic to reset the PLL and signal when the PLL is locked, updating clock ratios on warm resets (if so programmed), enabling clocks to each block in turn, and distributing reset so that its assertion is seen simultaneously in all clock domains.

The CTU also contains the JTAG block, which allows access to the shadow scan chains, plus has a CREG interface that allows the JTAG to issue reads of any I/O-addressable register, some ASI locations, and any memory location while OpenSPARC T1 is in operation.

4.3.9 EFuse

The eFuse (electronic fuse) block contains configuration information that is electronically burned in as part of manufacturing, including part serial number and strand-available information.

4.4 OpenSPARC T2 Overview

OpenSPARC T2 is a single chip multithreaded (CMT) processor. OpenSPARC T2 contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for eight processors, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The floating-point and memory pipelines are shared by all eight strands. The eight strands are hard-partitioned into two groups of four, and the four strands within a group share a single integer pipeline.

While all eight strands run simultaneously, at any given time at most two strands will be active in the physical core, and those two strands will be issuing either a pair of integer pipeline operations, an integer operation and a floating-point operation, an integer operation and a memory operation, or a floating-point operation and a memory operation. Strands are switched on a cycle-by-cycle basis between the available strands within the hard-partitioned group of four, using a least recently issued priority scheme.

When a strand encounters a long-latency event, such as a cache miss, it is marked unavailable and instructions will not be issued from that strand until the long-latency event is resolved. Execution of the remaining available strands will continue while the long-latency event of the first strand is resolved.

Each OpenSPARC T2 physical core has a 16-Kbyte, 8-way associative instruction cache (32-byte lines), 8-Kbyte, 4-way associative data cache (16-byte lines), 64-entry fully-associative instruction TLB, and 128-entry fully associative data TLB that are shared by the eight strands.—The eight OpenSPARC T2 physical cores are connected through a crossbar to an on-chip unified 4-Mbyte, 16-way associative L2 cache (64-byte lines).

The L2 cache is banked eight ways to provide sufficient bandwidth for the eight OpenSPARC T2 physical cores. The L2 cache connects to four on-chip DRAM Controllers, which directly interface to a pair of fully buffered DIMM

(FBD) channels. In addition, two 1-Gbit/10-Gbit Ethernet MACs and several on-chip I/O-mapped control registers are accessible to the SPARC physical cores.

A block diagram of the OpenSPARC T2 chip is shown in FIGURE 4-4.

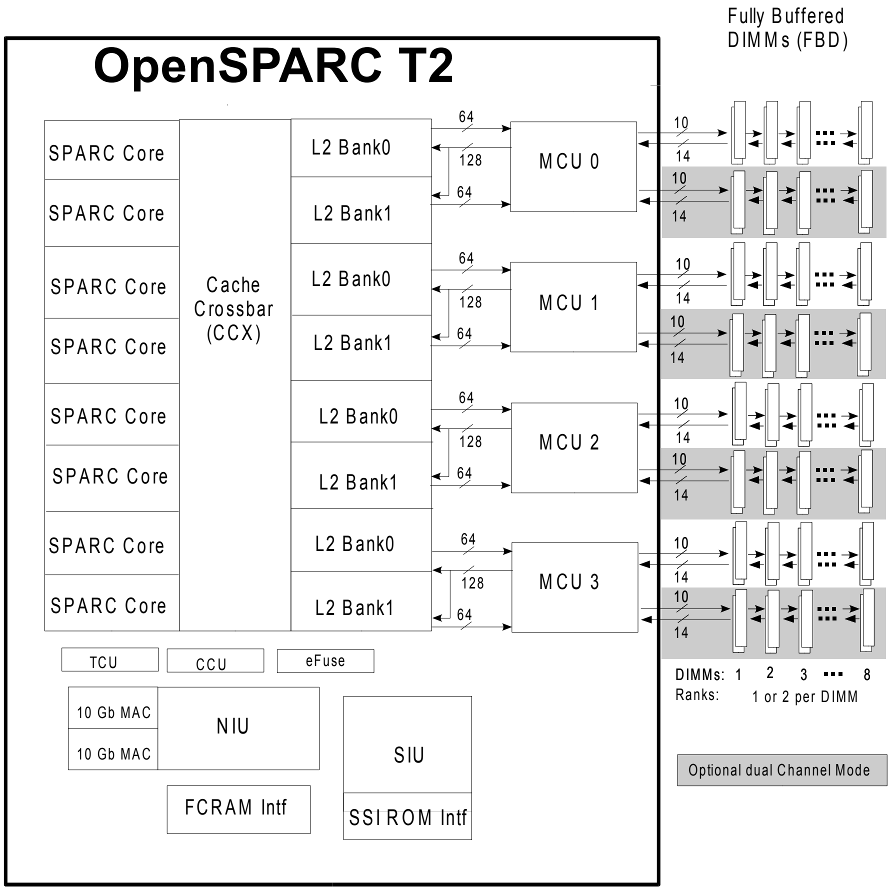


FIGURE 4-4 OpenSPARC T2 Chip Block Diagram

4.5 OpenSPARC T2 Components

This section describes the major components in OpenSPARC T2.

4.5.1 OpenSPARC T2 Physical Core

Each OpenSPARC T2 physical core has hardware support for eight strands. This support consists of a full register file (with eight register windows) per strand, with most of the ASI, ASR, and privileged registers replicated per strand. The eight strands share the instruction and data caches and TLBs. An autodemap feature is included with the TLBs to allow the multiple strands to update the TLB without locking.

Each OpenSPARC T2 physical core contains a floating-point unit, shared by all eight strands. The floating-point unit performs single- and double-precision floating-point operations, graphics operations, and integer multiply and divide operations.

4.5.2 L2 Cache

The L2 cache is banked eight ways. To provide for better partial-die recovery, OpenSPARC T2 can also be configured in 4-bank and 2-bank modes (with 1/2 and 1/4 the total cache size respectively). Bank selection is based on physical address bits 8:6 for 8 banks, 7:6 for 4 banks, and 6 for 2 banks. The cache is 4 Mbytes, 16-way set associative, and uses index hashing. The line size is 64 bytes.

4.5.3 Memory Controller Unit (MCU)

OpenSPARC T2 has four MCUs, one for each memory branch with a pair of L2 banks interacting with exactly one DRAM branch. The branches are interleaved based on physical address bits 7:6, and support 1–16 DDR2 DIMMs. Each memory branch is two FBD channels wide. A branch may use only one of the FBD channels in a reduced power configuration.

Each DRAM branch operates independently and can have a different memory size and a different kind of DIMM (for example, a different number of ranks or different CAS latency). Software should not use address space larger than four times the lowest memory capacity in a branch because the cache lines are interleaved across branches. The DRAM Controller frequency is the same as that of the DDR (double data rate) data buses, which is twice the DDR frequency. The FBDIMM links run at six times the frequency of the DDR data buses.

The OpenSPARC T2 MCU implements a DDR2 FBD design model that is based on various JEDEC-approved DDR2 SDRAM and FBDIMM standards. JEDEC has received information that certain patents or patent applications

may be relevant to FBDIMM Advanced Memory Buffer standard (JESD82-20) as well as other standards related to FBDIMM technology (JESD206) (For more information, see
`http://www.jedec.org/download/search/FBDIMM/Patents.xls).`
Sun Microsystems does not provide any legal opinions as to the validity or relevancy of such patents or patent applications. Sun Microsystems encourages prospective users of the OpenSPARC T2 MCU design to review all information assembled by JEDEC and develop their own independent conclusion.

4.5.4 Noncacheable Unit (NCU)

The NCU performs an address decode on I/O-addressable transactions and directs them to the appropriate block (for example, DMU, CCU). In addition, the NCU maintains the register status for external interrupts.

4.5.5 System Interface Unit (SIU)

The SIU connects the DMU and L2 cache. SIU is the L2 cache access point for the Network subsystem.

4.5.6 SSI ROM Interface (SSI)

OpenSPARC T2 has a 50 Mb/s serial interface (SSI), which connects to an external boot ROM. In addition, the SSI supports PIO accesses across the SSI, thus supporting optional Control and Status registers (CSRs) or other interfaces attached to the SSI.

4.6 Summary of Differences Between OpenSPARC T1 and OpenSPARC T2

OpenSPARC T2 follows the CMT philosophy of OpenSPARC T1, but adds more execution capability to each physical core, as well as significant system-on-a-chip components and an enhanced L2 cache.

4.6.1 Microarchitectural Differences

The following lists the microarchitectural differences.

- Physical core consists of two integer execution pipelines and a single floating-point pipeline. OpenSPARC T1 has a single integer execution pipeline and all cores shared a single floating-point pipeline.
- Each physical core in OpenSPARC T2 supports eight strands, which all share the floating-point pipeline. The eight strands are partitioned into two groups of four strands, each of which shares an integer pipeline. OpenSPARC T1 shares the single integer pipeline among four strands.
- Pipeline in OpenSPARC T2 is eight stages, two stages longer than OpenSPARC T1.
- Instruction cache is 8-way associative, compared to 4-way in OpenSPARC T1.
- The L2 cache is 4-Mbyte, 8-banked and 16-way associative, compared to 3-Mbyte, 4-banked and 12-way associative in OpenSPARC T1.
- Data TLB is 128 entries, compared to 64 entries in OpenSPARC T1.
- The memory interface in OpenSPARC T2 supports fully buffered DIMMS (FBDs), providing higher capacity and memory clock rates.
- The OpenSPARC T2 memory channels support a single-DIMM option for low-cost configurations.
- OpenSPARC T2 includes a network interface unit (NIU), to which network traffic management tasks can be off-loaded.

4.6.2 Instruction Set Architecture (ISA) Differences

There are a number of ISA differences between OpenSPARC T2 and OpenSPARC T1, as follows:

- OpenSPARC T2 fully supports all VIS 2.0 instructions. OpenSPARC T1 supports a subset of VIS 1.0 plus the SIAM (Set Interval Arithmetic Mode) instruction (on OpenSPARC T1, the remainder of VIS 1.0 and 2.0 instructions trap to software for emulation).
- OpenSPARC T2 supports the full CMP specification, as described in *UltraSPARC Architecture 2007*. OpenSPARC T1 has its own version of CMP control/status registers. OpenSPARC T2 consists of eight physical cores, with eight virtual processors per physical core.

- OpenSPARC T2 does not support OpenSPARC T1's `idle` state or its idle, halt, or resume messages. Instead, OpenSPARC T2 supports parking and unparking as specified in the CMP chapter of *UltraSPARC Architecture 2007 Specification*. Note that parking is similar to OpenSPARC T1's `idle` state. OpenSPARC T2 does support an equivalent to the `halt` state, which on OpenSPARC T1 is entered by writing to HPR $1E_{16}$. However, OpenSPARC T2 does not support OpenSPARC T1's STRAND_STS_REG ASR, which holds the strand state. Halted state is not software-visible on OpenSPARC T2.
- OpenSPARC T2 does not support the INT_VEC_DIS register (which allows any OpenSPARC T1 strand to generate an interrupt, reset, idle, or resume message to any strand). Instead, an alias to `ASI_INTR_W` is provided, which allows only the generation of an interrupt to any strand.
- OpenSPARC T2 supports the ALLCLEAN, INVALW, NORMALW, OTHERW, POPC, and FSQRT<s|d> instructions in hardware.
- OpenSPARC T2's floating-point unit generates *fp_unfinished_other* with FSR.ftt unfinished_FPop for most denorm cases and supports a nonstandard mode that flushes denorms to zero. OpenSPARC T1 handles denorms in hardware, never generates an unfinished_FPop, and does not support a nonstandard mode.
- OpenSPARC T2 generates an *illegal_instruction* trap on any quad-precision FP instruction, whereas OpenSPARC T1 generates an *fp_exception_other* trap on numeric and move-FP-quad instructions. See Table 5-2 of the *UltraSPARC T2 Supplement to the "UltraSPARC Architecture 2007 Specification*."
- OpenSPARC T2 generates a *privileged_action* exception upon attempted access to hyperprivileged ASIs by privileged software, whereas, in such cases, OpenSPARC T1 takes a *data_access_exception* exception.
- OpenSPARC T2 supports PSTATE.tct; OpenSPARC T1 does not.
- OpenSPARC T2 implements the SAVE instruction similarly to all previous UltraSPARC processors. OpenSPARC T1 implements a SAVE instruction that updates the locals in the new window to be the same as the locals in the old window, and swaps the *in*s (*out*s) of the old window with the *out*s (*in*s) of the new window.
- PSTATE.am masking details differ between OpenSPARC T1 and OpenSPARC T2, as described in Section 11.1.8 of the *UltraSPARC T2 Supplement to the "UltraSPARC Architecture 2007 Specification*."
- OpenSPARC T2 implements PREFETCH fcn = 18_{16} as a prefetch invalidate cache entry, for efficient software cache flushing.
- The Synchronous Fault register (SFSR) is eliminated in OpenSPARC T2.

- T1's *data_access_exception* is replaced in OpenSPARC T2 by multiple *DAE_** exceptions.
- T1's *instruction_access_exception* exception is replaced in OpenSPARC T2 by multiple *IAE_** exceptions.

4.6.3 MMU Differences

The OpenSPARC T2 and OpenSPARC T1 MMUs differ as follows:

- OpenSPARC T2 has a 128-entry DTLB, whereas OpenSPARC T1 has a 64-entry DTLB.
- OpenSPARC T2 supports a pair of primary context registers and a pair of secondary context registers. OpenSPARC T1 supports a single primary context and single secondary context register.
- OpenSPARC T2 does not support a locked bit in the TLBs. OpenSPARC T1 supports a locked bit in the TLBs.
- OpenSPARC T2 supports only the sun4v (the architected interface between privileged software and hyperprivileged software) TTE format for I/D-TLB Data-In and Data-Access registers. OpenSPARC T1 supports both the sun4v and the older sun4u TTE formats.
- OpenSPARC T2 is compatible with UltraSPARC Architecture 2007 with regard to multiple flavors of data access exception (*DAE_**) and instruction access exception (*IAE_**). As per UltraSPARC Architecture 2005, OpenSPARC T1 uses the single flavor of *data_access_exception* and *instruction_access_exception*, indicating the "flavors" in its SFSR register.
- OpenSPARC T2 supports a hardware Table Walker to accelerate ITLB and DTLB miss handling.
- The number and format of translation storage buffer (TSB) configuration and pointer registers differs between OpenSPARC T1 and OpenSPARC T2. OpenSPARC T2 uses physical addresses for TSB pointers; OpenSPARC T1 uses virtual addresses for TSB pointers.
- OpenSPARC T1 and OpenSPARC T2 support the same four page sizes (8 Kbyte, 64 Kbyte, 4 Mbyte, 256 Mbyte). OpenSPARC T2 supports an *unsupported_page_size* trap when an illegal page size is programmed into TSB registers or attempted to be loaded into the TLB. OpenSPARC T1 forces an illegal page size being programmed into TSB registers to be 256 Mbytes and generates a *data_access_exception* trap when a page with an illegal size is loaded into the TLB.
- OpenSPARC T2 adds a demap real operation, which demaps all pages with r = 1 from the TLB.

- OpenSPARC T2 supports an I-TLB probe ASI.
- Autodemapping of pages in the TLBs only demaps pages of the same size or of a larger size in OpenSPARC T2. In OpenSPARC T1, autodemap demaps pages of the same size, larger size, or smaller size.
- OpenSPARC T2 supports detection of multiple hits in the TLBs.

4.6.4 Performance Instrumentation Differences

Both OpenSPARC T1 and OpenSPARC T2 provide access to hardware performance counters through the PIC and PCR registers. However, the events captured by the hardware differ significantly between OpenSPARC T1 and OpenSPARC T2, with OpenSPARC T2 capturing a much larger set of events, as described in Chapter 10 of the *UltraSPARC T2 Supplement to the "UltraSPARC Architecture 2007 Specification*." OpenSPARC T2 also supports count events in hyperprivileged mode; OpenSPARC T1 does not.

In addition, the implementation of *pic_overflow* differs between OpenSPARC T1 and OpenSPARC T2. OpenSPARC T1 provides a disrupting *pic_overflow* trap on the instruction following the one that caused the overflow event. OpenSPARC T2 provides a disrupting *pic_overflow* on the instruction that generates the event, but that occurs within an epsilon number of event-generating instructions from the actual overflow.

Both OpenSPARC T2 and OpenSPARC T1 support DRAM performance counters.

4.6.5 Error Handling Differences

Error handling differs quite a bit between OpenSPARC T1 and OpenSPARC T2. OpenSPARC T1 primarily employs hardware correction of errors, whereas OpenSPARC T2 primarily employs software correction of errors.

- OpenSPARC T2 uses the following traps for error handling:
  - *data_access_error*
  - *data_access_MMU_error*
  - *hw_corrected_error*
  - *instruction_access_error*
  - *instruction_access_MMU_error*
  - *internal_processor_error*
  - *store_error*
  - *sw_recoverable_error*

OpenSPARC T1 uses the following traps:

- *data_access_error*
- *hw_corrected_error data_error*
- *instruction_access_error*
- *internal_processor_error*

- OpenSPARC T2 integer register file (IRF) and floating-point register file (FRF) ECC errors are handled in software. OpenSPARC T1 corrects single-bit transient errors in hardware.
- OpenSPARC T2 can disable both error reporting and error traps. OpenSPARC T1 can disable only error traps.
- OpenSPARC T2 takes a deferred *store_error* trap on store buffer uncorrectable errors. OpenSPARC T1 does not have error correction on its store buffers.
- OpenSPARC T2 generates a trap on multiple hits in the ITLB, DTLB, I-cache, or D-cache. OpenSPARC T1 simply uses one of the matching entries.
- OpenSPARC T2 protects its MMU register array with parity, taking a trap if an error is detected during a tablewalk. OpenSPARC T1 MMU registers are not protected by parity. OpenSPARC T2 MMU error handling is described in Section 16.7.1, *ITLB Errors*, Section 16.7.2, *DTLB Errors*, and Section 16.7.11, *MMU Register Array (MRAU)* of the *UltraSPARC T2 Supplement to the "UltraSPARC Architecture 2007 Specification*."
- OpenSPARC T2 protects the TICK (TICK, STICK, HSTICK) compare registers, scratchpad registers, and trap stack registers with SECDED ECC, taking a trap if an error is detected while accessing the registers. OpenSPARC T1 leaves these registers unprotected by ECC.
- OpenSPARC T2 supports NotData in the L2 cache (NotData is not supported in memory in either OpenSPARC T1 or OpenSPARC T2).
- OpenSPARC T2 protects the vuad bits by SECDED ECC. OpenSPARC T1 protects the vuad bits by parity.

4.6.6 Power Management Differences

Both OpenSPARC T2 and OpenSPARC T1 support memory access throttling. The mechanisms for supporting CPU throttling differ between OpenSPARC T1 and OpenSPARC T2. OpenSPARC T2 power management is described in Chapter 18 of the *UltraSPARC T2 Supplement to the "UltraSPARC Architecture 2007 Specification*."

4.6.7 Configuration, Diagnostic, and Debug Differences

OpenSPARC T2 configuration and diagnostic support is described in Chapter 28 and debug support is described in Chapter 29 of the *UltraSPARC T2 Supplement to the "UltraSPARC Architecture 2007 Specification*." OpenSPARC T2 additions over OpenSPARC T1 include the following:

- OpenSPARC T2 supports instruction VA watchpoints.
- OpenSPARC T2 supports PA watchpoints.
- OpenSPARC T2 supports the *control_transfer_instruction* trap.
- OpenSPARC T2 implements Prefetch fcn = 18_{16} as a prefetch invalidate cache entry, for efficient software L2 cache flushing. In OpenSPARC T1, flushing of a cache line requires entering "direct-mapped replacement mode," where the L2 LRU is overridden by the address and then forcing out all 12-ways in a set via a displacement with the proper address.
- OpenSPARC T2 supports diagnostic access to the integer register file, store buffers, scratchpad, TICK (TICK, STICK, HSTICK) compare, trap stack, and MMU register arrays.
- OpenSPARC T2 does not require the diagnostic virtual address to match a valid tag for `ASI_DCACHE_DATA`.

CHAPTER **5**

OpenSPARC T2 Memory Subsystem — A Deeper Look

Much of the material in this chapter appeared in UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007, *Part number 950-9556, Revision Draft 1.5, 03, Apr 2008.*

Each SPARC physical core has a 16-Kbyte, 8-way associative instruction cache (32-byte lines), 8 Kbytes, 4-way associative data cache (16-byte lines), 64-entry fully associative instruction Translation Lookaside Buffer (TLB), and 128-entry fully associative data TLBs, each of which is shared by the eight strands. The eight SPARC physical cores are connected through a crossbar to an on-chip unified 4 Mbyte, 16-way associative L2 cache (64-byte lines). The L2 cache is banked eight ways to provide sufficient bandwidth for the eight SPARC physical cores. The L2 cache connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels.

The chapter contains these sections:

- *Caches* on page 44
- *Memory Controller Unit (MCU)* on page 47
- *Memory Management Unit (MMU)* on page 50
- *Noncacheable Unit (NCU)* on page 55
- *System Interface Unit (SIU)* on page 55
- *Data Management Unit (DMU)* on page 56
- *Memory Models* on page 56

A block diagram of OpenSPARC T2 is shown in FIGURE 4-4 on page 34.

5.1 Caches

This section describes the L1 instruction cache and the data cache, followed by details of the combined instruction/data L2 cache.

5.1.1 L1 I-Cache

The L1 instruction cache is 16 Kbytes, physically tagged and indexed, with 32-byte lines, and 8-way associative with random replacement. The I-cache direct-mapped mode works by forcing all replacements to the "way" identified by bits 13:11 of the virtual address. Since lines already present are not affected but only new lines brought into the cache are affected, it is safe to turn on (or off) the direct-mapped mode at any time. Clearing the I-cache enable bit stops all accesses to the I-cache for that strand. All fetches will miss, and the returned data will not fill the I-cache. Invalidates are still serviced while the I-cache is disabled.

5.1.2 L1 D-Cache

The L1 data cache is 8 Kbytes, write-through, physically tagged and indexed, with 16-byte lines, and 4-way associative with true least-recently-used (LRU) replacement. The D-cache replacement algorithm is true LRU. Six bits are maintained for each cache index. In direct-mapped mode, the D-cache works by changing the replacement algorithm from LRU to instead use two bits of index (address 12:11) to select the way. Since lines already present are not affected but only new lines brought into the cache are affected, it is safe to turn on (or off) the direct-mapped mode at any time.

The D-cache may be disabled and when it is disabled, accesses to the D-cache behave as follows. A load that hits in the D-cache ignores the cached data and fetches the data from L2. A load that misses in the cache fetches the data from L2 but does not allocate the line in the data cache. Stores that miss in the data cache never allocate in the data cache (as normal). Stores that hit in the data cache are performed in the L2, then update the data cache (as normal). Even if the D-cache is disabled, L2 still keeps the D-cache coherent. Invalidations caused by L2 replacements, stores from other cores, or direct memory access (DMA) stores from I/O activity that hit data in the D-cache cause those lines to be invalidated. For the D-cache to be fully disabled, a certain `dc` bit must be off on all strands in the virtual processor, and the D-cache must be flushed in a way that doesn't bring new lines back in. This can be done by storing (from a different core) to each line that is in the D-cache or by displacement flushing the L2 cache so that inclusion will force all D-cache lines to be invalidated.

5.1.3 L2 Cache

The L2 combined instruction/data cache is 4 Mbytes, write-back, physically tagged and indexed, with 64-byte lines, 8-banked, and 16-way associative with pseudo-LRU replacement. The L2 cache is banked eight ways. To provide for better partial-die recovery, OpenSPARC T2 can also be configured in 4-bank and 2-bank modes (with 1/2 and 1/4 the total cache size, respectively). The cache is 4 Mbytes and 16-way set associative with a pseudo-LRU replacement (replacement is based on a used-bit scheme). The line size is 64 bytes. Unloaded access time is 26 cycles for an L1 data cache miss and 24 cycles for an L1 instruction cache miss.

A used-bit scheme is used to implement a not recently used (NRU) replacement. The used bit is set each time a cache line is accessed or when initially fetched from memory. If setting the used bit causes all used bits (at an index) to be set, the remaining used bits are cleared instead. In addition, each line has an allocate bit (a), which is set while a line is in a multicycle operation. This operation can be a cache fill, in which case the a bit gets set when the location is allocated and the a bit gets cleared when the location is filled with memory data. Alternatively, this could be a multipass operation, either an atomic operation or a subword store (which requires read-modify-write) whereby the a bit is set on the first pass and cleared on the second or final pass. Any line that has the a bit set is ineligible for replacement.

Each L2 bank has a single rotating replacement pointer, which is the "starting point" to find the way to replace. On a miss, the L2 looks for the first line at that index with both u bit and a bit clear, starting with the way pointed at by the replacement pointer. If all lines have the u bit or a bit set, all u bits are cleared and the scan repeated. The replacement pointer is then rotated forward one way. Since the replacement pointer is used by all sets of the L2, replacement is somewhat more random than if each set or index had its own replacement pointer. The replacement pointer is incremented on any L2 miss that causes a cache fill (that is, not DMA reads or full-line DMA writes). The replacement pointer is only reset (put to a known state) by power-on reset (POR), warm reset, or debug reset. Valid bits do not affect the NRU replacement.

The L2 cache has a directory of all L1 lines, both I-cache and D-cache, implemented as duplicate tags. Thus, the L2 always knows exactly which lines are in which L1 caches, and in which way of each cache. When the L1 requests a line from the L2, the virtual processor specifies whether the line will be allocated (put into the cache), and which way it will go into. The L2-to-virtual processor (CPX) protocol allows the L2 to issue invalidates to any or all of the cores simultaneously, but only a single invalidation to each core. For this reason, for a given virtual processor, an L1 line is only allowed to be in either the I-cache or the D-cache, but not both. The invalidate transaction includes only index, way, and L1 cache (I or D); it does not include the address.

Since the L2 tracks which lines are in which L1 ways, just invalidating an L1 line with the address space identifier (ASI) mechanism is not safe and can lead to stale data problems and data corruption. The problem occurs if a line is marked invalid and a subsequent access to the L1 cache refetches the line, but into a different way.

At this time, the L2 directory has the same line in two places in its directory. Later, when the L2 wants to invalidate that address, it gets a double hit on its CAM access, which the logic does not support. (To invalidate an L1 line, inject an error into its tag and then access it. The hardware error handling will invalidate the line and inform the L2 directory.)

The L2 cache direct-mapped mode works by changing the replacement algorithm from NRU to instead use four bits of index (address 21:18) to select the way. Since lines already present are not affected but only new lines brought into the cache are affected, it is safe to turn on (or off) the direct-mapped mode at any time.

The L2 cache disable actually disables an L2 bank. Thus, it is recommended that the L2 be flushed first so that modified lines are written back to memory. While an L2 bank is disabled, the cache effectively has only a single line, which is invalidated or written back at the end of the access. Thus, a store will miss to memory, perform the write into the one-line cache, then flush. Then, the next cache access can be started. The L2 directory is not used while the L2 cache is disabled. Thus, all L1 caches must be disabled and emptied before any (or all) L2 banks are disabled.

5.2 Memory Controller Unit (MCU)

OpenSPARC T2 has four MCUs, one for each memory branch with a pair of L2 banks interacting with exactly one DRAM branch. The branches are interleaved and support 1–16 DDR2 DIMMs. Each memory branch is two FBD channels wide. A branch may use only one of the FBD channels in a reduced power configuration. Each DRAM branch operates independently and can have a different memory size and a different kind of DIMM (for example, a different number of ranks or a different CAS latency). Software should not use address space larger than four times the lowest memory capacity in a branch because the cache lines are interleaved across branches. The DRAM controller frequency is the same as that of the double data rate (DDR) data buses, which is twice the DDR frequency. The FBD links run at six times the frequency of the DDR data buses.

OpenSPARC T2 interfaces to external registered DDR2 fully buffered DIMMs (FBDs) through unidirectional high-speed links. OpenSPARC T2 interfaces directly to external registered DDR2 DIMMs. There are four memory branches on OpenSPARC T2. Each memory branch services 64-byte read and write requests from two L2 cache banks of the on-chip L2 cache unit. The features of the OpenSPARC T2 memory controller are as follows:

- Uses 10-bit southbound and 14-bit northbound FBD channel protocols running at 12 times the SDRAM cycle rate
- Supports 256-Mbit DRAM components for x4 data width; supports 512-Mbit, 1-Gbit, and 2-Gbit DRAM components for x4 and x8 data widths
- Maximum memory of 128 Gbytes per branch using sixteen 8-Gbyte DDR2 FBDs
- Supports up to 16 ranks of DDR2 DIMMs per branch (8 pairs of double-sided FBDs)
- Supports registered DDR2 DIMMs of clock frequency up to 400 MHz
- Supports 128 bits of write data and 16 bits ECC per SDRAM cycle and 256 bits of read data and 32 bits ECC per SDRAM cycle
- Supports DDR2 SDRAM burst length of 4 when using both FBD channels in a branch, burst length of 8 when using a single channel per branch
- ECC generation, check, correction, and extended ECC
- Programmable DDR2 SDRAM power throttle control

- System peak memory bandwidth (4 branches): 50 Gbytes/s for reads, 25 Gbytes/s for writes

The MCU employs the following design requirements:

- x4 and x8 DRAM parts are supported. Extended ECC is not supported for x8 DRAM parts.
- DIMM capacity, configuration, and timing parameters cannot be different within a memory branch.
- DRAM banks are always closed after read or write command by issuing an autoprecharge command.
- Burst length is 4 (bl = 4) when two channels per DDR branch are used. Burst length is 8 (bl = 8) when a single channel per branch is used.
- There is a fixed 1 dead cycle for switching commands from one rank on a DIMM to the other rank on the same DIMM.
- Reads, writes, and refreshes across DDR branches have no relationship to each other. They are all independent.

Each CPU chip has four independent DDR branches, each controlled by a separate MCU. Each branch can be configured with one or two channels and supports up to 16 ranks of DIMMs. Each channel can be populated with up to eight single- or dual-rank FBDs. When a branch is configured with two channels, the two FBDs that share the same advanced memory buffer (AMB) ID are accessed in lockstep. Data is returned 144 bits per frame for 8 frames in single-channel mode and 288 bits per frame for 4 frames in dual-channel mode. In either mode, the total data transfer size is 512 bits, or 64 bytes, the cache line size for the L2 cache.

Each FBD contains four or eight internal banks that can be controlled independently. These internal banks are controlled inside the SDRAM chips themselves. Accesses can overlap between different internal banks. In a normal configuration, every read and write operation to SDRAM will generate a burst length of 4 with 16 bytes of data transferred every half memory clock cycle. In single-channel mode, reads and writes will have a burst length of 8 with 8 bytes of data transferred every half memory cycle.

FIGURE 5-1 illustrates the DDR branch configuration.

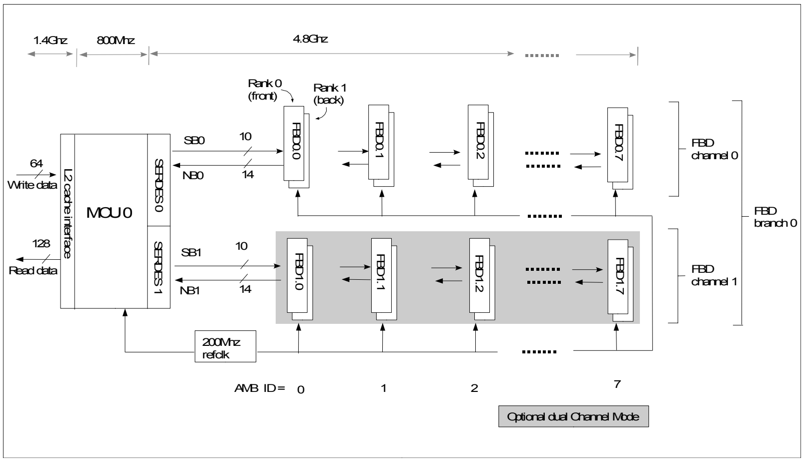


FIGURE 5-1 DDR Branch Configuration

The FBD specification supports two southbound channel configurations and five northbound channel configurations. OpenSPARC T2 supports both southbound configurations—the 10-bit and 10-bit failover modes—and two of the northbound configurations, the 14-bit and 14-bit failover modes. These modes support data packets of 64-bit data and 8-bit ECC. The 10-bit southbound mode provides 22 bits of CRC, and the 10-bit failover mode has 10 bits of CRC. The 14-bit northbound mode provides 24 bits of CRC on read data (12 bits per 72-bit data packet); the 14-bit failover mode provides 12 bits of CRC (6 bits per 72-bit data packet). During channel initialization, software determines if a channel can be fully utilized (10-bit southbound or 14-bit northbound mode) or if a failover mode must be used in one of the bit lanes that is muxed out.

The power used by the SDRAMs is a significant portion of the system power usage. Some high-performance systems may be able to handle the maximum power consumption rates, but low-cost systems may need to limit their power usage because of cooling issues, etc. The power-throttling scheme of OpenSPARC T2 limits the number of SDRAM memory access transactions during a specified time period by counting the number of banks that are opened (that is, active cycles) during this time. Since all of the write and read transactions of OpenSPARC T2 use autoprecharge, the number of banks opened is equivalent to the number of write and read transactions. If the number of transactions during this time period exceeds a preprogrammed limit, no more memory transactions are dispatched until the time period expires.

5.3 Memory Management Unit (MMU)

The OpenSPARC T2 MMU consists of two Translation Lookaside Buffers (TLBs), one Instruction TLB (ITLB), and one Data TLB (DTLB). Each TLB consists of many Translation Table Entries (TTEs) which provide first-level translation for instruction and data accesses. The TLBs are accessed in parallel with the caches and the tags. The MMU is supported by other memory resident software structures called translation storage buffer (TSB) and software translation table. The following subsections describe the MMU in greater detail:

- *Address Translation Overview* on page 50
- *TLB Miss Handling* on page 51
- *Instruction Fetching* on page 52
- *Hypervisor Support* on page 53
- *MMU Operations* on page 54

5.3.1 Address Translation Overview

FIGURE 5-2 illustrates the concepts and structures discussed for address translation.

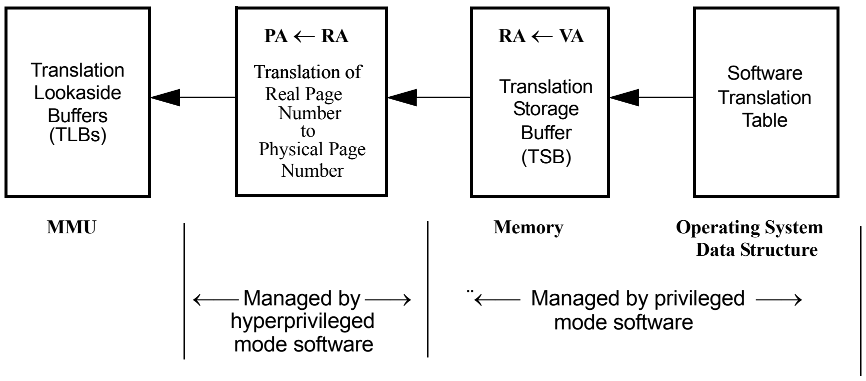

Translation Lookaside Buffers (TLBs)
PA ← RA
Translation of Real Page Number to Physical Page Number
RA ← VA
Translation Storage Buffer (TSB)
Software Translation Table
MMU
Memory
Operating System Data Structure
Managed by hyperprivileged mode software
Managed by privileged mode software

FIGURE 5-2 Concepts of Address Translation

Before details of address translation are discussed, consider the structures illustrated in FIGURE 5-2. From right to left:

- **Software translation table** — An arbitrary data structure in which privileged software maintains translation information. It, in conjunction with the translation storage buffer, will quickly reload the TLB in the event of a TLB miss. The software translation table is likely to be large and complex.
- **TSB** — The interface between the software translation table and the underlying memory management hardware. The TSB is an array of translation table entries (TTEs) and serves as a cache of the software translation table. A TSB is arranged as a direct-mapped cache of TTEs.
- **TLB** — An MMU entity that acts as an independent cache of the software translation table, providing appropriate concurrency for virtual-to-physical address translation. The TLBs are small and fast.

UltraSPARC T2 TLBs store physical addresses. Privileged code manages the VA-to-RA translations, while hyperprivileged code manages the RA-to-PA translations. The TLBs contain VA-to-PA translations or RA-to-PA translations (the latter are distinguished from the former by a real bit in the TLB).

The TLB receives a virtual address or real address, a partition identifier, and context identifier as input and produces a physical address and page attributes as output in case of a TLB hit. A TLB miss results in an access to the TSB. The UltraSPARC T2 MMU provides hardware tablewalk support and precomputed pointers into the TSB(s) for both zero and nonzero contexts.

5.3.2 TLB Miss Handling

A TLB miss results in an access to the TSB also known as hardware tablewalk. Note that hardware tablewalk can be disabled and a full software implementation for TLB miss handling can be used. The TSB exists as a normal data structure in memory and therefore may be cached. Indeed, the speed of the TLB miss handler relies on the TSB accesses hitting the level-2 cache at a substantial rate. This policy may result in some conflicts with normal instruction and data accesses, but the dynamic sharing of the L2 cache resource should provide a better overall solution than that provided by a fixed partitioning.

A TLB miss is handled as follows:

- With hardware tablewalk implemented and enabled, hardware processes the TLB miss directly.
- With the hardware tablewalk unimplemented or disabled, the MMU immediately traps to hyperprivileged software for TLB miss processing.

The following is the TLB miss and reload sequence when hardware tablewalk is enabled:

- Hardware tablewalk uses the TSB Configuration registers and the virtual address (VA) of the access to calculate the physical address of the TSB TTE to examine. The TSB Configuration register provides the base address of the TSB as well as the number of TTEs in the TSB and the size of the pages translated by the TTEs. Hardware tablewalk uses the page size from the TSB Configuration register to calculate the presumed virtual page number (VPN) for the given VA. Hardware tablewalk then uses the number of TTE entries and the presumed VPN to generate an index into the TSB. This index is concatenated with the upper bits of the base address to generate the TTE address.
- Hardware tablewalk compares the VPN and context ID of the request to that from the TTE, masking the VPN based on the page size in the TTE. If the VPN and context ID match, hardware tablewalk returns the TTE with the real page number (RPN) translated into a physical page number (PPN). Hardware tablewalk checks the TTE from each enabled TSB until it either finds a match or has searched all enabled TSBs.
- If none of the TTE entries from the enabled TSBs match on page size, VPN, and context ID, hardware generates an *instruction_access_MMU_miss* or *data_access_MMU_miss* trap.

5.3.3 Instruction Fetching

OpenSPARC T2 speculatively fetches instructions. Under certain conditions, this can cause the memory controller to receive an unsupported physical address. Consider the following instruction sequence which exits hyperprivileged mode and returns to user or privileged mode (executed with HPSTATE.hpriv initially set to 1):

```
jmpl %g3 + %g0, %g0
wrhpr %g0, %g0, %hpstate
```

This will cause the IMMU to go from bypass (during which VA{39:0} is passed directly to PA{39:0}) into either RA → PA or VA → PA translation. However, since the fetch of the target of the `jmpl` is fetched speculatively, the memory controller may see VA{39:0} of the target of the `jmpl` as a physical address. This address may not be supported, in which case a disrupting *software_recoverable_error* trap could result, even though no real error has occurred. To avoid this disrupting trap, Hypervisor should avoid changing translation in the delay slot of delayed control-transfer instructions. For example, the sequence above could be replaced with the following code:

```
mov %tl, %g5
add %g5, 1, %g5
mov %g5, %tl
mov %g3, %tnpc
mov 0, %hpstate
done
```

Although the example refers to changes in HPSTATE, any instruction that can potentially change translation should avoid being placed in the delay slot of delayed control-transfer instructions. These include writes to PSTATE, I-/D-TLB Data In and Data Access registers, I-/D-MMU Demap registers, and the `ASI_LSU_CONTROL_REG` register.

OpenSPARC T2 fetches instructions sequentially (including delay slots). OpenSPARC T2 fetches delay slots before the branch is resolved (before whether the delay slot will be annulled is known). OpenSPARC T2 also fetches the target of a delayed control-transfer instruction (DCTI) before the delay slot executes. For both these cases, OpenSPARC T2 may fetch from a nonexistent PA (in the case of a fetch from memory) or from an I/O address with side effects. Hypervisor should protect against this for virtual- and real-to-physical translations by maintaining valid mappings of sequential and target addresses at all times. Hypervisor should protect against this for bypassing translations by ensuring that all sequential and target addresses are backed by memory.

5.3.4 Hypervisor Support

To support Hypervisor, a number of additions to the MMU are included:

- A 3-bit partition ID (PID) field is included in each TLB entry to allow multiple guest OSes to share the MMU. This field is loaded with the value of the Partition Identifier register when a TLB entry is loaded. In addition, the PID entry of a TLB is compared with the Partition Identifier register to determine if a TLB hit occurs.
- The MMU is designed to support both virtual-to-physical and real-to-physical translations, using a single r (real translation) bit included in the TLB entry. This field is loaded with bit 10 from the VA used by the store to the I-/D-TLB Data In register or the I-/D-TLB Data Access register. The real bit distinguishes between VA → PA translations (r = 0) and RA → PA translations (r = 1).

  If the real bit is 1, the context ID is ignored when determining a TLB hit. TLB misses on real-to-physical translations generate a *data_real_translation_miss* or *inst_real_translation_miss* trap instead of *fast_data_access_MMU_miss* or *fast_instruction_access_MMU_miss* traps, respectively.

- The translation operation performed depends on the state of HPSTATE.hpriv, PSTATE.priv, the MMU enables, and PSTATE.red (for IMMU).

5.3.5 MMU Operations

This section describes additional operations performed by the OpenSPARC T2 MMU.

5.3.5.1 TLB Operation Summary

The TLB supports exactly one of the following operations per clock cycle:

- **Translation.** The TLB receives a virtual address or real address, a partition identifier, and context identifier as input and produces a physical address and page attributes as output.
- **Demap operation.** The TLB receives a virtual address and a context identifier as input and sets the valid bit to zero for any entry matching the demap page or demap context criteria. This operation produces no output.
- **Read operation.** The TLB reads either the CAM or RAM portion of the specified entry. (Since the TLB entry is greater than 64 bits, the CAM and RAM portions must be returned in separate reads.)
- **Write operation.** The TLB simultaneously writes the CAM and RAM portion of the specified entry, or the entry given by the replacement policy.
- **No operation.** The TLB performs no operation.

5.3.5.2 Demap Operations

Demap is an MMU operation whose purpose is to remove zero, one, or more entries in the TLB. Four types of demap operations are provided:

- **Demap page.** Removes zero or one TLB entry that matches exactly the specified virtual page number and real bit.
- **Demap context.** Removes zero, one, or many TLB entries that match the specified context identifier and have the real bit cleared; never demaps a real translation (r = 1).
- **Demap all.** Removes all pages, regardless of their context or real bit.
- **Demap all pages.** Removes all pages that either have their real bit set (if the r bit in the demap address is set) or have their real bit clear (if the r bit in the demap address is clear), regardless of their context.

All demap operations demap only those pages whose PID matches the PID specified in the Partition Identifier register. A demap operation does not invalidate the TSB in memory. It is the responsibility of the software to modify the appropriate TTEs in the TSB before initiating any demap operation. The demap operation produces no output.

5.4 Noncacheable Unit (NCU)

The NCU performs an address decode on I/O-addressable transactions and directs them to the appropriate block (for example, network interface unit (NIU), DMU, clock control unit (CCU)). In addition, the NCU maintains the register status for external interrupts.

The main functions of the NCU are to route PIO accesses from the CMP virtual processors to the I/O subsystem and to vector interrupts from the I/O subsystem to the CMP virtual processors. The NCU provides CSRs for NCU management, configuration of the PCI-Express (PCIE) address space, and mondo interrupt management. The NCU decodes the I/O physical address space.

OpenSPARC T2 supports 40-bit physical addresses, where the MSB (bit 39) is 0 for cacheable accesses (memory system) and 1 for noncacheable accesses (I/O subsystem). The NCU determines the destination of a PIO access by examining the 8 MSB (bit 39:32) of the physical address. All accesses received by the NCU have bit 39 of the physical address set to 1.

The NCU provides a unique serial number for each OpenSPARC T2 chip. In addition, the NCU contains registers showing the eFuse, SPARC core, and L2 bank status.

5.5 System Interface Unit (SIU)

The SIU connects the NIU, DMU, and L2 cache. SIU is the L2 cache access point for the Network and PCI-Express subsystems. The SIU-L2 cache interface is also the ordering point for PCI-Express ordering rule.

5.6 Data Management Unit (DMU)

The DMU manages Transaction Layer Packet (TLP) to and from the PCI-Express unit (PEU) and maintains the same ordering as from the PEU and then to the SIU. For maintaining ordering between PEU and SIU, the DMU requires the policy that has PIO reads pulling DMA writes to completion. When the PEU issues complete TLP transactions to the DMU, the DMU segments the TLP packet into multiple cache-line-oriented SIU commands and issues them to the SIU. The DMU also queues the response cache lines from SIU and reassembles the multiple cache lines into one TLP packet with maximal payload size. Furthermore, the DMU accepts and queues the PIO transactions requests from NCU and coordinates with the appropriate destination to which the address and data will be sent.

The DMU encapsulates the functions necessary to resolve a virtual PCI-Express packet address into an L2 cache line physical address that can be presented on the SIU interface. The DMU also encapsulates the functions necessary to interpret PCI-Express message signaled interrupts, emulated INTX interrupts. The DMU also provides the functions to post interrupt events to queues managed by software in main memory and generates the Solaris Interrupt Mondo to notify software.

The DMU decodes INTACK and INTNACK from interrupt targets and conveys the information to the interrupt function so that it can move on to service the next interrupt if any (for INTACK) or replay the current interrupt (for INTNACK).

5.7 Memory Models

SPARC V9 defines the semantics of memory operations for three memory models. From strongest to weakest, they are Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Order (RMO). The differences in these models lie in the freedom an implementation is allowed in order to obtain higher performance during program execution. The purpose of the memory models is to specify any constraints placed on the ordering of memory operations in uniprocessor and shared-memory multiprocessor environments. OpenSPARC T2 supports only TSO, with the exception that certain ASI accesses (such as block loads and stores) can operate under RMO. Although a program written for a weaker memory model potentially benefits from higher execution rates, it may require explicit memory synchronization instructions to function correctly if data is shared. MEMBAR is a SPARC V9 memory synchronization primitive that enables a programmer to explicitly control the ordering in a sequence of memory operations. Processor consistency is guaranteed in all memory models.

The current memory model is indicated in the PSTATE.mm field. It is unaffected by normal traps but is set to TSO (PSTATE.mm = 0) when the virtual processor enters `RED_state`. OpenSPARC T2 ignores the value set in this field and always operates under TSO. A memory location is identified by an 8-bit ASI and a 64-bit virtual address. The 8-bit ASI may be obtained from an ASI register or included in a memory access instruction. The ASI is used to distinguish between and provide an attribute for different 64-bit address spaces. For example, the ASI is used by the OpenSPARC T2 MMU and memory access hardware to control virtual-to-physical address translations, access to implementation-dependent control and data registers, and for access protection. Attempts by nonprivileged software (PSTATE.priv = 0) to access restricted ASIs (ASI{7} = 0) cause a *privileged_action* trap.

Memory is logically divided into real memory (cached) and I/O memory (noncached with and without side effects) spaces, based on bit 39 of the physical address (0 for real memory, 1 for I/O memory). Real memory spaces can be accessed without side effects. For example, a read from real memory space returns the information most recently written. In addition, an access to real memory space does not result in program-visible side effects. In contrast, a read from I/O space may not return the most recently written information and may result in program-visible side effects.

5.8 Memory Transactions

In this section, the following memory interactions are addressed:

- *Cache Flushing* on page 58
- *Displacement Flushing* on page 58
- *Memory Accesses and Cacheability* on page 59
- *Cacheable Accesses* on page 59
- *Noncacheable and Side-Effect Accesses* on page 60
- *Global Visibility and Memory Ordering* on page 60
- *Memory Synchronization: MEMBAR and FLUSH* on page 61
- *Atomic Operations* on page 62
- *Nonfaulting Load* on page 63

5.8.1 Cache Flushing

Data in the level-1 (read-only or write-through) caches can be flushed by invalidation of the entry in the cache (in a way that also leaves the L2 directory in a consistent state). Modified data in the level-2 (write-back) cache must be written back to memory when flushed.

Cache flushing is required in these cases for the following caches:

- I-cache: Flush is needed before executing code that is modified by a local store instruction. This is done with the FLUSH instruction. Flushing the I-cache with ASI accesses does not work, because it leaves the I-cache and the L2 directory inconsistent, thus breaking coherency and leading to the possibility of data corruption.
- D-cache: Flush is needed when a physical page is changed from (physically) cacheable to (physically) noncacheable. This is done with a displacement flush (see “Displacement Flushing,” below).
- L2 cache: Flush is needed for stable storage. Examples of stable storage include battery-backed memory and transaction logs. The recommended way to perform this is with the PrefetchICE instruction. Alternatively, this can be done by a displacement flush. Flushing the L2 cache flushes the corresponding blocks from the I- and D-caches, because OpenSPARC T2 maintains inclusion between the L2 and L1 caches.

5.8.2 Displacement Flushing

Cache flushing of the L2 cache or the D-cache can be accomplished by a displacement flush. One does this by placing the cache in direct-map mode and reading a range of read-only addresses that map to the corresponding cache line being flushed, forcing out modified entries in the local cache. Care must be taken to ensure that the range of read-only addresses is mapped in the MMU before a displacement flush is started; otherwise, the TLB miss handler may put new data into the caches. In addition, the range of addresses used to force lines out of the cache must not be present in the cache when the displacement flush is started. (If any of the displacing lines are present before the displacement flush is started, fetching the already present line will not cause the proper way in the direct-mapped mode L2 to be loaded; instead, the already present line will stay at its current location in the cache.)

5.8.3 Memory Accesses and Cacheability

In OpenSPARC T2, all memory accesses are cached in the L2 cache (as long as the L2 cache is enabled). The cp bit in the TTE corresponding to the access controls whether the memory access will be cached in the primary caches (if cp = 1, the access is cached in the primary caches; if cp = 0 the access is not cached in the primary caches). Atomic operations are always performed at the L2 cache. Note that diagnostic accesses to the L2 cache can be used to invalidate a line, but they are not an alternative to PrefetchICE or displacement flushing. L2 diagnostic accesses do not cause invalidation of L1 lines (breaking L1 inclusion), and modified data in the L2 cache will not be written back to memory using these ASI accesses.

Two types of memory operations are supported in OpenSPARC T2: cacheable and noncacheable accesses, as indicated by the page translation. Cacheable accesses are inside the coherence domain; noncacheable accesses are outside the coherence domain. SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses. OpenSPARC T2 maintains TSO ordering, regardless of the cacheability of the accesses, relative to other access by processors. See *The SPARC Architecture Manual-Version 9* for more information about the SPARC V9 memory models. On OpenSPARC T2, a MEMBAR `#Lookaside` is effectively a NOP and is not needed for forcing order of stores vs. loads to noncacheable addresses.

5.8.4 Cacheable Accesses

Accesses that fall within the coherence domain are called cacheable accesses. They are implemented in OpenSPARC T2 with the following properties:

- Data resides in real memory locations.
- The cacheable accesses observe the supported cache coherence protocol.
- The unit of coherence is 64 bytes at the system level (coherence between the virtual processors and I/O), enforced by the L2 cache.
- The unit of coherence for the primary caches (coherence between multiple virtual processors) is the primary cache line size (16 bytes for the data cache, 32 bytes for the instruction cache), enforced by the L2 cache directories.

5.8.5 Noncacheable and Side-Effect Accesses

Accesses that are outside the coherence domain are called noncacheable accesses. Accesses of some of these memory (or memory-mapped) locations may result in side effects. Noncacheable accesses are implemented in OpenSPARC T2 with the following properties:

- Data may or may not reside in real memory locations.
- Accesses may result in program-visible side effects; for example, memory-mapped I/O control registers in a UART may change state when read.
- Accesses may not observe supported cache coherence protocol.
- The smallest unit in each transaction is a single byte.

Noncacheable accesses are all strongly ordered with respect to other noncacheable accesses (regardless of the e bit). Speculative loads with the e bit set cause a *DAE_so_page* trap.

5.8.6 Global Visibility and Memory Ordering

To ensure the correct ordering between the cacheable and noncacheable domains, explicit memory synchronization is needed in the form of MEMBARs or atomic instructions. CODE EXAMPLE 5-1 illustrates the issues involved in mixing cacheable and noncacheable accesses.

Assume that all accesses go to non-side-effect memory locations.

CODE EXAMPLE 5-1 Mixing Cacheable and Noncacheable Accesses

```
Process A:
while (true)
{
        Store D1                              // data produced
    1   MEMBAR #StoreStore                    // needed in PSO, RMO
        Store 1 to F1                         // set flag
        while (F1 ≠ 0)                        // spin while flag is set
        {
            Load F1
        }
    2   MEMBAR #LoadLoad | #LoadStore         // needed in RMO

        Load D2
```

CODE EXAMPLE 5-1 Mixing Cacheable and Noncacheable Accesses *(Continued)*

```
}

Process B:
while (true)
{
        while (F1 = 0)                          // spin while flag is clear
        {
            Load F1
        }
    2   MEMBAR #LoadLoad | #LoadStore           // needed in RMO

        Load D1                                 // data consumed

        Store D2

    1   MEMBAR #StoreStore                      // needed in PSO, RMO

        Store 0 to F1                           // clear flag
}
```

Due to load and store buffers implemented in OpenSPARC T2, the above code may not work for RMO accesses without the MEMBARs shown in the program segment. Under TSO, loads and stores (except block stores) cannot pass earlier loads, and stores cannot pass earlier stores; therefore, no MEMBAR is needed. Under RMO, there is no implicit ordering between memory accesses; therefore, the MEMBARs at both #1 and #2 are needed.

5.8.7 Memory Synchronization: MEMBAR and FLUSH

The MEMBAR (STBAR in SPARC V8) and FLUSH instructions provide for explicit control of memory ordering in program execution. MEMBAR has several variations; their implementations in OpenSPARC T2 are described below. See the references to “Memory Barrier,” “The MEMBAR Instruction,” and “Programming With the Memory Models” in *The SPARC Architecture Manual-Version 9* for more information.

- MEMBAR `#LoadLoad` — All loads on OpenSPARC T2 switch a strand out until the load completes. Thus, MEMBAR `#LoadLoad` is treated as a NOP on OpenSPARC T2.
- MEMBAR `#StoreLoad` — Forces all loads after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility. MEMBAR `#StoreLoad` behaves the same as MEMBAR `#Sync` on OpenSPARC T2.
- MEMBAR `#LoadStore` — All loads on OpenSPARC T2 switch a strand out until the load completes. Thus, MEMBAR `#LoadStore` is treated as a NOP on OpenSPARC T2.
- MEMBAR `#StoreStore` and STBAR — Stores on OpenSPARC T2 maintain order in the store buffer. Thus, MEMBAR `#StoreStore` is treated as a NOP on OpenSPARC T2.
- MEMBAR `#Lookaside` — Loads and stores to noncacheable addresses are self-synchronizing on OpenSPARC T2. Thus, MEMBAR `#Lookaside` is treated as a NOP on OpenSPARC T2.
- MEMBAR `#MemIssue` — Forces all outstanding memory accesses to be completed before any memory access instruction after the MEMBAR is issued. It must be used to guarantee ordering of cacheable accesses following noncacheable accesses.
- MEMBAR `#Sync` (Issue Barrier) — Forces all outstanding instructions and all deferred errors to be completed before any instructions after the MEMBAR are issued.
- Self-modifying code (FLUSH) — The SPARC V9 instruction set architecture does not guarantee consistency between code and data spaces. A problem arises when code space is dynamically modified by a program writing to memory locations containing instructions. Dynamic optimizers, LISP programs, dynamic linking, and some types of debuggers require this behavior. SPARC V9 provides the FLUSH instruction to synchronize instruction and data memory after code space has been modified. In OpenSPARC T2, FLUSH behaves like a store instruction for the purpose of memory ordering. In addition, all instruction fetch (or prefetch) buffers are invalidated. The issue of the FLUSH instruction is delayed until previous (cacheable) stores are completed. Instruction fetch (or prefetch) resumes at the instruction immediately after the FLUSH.

5.8.8 Atomic Operations

SPARC V9 provides three atomic instructions to support mutual exclusion. These instructions behave like both a load and a store, but the operations are carried out indivisibly. Atomic instructions may be used only in the cacheable domain. An atomic access with a restricted ASI in unprivileged mode (PSTATE.priv = 0) causes a *privileged_action* trap. An atomic access with a noncacheable address causes a *DAE_nc_page* trap. An atomic access with an unsupported ASI causes a *DAE_invalid_ASI* trap.

- SWAP instruction — SWAP atomically exchanges the lower 32 bits in an integer register with a word in memory. This instruction is issued only after store buffers are empty. Subsequent loads interlock on earlier SWAPs.
- LDSTUB instruction — LDSTUB behaves like SWAP, except that it loads a byte from memory into an integer register and atomically writes all 1's (FF_{16}) into the addressed byte.
- Compare and Swap (CASX) instruction — Compare-and-swap combines a load, compare, and store into a single atomic instruction. It compares the value in an integer register to a value in memory; if they are equal, the value in memory is swapped with the contents of a second integer register. All of these operations are carried out atomically; in other words, no other memory operation may be applied to the addressed memory location until the entire compare-and-swap sequence is completed.

5.8.9 Nonfaulting Load

A nonfaulting load behaves like a normal load except that it does not allow side-effect access. An access with the e bit set causes a *DAE_side_effect_page* trap. It can also be applied to a page with the nfo (nonfault access only) bit set; other types of accesses cause a *DAE_nfo_page* trap.

When a nonfaulting load encounters a TLB miss, the operating system should attempt to translate the page. If the translation results in an error (for example, address out of range), a 0 is returned and the load completes silently.

Typically, optimizers use nonfaulting loads to move loads before conditional control structures that guard their use. This technique potentially increases the distance between a load of data and the first use of that data, to hide latency; it allows for more flexibility in code scheduling. It also allows for improved performance in certain algorithms by removing address checking from the critical code path. For example, when following a linked list, nonfaulting loads allow the null pointer to be accessed safely in a read-ahead fashion if the operating system can ensure that the page at virtual address 0_{16} is accessed with no penalty.

The nfo bit in the MMU marks pages that are mapped for safe access by nonfaulting loads but can still cause a trap by other, normal, accesses. This allows programmers to trap on wild pointer references (many programmers count on an exception being generated when accessing address 0_{16} to debug code) while benefiting from the acceleration of nonfaulting access in debugged library routines.

CHAPTER **6**

OpenSPARC Processor Configuration

The original UltraSPARC T1 processor was designed for high performance in a very custom semiconductor process, and not many parameters were placed into the code to make it flexible. In the process of opening up the code for OpenSPARC, several compilation options for the core were inserted into the code to allow for more flexibility.

This chapter describes processor configuration in the following sections:

The first section describes existing compilation options that are available in the T1 core. The remaining sections describe other changes that could be made to the core. Most of the changes are described for the T1 core, although similar changes could be made for the T2 core. One change described is specific to the T2 core: the addition of a second floating-point unit. The instructions for these changes are written at a high level and do not describe all the details that will need to be taken care of to successfully implement these changes.

6.1 Selecting Compilation Options in the T1 Core

The new compilation options and, parameters discussed in this section are available in release 1.5 (and later releases) of OpenSPARC T1. For more information about the use of these parameters, please see the *OpenSPARC T1 Design and Verification Manual*, which is included in the OpenSPARC download package.

6.1.1 FPGA_SYN

The FPGA_SYN compiler option enables optimizations to the code for field-programmable gate array (FPGA) synthesis. These optimizations enable the synthesis tool to effectively use block RAM on the FPGA, to increase the synthesis efficiency, and to replace non-synthesizable constructs with synthesizable ones. This option is required for synthesis of the OpenSPARC T1 core for an FPGA. It is also required to enable all the other possible options for the core. To specify this option on a compilation command line, use the following option.

```
+define+FPGA_SYN
```

To specify this option in a Verilog file, use the following line.

```
'define FPGA_SYN
```

6.1.2 FPGA_SYN_1THREAD

The FPGA_SYN_1THREAD option selects a one-thread version of the OpenSPARC T1 core. The one-thread option simplifies the operation of the core significantly and reduces the size of the netlist by up to 40%. The external interfaces of the chip are not affected by this change.

6.1.3 FPGA_SYN_NO_SPU

The FPGA_SYN_NO_SPU option removes the stream processing unit (SPU) from the design. This removal reduces the size of the synthesized design. The SPU is used to accelerate cryptographic functions. It contains a modular arithmetic unit. If this functionality is not needed, then you can use this option to remove the block. Doing so saves about 5000 lookup tables (LUTs) on the Xilinx Virtex-4 FPGA family.

6.1.4 `FPGA_SYN_8TLB`

The `FPGA_SYN_8TLB` option reduces both the instruction TLB size and data TLB size from 64 entries to 8 entries. Since these TLBs are fully associative, they do not synthesize into block RAM but are implemented in logic gates. This option reduces the size of the synthesized design by up to 8000 LUTs on the Xilinx Virtex-4 family, but because of reduced TLB hit rates, it does reduce performance when an operating system is running on the core.

6.1.5 `FPGA_SYN_16TLB`

The `FPGA_SYN_16TLB` option is similar to the `FPGA_SYN_8TLB` option. This option reduces the TLB size from 64 entries to 16 entries. Only one of these TLB size options may be defined.

6.1.6 Future Options

Additional compilation options may be added to the code in future versions of the OpenSPARC T1. Here is a list of possible options:

- Afford more flexibility in selecting TLB sizes: The current design allows for three TLB sizes: 64 entries (the default), 16, or 8 entries. Options could be added to allow for a 32-entry TLB size.
- Remove parity checking, ECC checks, and extra columns: The code for OpenSPARC T1 has parity checking on I-cache and D-cache, and ECC checking on register files. It also has extra columns in many arrays to allow remapping of bad columns. This extra circuitry could be removed to save area on an FPGA design.

6.2 Changing Level-1 Cache Sizes

The instruction cache and data cache sizes on the OpenSPARC T1 core are not parameterized. Here are some instructions on how to change various parameters of the level-1 caches. The instructions here are shown for

OpenSPARC T1. The equivalent changes for OpenSPARC T2 will be similar, although file and signal names will change.

6.2.1 Doubling the Size of the I-cache

This modification involves doubling the number of sets in the I-cache. For this to be accomplished, one additional bit of the physical address (PA) is allocated to the set number, and one fewer bit is needed for the tag.

Changing the cache size involves changes to the following files.

IFU files:

`sparc_ifu.v`	`IFU top level`
`sparc_ifu_ifqctl.v`	`IFU cache fill queue control`
`sparc_ifu_ifqdp.v`	`IFU cache fill queue datapath`
`bw_r_icd.v`	`I-cache data array`
`bw_r_idct.v`	`I-cache or D-cache tag array`

L2 cache files:

`sctag.v`	`L2 Cache top level`
`sctag_dir_ctl.v`	`Reverse L1 directory control`
`bw_r_dcm.v`	`Reverse L1 directory array`
`sctag_dirvec_dp.v`	`Directory vector datapath`

Here is a list of changes necessary to double the I-cache size:

1. The first change is to double the size of the tag array and data array and to increase the address for these arrays by one bit.
2. The data width of the tag array is reduced by one bit or the LSB of the tag array is always written to zero.
3. In the L2 cache, the directory arrays that keep track of the L1 caches must also be doubled in size.
4. Since the L1 directory arrays are doubled in size, the address signals to those arrays must also double in size.
5. The size of the processor-to-cache (PCX) or cache-to-processor (CPX) interfaces does not have to change, and most packet formats stay the same. However, the CPX store acknowledge packet and the invalidation need to change to enable reporting of the set number that must be invalidated. Currently the set number is reported in bits 117:12 (for PA{11:6}) and bits 122:121 (for PA{5:4}). The field needs to be enlarged to allow PA{12} to be included in the invalidation packet as well. This can be done by shifting bits 125:118 up one (bits 127:126 are not used).

6.2.2 Doubling the Number of Ways in the I-cache

Doubling the number of ways in the cache doubles the size of the cache without affecting the address bits assigned to the tag and set. However, the way number is passed in several packet formats to the L2 cache over the PCX interface. This means that one extra bit is required on the PCX interface to send this information. In addition, the way number is sent in invalidation packets from the L2 back to the core over the CPX interface. This means that invalidation packet formats will have to change as well, although this change could probably be made without the need to increase the CPX packet size. Here is a partial list of files that would be affected by this change.

IFU files:

`sparc.v`	`T1 core top level`
`sparc_ifu.v`	`IFU top level`
`sparc_ifu_ifqctl.v`	`IFU cache fill queue control`
`sparc_ifu_ifqdp.v`	`IFU cache fill queue datapath`
`sparc_ifu_invctl.v`	`IFU buffering block`
`sparc_ifu_errdp.v`	`IFU error checking logic`
`bw_r_icd.v`	`I-cache data array`
`bw_r_idct.v`	`I-cache or D-cache tag array`
`bw_r_rf16x32.v`	`I-cache or D-cache valid bit array`

LSU files:

`lsu.v`	`Top-level LSU block`
`lsu_qctl1.v`	`LSU Queue control`
`lsu_qctl2.v`	`LSU queue control`
`lsu_qdp1.v`	`LSU queue datapath`

Crossbar files:

`ccx.v`	`Top-level Crossbar`
`pcx.v`	`Processor-to-cache side.`
`pcx_*.v`	`Various PCX files.`

L2 cache files:

`iop.h`	`Top-level include file`
`sctag.v`	`L2 Cache top level`
`sctag_dir_ctl.v`	`Reverse L1 directory control`
`bw_r_dcm.v`	`Reverse L1 directory array`
`sctag_dirvec_dp.v`	`Directory vector datapath`

Here is a list of changes necessary to double the number of ways in the I-cache:

1. First, the I-cache data, tag, and valid bit arrays will need to be modified to increase the number of ways to eight. Note that tag and valid bit array files are shared with the data cache, so the files will need to be copied if the same changes are not being made to the data cache.
2. With eight ways, the number of bits needed for encoding the way number increases by one bit. This change needs to be made in all the miss buffers for the cache. In file `sparc_ifu_ifqdp.v`, the way number is currently stored as bits 41:40 of the miss buffer entry. This will need to be increased to bits 42:40, and the tag parity bit will need to move to bit 43.
3. Since the number of ways in the cache is now eight, the tag parity checking logic must check parity on eight ways.
4. The tags read from the I-cache tag array are fed to the ITLB. Therefore, the ITLB must be modified to support tags for an 8-way cache.
5. I-cache miss requests are forwarded to the LSU, which then forwards the request to the L2 cache. All files on this path will need to be modified to add support for one more bit for the cache way number. Note that all other packet types will need to change as well so that the packet fields remain consistent. This could affect several more core files.
6. The PCX side of the cache crossbar will need to increase the width of the bus by one bit.
7. The L2 cache needs to double the size of the I-cache directory. In addition, it needs to modify the pipeline to accommodate the increased size of the way number.
8. The invalidation vector sent from L2 to the core must increase by 4 bits. The CPX packet may be able to accommodate this change without increasing the size of the CPX packet. Fortunately, the L2 cache code is parameterized, so many changes can be made by changes to the compiler defines in file `iop.h`.

6.2.3 Changing Data Cache Sizes

Changing the size of the data cache is similar to changing the size of the I-cache. In fact, the valid bit array and the tag array are implemented with the same files as the I-cache. Here are some of the files that would need to be changed to implement a cache size change.

LSU files:

`lsu.v`	Top-level LSU block
`lsu_dctl.v`	Data cache control
`lsu_dctldp.v`	Data cache datapath

`lsu_dcdp.v`	Data cache datapath
`lsu_qctl1.v`	LSU Queue control
`lsu_qctl2.v`	LSU queue control
`lsu_qdp1.v`	LSU queue datapath

Crossbar files:

`ccx.v`	Top-level Crossbar
`pcx.v`	Processor-to-cache side.
`pcx_*.v`	Various PCX files.

L2 cache files:

`iop.h`	Top-level include file
`sctag.v`	L2 Cache top level
`sctag_dir_ctl.v`	Reverse L1 directory control
`bw_r_dcm.v`	Reverse L1 directory array
`sctag_dirvec_dp.v`	Directory vector datapath

Changes to the data cache will require the same steps as the corresponding changes to the I-cache.

6.3 Changing Number of Threads

The OpenSPARC T1 release 1.5 (and later) code contains a compiler define to select between a four-thread core (the original) and a reduced one-thread core. If a different number of threads is desired (for example, a two-thread core), the code could be modified to support this. All the places where the code needs to change can be found by a search for the compiler define `FPGA_SYN_1THREAD` in all the RTL files. Code can then be inserted in these places to support a two-thread configuration.

6.4 Removing the Floating-Point Front-End Unit (FFU)

In OpenSPARC T1, the floating-point unit (FPU) is external to the core. However, the core does contain a block called the floating-point front-end unit (FFU), which contains the floating-point register file. These registers are not used by the floating-point software emulation code, so they are not needed if the external FPU is not in the system. Here is the list of file that must be edited.

`sparc.v`	T1 core top level

The removal of the FFU requires the following steps:

1. The instance of module `sparc_ffu` is removed from the OpenSPARC T1 core.
2. Output signals from this module must be tied appropriately. TABLE 6-1 lists the output signals.

TABLE 6-1 Treatment of FFU Output Signals

Signal	**Action**
ffu_lsu_data	Tie to zero
ffu_lsu_cc_vld_w2{3:0}	Tie to zero
ffu_lsu_cc_w2{7:0}	Tie to zero
ffu_ifu_ecc_ce_w2	Tie to zero
ffu_ifu_ecc_ue_w2	Tie to zero
ffu_ifu_err_reg_w2	Tie to zero
ffu_ifu_err_reg_w2	Tie to zero
ffu_ifu_err-synd_w2	Tie to zero
ffu_ifu_fpop_done_w2	Tie to zero
ffu_ifu_fst_ce_w	Tie to zero
ffu_ifu_inj_ack	Tie to zero
ffu_ifu_stallreg	Tie to zero
ffu_ifu_tid_w2{1:0}	Tie to zero
ffu_lsu_blk_st_e	Tie to zero
ffu_lsu_blk_st_va_e{5:3}	Tie to zero
ffu_lsu_fpop_rq_vld	Tie to zero
ffu_lsu_kill_fst_w	Tie to zero
ffu_tlu_fpu_cmplt	Tie to zero
ffu_tlu_fpu_tid{1:0}	Tie to zero
ffu_tlu_ill_inst_m	Tie to zero
ffu_tlu_trap_ieee754	Tie to zero
ifu_tlu_trap_other	Tie to zero
ffu_tlu_trap_ue	Tie to zero
so	Tie to zero

6.5 Adding a Second Floating-Point Unit to the OpenSPARC T2 Core

The OpenSPARC T2 core contains a single floating-point and graphics unit (FGU), which is shared by all eight threads in the core. To increase floating-point performance, a second FGU could be added so that each FGU is shared by only four threads. This is a more advanced modification because it involves changes to several blocks and requires understanding of the interactions of all these blocks. The following paragraphs give a high-level description of the different blocks that would need to be changed in order to implement this modification.

The first step is to simply instantiate a second floating-point unit at the top level. The floating-point unit may be instantiated as is, but to save area, some changes could be made. Currently the FGU contains the floating-point register file, which contains the floating point registers for all eight threads. If the FGU is shared by only four threads, the code could be edited to reduce the size of the register file by half. In addition, the thread ID is now reduced from three bits to two. This can be easily accomplished by tying one bit of the thread ID to zero.

The next step is to edit the decode block, which completes the instruction decoding and issues instructions to the floating-point unit. It receives two instructions from the pick unit (PKU), one for each integer pipeline. If both instructions are floating-point operations, the decode block currently must select one of them to be issued and stall the other for one cycle. This is performed by an arbiter that alternately picks one thread or the other. With two floating-point units, this arbiter is no longer necessary, since the decode block will be able to issue two floating-point instructions in the same cycle. Instead, the decode block will need a second floating-point interface.

Next up is the PKU. There is a special case for floating-point divides. If a divide is active, no other floating-point instruction can be issued. The PKU contains logic to disqualify threads from being ready to pick if the next instruction in the thread is a floating-point instruction and a divide is currently active. Currently, an active divide affects all eight threads. Now, with two FGUs, an active divide affects only the four threads in its thread group. This is a simple change, involving only some rewiring at the top level of the PKU.

Next, the interface with the execution unit (EXU) must be modified. The FGU is used to perform several integer instructions as well as floating-point instructions. The operands for these integer instructions come from the integer register file, not from the FRF. Therefore, these operands must be passed directly from the execution (integer) units to the FGU. Once the FGU has completed the instruction, it must pass the result, as well as the condition codes, back to the EXU. With two FGUs in the core, the EXU needs another FGU interface.

Once the pipeline is able to issue instructions to both FGUs, the trap logic unit (TLU) must be modified to handle exceptions from both units. Currently, the FGU sends floating-point exception signals along with a 3-bit TID to the TLU if an exception occurred. In addition, a floating-point trap prediction signal is sent four cycles earlier if a trap is likely to be taken. These signals will be duplicated because there are now two FGUs, but the thread ID will be reduced to two bits. It's a matter of recoding the input logic to ensure that any incoming floating-point exceptions are reported for the correct thread.

Finally, the interface with the load-store unit (LSU) must be modified. Like the floating-point unit, the LSU is shared by all eight threads. The FGU interacts with the LSU for floating-point loads and stores. If we duplicate the FGU, the LSU will need to duplicate the signals on the FGU/LSU interface. In addition, the LSU will need to be able to handle two floating-point I/O operations in the same cycle.

6.6 Changing Level-2 Cache Sizes

The OpenSPARC T1 level-2 cache is contained in two blocks: `sctag`, which is the L2 pipeline and tag array, and `scdata`, which is the data array. The OpenSPARC T1 chip has four instances of each block (four banks). Each bank is 768 KB, with 12 ways and 1024 sets. Bits 7:6 of the physical address are used to select the L2 bank, and bits 17:8 are used to select the L2 index.

Here is a partial list of files that would be affected by a change in the L2 cache size.

`sctata files:`

```
    scdata.v
    scdata_rep.v
    scdata_ctl_io.v
    scdata_periph_io.v
    scdata_subbank.v
    bw_r_l2d.v
```

```
sctag files:
   sctag.v
   sctag_tagctl.v
   sctag_tagdp.v
   bw_r_l2t.v
   sctag_vuad_ctl.v
   sctag_vuad_dpm.v
   sctag_vuad_io.v
   sctag_vuadcol_dp.v
   sctag_vuaddp_ctl
   bw_r_rf32x108.v
```

6.6.1 Changing the Number of L2 Banks

While changing the number of sets or ways in the L2 cache would be contained within the L2 cache, changing the number of L2 banks would involve changes to the core and the crossbar. This is because the core selects the L2 bank destination by using address bits 7 and 6 to select the bank. The core sends a 5-bit request signal to the crossbar so that the crossbar can route the request to one of the four L2 banks or to the I/O block. If the number of L2 banks changes, the load-store unit (LSU) of the OpenSPARC core will need to be changed to modify the number of request bits. The cache crossbar will then need to be changed to reduce the number of destinations.

6.7 Changing the Number of Cores on a Chip

Changing the number of cores on the OpenSPARC T1 or OpenSPARC T2 is a straightforward task. In the OpenSPARC T1 code, file `iop.v` has compiler defines to allow for a variable number of cores in the system. These are currently available to support reduced-size simulation environments. However, the compiler defines can also be used to create systems with different numbers of cores. When a chip top-level is being composed, a variable needs to be defined for every core that will be included. For example, for a design with two cores, the following compiler defines must be set:

```
   // create a chip with two cores
   `define RTL_SPARC0
   `define RTL_SPARC1
```

6.8 Cookbook Example 1: Integrating a New Stall Signaller Into the T1 Thread Scheduler

Stall signals are necessary to pause the execution of an active (schedulable) thread during situations such as an I-cache miss, retirement queues fill, or a pipeline resource that becomes busy.

6.8.1 Background

The OpenSPARC T1's instruction selection stage chooses ready instructions from a pool of active threads (see *OpenSPARC T1 Microarchitecture Specification*, Sun Microsystems, August 2006, available on the OpenSPARC website, for more information on the thread scheduler states). You are primarily interested in transitioning a thread from `Ready` and `Run` states to the `Wait` state.

The signal-level interface that must be provided comprises two signals: a thread stall signal and a thread ready signal, called `my_stall` and `my_resume` in this section. The thread stall signal is asserted whenever a thread should first be placed in the `Wait` state. The thread ready signal is asserted whenever the thread can be started again. The signals should not be asserted simultaneously; however, it is acceptable for neither to be asserted in a given cycle and the values need only be asserted for a single clock. This section assumes that these signals contain one bit per thread (e.g., four bits {3:0}), although a single signal that affects all threads can trivially be substituted.

With these input signals, you will generate two signals: a wait-mask (wm) signal and wait completion signal. These will be combined with existing wait-masks and the completion signal such that any wait-mask can stall a thread, but the thread can only resume when the wait-masks are clear and the operation blocking execution has completed.

This logic closely follows that used for the existing instruction miss (imiss) logic. However, because the different stall causes may become ready independently and the thread can start only when *all* wait-masks have been reset, you should create a new wait-mask instead of piggybacking on an existing one.

TABLE 6-2 lists information about the two modules to be modified.

TABLE 6-2 Modules for Installation of New Stall Signal

Module	Hierarchical Name	Description	Filename
`sparc_ifu_swl`	`ifu.swl`	Switch logic unit	`design/sys/iop/sparc/ifu/rtl/sparc_ifu_swl.v`
`sparc_ifu_thrcmpl`	`ifu.swl.compl`	Stall completion logic	`design/sys/iop/sparc/ifu/rtl/sparc_ifu_thrcmpl.v`

6.8.2 Implementation

To install a new stall signal into the thread scheduler for OpenSPARC T1:

1. Add the stall and thread ready signals to both the thread switch logic and completion logic modules.

   The wait-mask logic is generated with logic to "set" and "reset" the wait-mask value for each thread. The wait-mask is set whenever the stall signal is first asserted and is maintained until reset by the `my_resume` signal. The signal `wm_mywm` is the current wait-mask for this resource.

   ```
   wire [3:0] wmf_next;

   assign  wmf_nxt  = my_stall | // set
                      ( wm_mywm & ~my_resume ); // reset
   ```

2. Within the thread completion logic, add a four-bit register to store the new wait-mask:

   ```
   dffr #(4) wmf_ff(.din ( wmf_nxt ),
                    .q ( wm_mywm ),
                    .clk ( clk ),
                    .rst ( reset ),
                    .se ( se ), .si(), .so() );
   ```

3. At the end of the module, add the following logic to the completion signal:

```
assign completion =((imiss_thrrdy | ~wm_imiss) &
                    (other_thrrdy | ~wm_other) &
                    (stb_retry | ~wm_stbwait) &
                    (my_resume | ~wm_mywm) &
                    (wm_imiss | wm_other | wm_stbwait | wm_mywm));
```

   The completion signal allows the thread to start running again. This will only happen when all wait-masks are clear *and* at least one resume signal has been asserted.

4. Route the new wait-mask signal out of the thread completion unit and into its parent module switch logic. The individual wait-mask signals are combined into `start_thread` and `thaw_thread` signals that control the scheduler finite state machine (FSM). The necessary additions are as follows:

```
assign start_thread = resum_thread &  (~wm_imiss | ifq_dtu_thrrdy) &
                                      (~wm_stbwait | stb_retry) &
                                      (~wm_mywm | my_resume);

assign thaw_thread = resum_thread &   (wm_imiss & ~ifq_dtu_thrrdy |
                                       wm_stbwait & ~stb_retry     |
                                       wm_mywm & ~my_resume);
```

6.8.3 Updating the Monitor

The procedure in the preceding section is sufficient for modifying the core logic to support an additional stall signal. However, a thread scheduler monitor that is used for verification and debugging must also be modified to be aware of the new stall signal.

Edit the files `verif/env/cmp/monitor.v` and `verif/env/cmp/thrfsm_mon.v`. The first file instantiates various microarchitectural monitors and needs to route the wait-mask signal (e.g., `wm_mywm`) into the `thrfsm` monitor. From there, simply copy the logic used for `wm_imiss`, replacing `wm_imiss` with `wm_mywm` on the new lines.

Alternatively, this monitor can be disabled with the `-sim_run_args=+turn_off_thread_monitor` command-line switch to sims.

6.9 Cookbook Example 2: Adding a Special ASI to the T1 Instruction Set

Address space identifiers (ASIs) are useful for exposing internal registers to user-level and privileged software. Because ASI accesses take the form of 64-bit load and store instructions, they offer a convenient and flexible way to read and write both small and large internal structures.

6.9.1 Background

ASI accesses look very much like normal load and store instructions to the assembly programmer. For example, the following program writes a value in architectural register `%l0` to virtual address (VA) 8_{16} at ASI $1A_{16}$ (implemented later in this document) and then reads from the same location into architectural register `%l1`:

```
#define ASI_EXAMPLE     0x1a

        setx 0x08,        %g2, %g1
        setx 0xdeadbeef0, %g2, %l0

        stxa %l0, [%g1] ASI_EXAMPLE
        ldxa [%g1] ASI_EXAMPLE, %l1
```

For simplicity, this section shows how to use non-translating ASIs where the supplied virtual address is used directly by hardware. This is simple and generally the best choice for internal registers because of its simplicity. Other types of ASIs that use mappings to real or physical addresses are also available, but they require more work (see Section 10.2 in the *UltraSPARC Architecture 2005 Specification*, available on the OpenSPARC website, for more information).

In this section, you will piggyback on the existing interface to the scratchpad registers (defined as 20_{16} and $4F_{16}$ for privileged and hyperprivileged registers, respectively) to provide a read-write interface to ASI $1A_{16}$ (an unallocated ASI that can be accessed by both privileged and hyperprivileged programs). The load-store unit (LSU) is responsible for decoding the ASI and routing load and store data between internal registers and the general-purpose registers. Therefore, you can concentrate your changes within the LSU. (The trap logic unit (TLU) contains SRAM for the actual scratchpad registers, but you need not modify the TLU in this example.)

To process an ASI, use the existing signals listed in TABLE 6-3 (all available at the "`sparc`" level of the RTL hierarchy). For simplicity, control signals from the E pipeline stage are chosen, although signals at later stages are also sometimes available. ASI write data is available only in the W/G stage.

Note | The pipeline stages generally follow this convention: F - S - D - E - M - W/G - W2.

TABLE 6-3 Signals for Processing Sample Newly Created ASI

Name	Description
`ifu_lsu_alt_space_e`	Decode signal indicating an ASI load or store
`ifu_lsu_ld_inst_e`	Decode signal indicating a load
`ifu_lsu_st_inst_e`	Decode signal indicating a store
`ifu_tlu_thrid_e`	Decode signal indicating the current thread ID
`lsu_spu_asi_state_e`{7:0}	LSU signal specifying the ASI number
`exu_lsu_ldst_va_e`{47:0}	Virtual address for the ASI access
`lsu_tlu_rs3_data_g`{63:0}	Write data for ASI store instructions (to your internal registers)

You will also create a set of new signals that ASI load instructions use to return data to the LSU. These signals are asserted in the W2 stage. Note that if an ASI load is executed, the valid signal must be asserted. If this does not occur, the load instruction will never complete (the RTL model will eventually halt with a timeout)!

TABLE 6-4 Signals Used by ASI Load Instructions for Data Return to LSU

Name	Description
`myasi_lsu_ldxa_vld_w2`	ASI load data valid signal
`myasi_lsu_ldxa_tid_w2`	ASI load's thread ID
`myasi_lsu_ldxa_data_w2`{63:0}	ASI load data (from your internal registers)

6.9.2 Implementation

Now create a module (`myasi_mod` in this example) that pipes the control signals to the appropriate stages and acts upon them (here, writing to the register for the ASI store and reading from the same register for the ASI load). Next, you must also add logic to tell the LSU that ASI $1A_{16}$ is a valid ASI and route the loaded value through the LSU's bypass network.

Use the above signals to interface with an example internal register in the following simplified Verilog module, which you instantiate in the top-level "`sparc`" Verilog module.

1. Create the module listed below.

CODE EXAMPLE 6-1 The `myasi_mod` Module

```
module myasi_mod ( // Inputs
                   clk,
                   ifu_lsu_alt_space_e,
                   ifu_lsu_ld_inst_e,
                   ifu_lsu_st_inst_e,
                   ifu_tlu_thrid_e,
                   lsu_spu_asi_state_e,
                   exu_lsu_ldst_va_e,
                   lsu_tlu_rs3_data_g,
                   // Outputs
                   myasi_lsu_ldxa_vld_w2,
                   myasi_lsu_ldxa_tid_w2,
                   myasi_lsu_ldxa_data_w2 );

    // ... [input/output/wire declarations] ...

  // Enable signals for loading/storing your ASI in the E stage
  assign asi_ld_e = ifu_lsu_alt_space_e & ifu_lsu_ld_inst_e &
                      ( lsu_spu_asi_state_e == 8'h1A );

  assign asi_st_e = ifu_lsu_alt_space_e & ifu_lsu_st_inst_e &
                      ( lsu_spu_asi_state_e == 8'h1A );

  // ... [Pipe asi_ld_e to stage W2 and asi_st_e to stage W/G] ...
  //      [through stages E -> M -> W/G -> W2]
  //
  // Can also pipe the VA for special operations
  // based upon the VA.

  // Our internal register is written by the store ASI and
  // read by the load ASI instructions
  dffe #(64)
```

CODE EXAMPLE 6-1 The `myasi_mod` Module *(Continued)*

```
     internal_ff ( .clk ( clk ), .en ( asi_st_g ),
                   .din ( lsu_tlu_rs3_data_g ),
                   .q   ( myasi_lsu_ldxa_data_w2 ) );

   // Valid signal and TID asserted at the appropriate stage
   assign myasi_lsu_ldxa_vld_w2 = asi_ld_w2;
   assign myasi_lsu_ldxa_tid_w2 = ifu_tlu_thrid_w2;

endmodule
```

2. Tell the LSU that ASI $1A_{16}$ is now valid by editing the file `sparc/lsu/rtl/lsu_asi_decode.v` and locating the assign statement for `asi_internal_d`. Add a condition for the new ASI value:

```
    assign asi_internal_d =
        (asi_d[7:0] == 8'h1A) |
        [ ... remainder of original assign ... ];
```

3. Route the new signals—
   `myasi_lsu_ldxa_vld_w2`
   `myasi_lsu_ldxa_tid_w2`
   `myasi_lsu_ldxa_vld_w2`
   —through the `lsu` module into `lsu_qdp1` (data and vld signals and `lsu_dctl` (vld and tid).

   a. In `lsu_dctl`, first locate the assign statements for `lmq_byp_data_fmx_sel`{3:0}.

      This signal is normally asserted when the TLU processes an ASI load instruction. It is also asserted when the module replies to an ASI load (there is no conflict here because only one ASI load instruction can be in each pipeline stage at a time).

   b. Ensure that the final code looks like this:

```
assign lmq_byp_data_fmx_sel[0] = ( int_ldxa_vld | myasi_lsu_ldxa_
  vld_w2 ) & thread0_w2 ;
assign lmq_byp_data_fmx_sel[1] = ( int_ldxa_vld | myasi_lsu_ldxa_
  vld_w2 ) & thread1_w2 ;
assign lmq_byp_data_fmx_sel[2] = ( int_ldxa_vld | myasi_lsu_ldxa_
  vld_w2 ) & thread2_w2 ;
assign lmq_byp_data_fmx_sel[3] = ( int_ldxa_vld | myasi_lsu_ldxa_
  vld_w2 ) & thread3_w2 ;
```

4. Also in `lsu_dctl`, locate the assign statement for `ldxa_thrid_w2`{1:0} and mux the current TID from the TLU with the new TID from your unit.

```
// ldxa thread id
// assign ldxa_thrid_w2[1:0] = tlu_lsu_ldxa_tid_w2[1:0] ; //Removed
                                                             original
                                                            TID assign

mux2ds #(2)  // Added mux
    mux_ldxa_thrid  ( .dout ( ldxa_thrid_w2 ),
                      .in0  ( tlu_lsu_ldxa_tid_w2[1:0] ),
                      .in1  ( myasi_lsu_ldxa_tid_w2[1:0] ),
                      .sel0 ( ~myasi_lsu_ldxa_vld_w2 ),
                      .sel1 (  myasi_lsu_ldxa_vld_w2 ) );
```

5. In `lsu_qdp1`, mux your load value with the TLU's value, using your valid signal to distinguish the two requests, and route the signal into the existing `ldbyp0_fmx` mux.

```
wire [63:0]            ldxa_data_w2;

mux2ds #(64) ldbyp0_myasi ( .in0 ( tlu_lsu_int_ldxa_data_w2[63:0] ),
                            .in1 ( myasi_lsu_ldxa_data_w2[63:0] ),
                            .sel0 ( ~myasi_lsu_ldxa_vld_w2 ),
                            .sel1 (  myasi_lsu_ldxa_vld_w2 ),
                            .dout (  ldxa_data_w2 ) );

// Existing mux
mux2ds  #(64) ldbyp0_fmx (
  .in0  (lmq0_bypass_misc_data[63:0]),
  .in1  (ldxa_data_w2[63:0]),    // This input changed from
                                   tlu_lsu_int_ldxa_data_w2[63:0]
  .sel0 (~lmq_byp_data_fmx_sel[0]),
  .sel1 (lmq_byp_data_fmx_sel[0]),
  .dout (lmq0_bypass_data_in[63:0]) );
```

6.9.3 Caveats

This procedure implements a new ASI load/store in OpenSPARC T1. This example avoids processing the virtual address (this could, for example, address entries in an SRAM or read from other internal registers). This example also ignores validity checking on the VA address (this raises an exception). This implementation also requires strict timing on responses to ASI loads. ASIs with variable-latency responses can also be implemented.

CHAPTER 7

OpenSPARC Design Verification Methodology

Verification is the process of matching design behavior with designer intent. Commercial design verification is far from being a simple process. It involves the following:

- Many abstraction levels, such as architectural, register transfer logic (RTL), gate, and circuit
- Many different aspects of design, such as functionality, timing, power, and manufacturability
- Many different design styles, such as system-on-a-chip (SoC, also called server-on-a-chip), ASIC, custom, synchronous, and asynchronous

As numerous publications attest, verification continues to be the long pole in product development and more than half of time and resources are spent in verification alone. Challenges of SoC verification are on the rise as submicron process technology allows exponential growth in functionality that can be included on a piece of silicon. And this is no different for the OpenSPARC processors.

Despite the best efforts in the electronic design automation (EDA) industry, the verification tools and technologies have not kept pace with growth of design size and complexity of I/Os that are part of the SoCs today. This verification challenge can be met by enhancement of verification efficiency, adoption of aggressive changes in verification methodology, and use of state-of-the-art verification tools and technologies.

A robust verification methodology encompassing commercial and in-house tools was deployed on UltraSPARC processors (open-sourced as OpenSPARC designs) to minimize post-silicon defects and to facilitate concurrent hardware and software engineering in order to speed up the time-to-ramp on Sun Fire servers.

This chapter contains the following sections:

7.1 Verification Strategy

The design process involves transforming the product specifications into an implementation. It is nearly impossible to go directly from specifications of the product to an implementation description that can be used by a foundry to transform the specification into silicon. This is largely due to the lack of a specification language that can completely describe high-level functionality along with timing and power targets and related tools that can synthesize this specification into an implementation.

Before an implementation version of the design can be obtained, the high-level specifications are transformed into a number of abstraction levels targeted to accomplish a set of specific verification objectives. It is during this process that a number of errors may be (and usually are) introduced. The later the error is discovered, the costlier the impact is on both product cost and development schedule, as depicted in Table 9.1. It is therefore critical to deploy a verification strategy that not only minimizes errors introduced in this process but efficiently finds errors so that they can be rooted out before they show up as silicon defects resulting in silicon respins.

TABLE 7-1 Analysis of Generalized Cost of Show-Stopper Bug

Discovery	Cost
Early in design phase	Almost negligible
Close to tape-out	Schedule impact
After tape-out	A few silicon respins, schedule impact
Close to revenue release	Loss of revenue at \$1+ million per day
After revenue release	Cost of recall at \$50 million per day; damage to reputation

The famous Pentium chip FDIV bug cost Intel \$475 million in 1994.

An effective strategy incorporates an aggressive approach to finding design errors early at the targeted abstraction levels to minimize functional issues in silicon, with the goal of finding problems before design tape-out. Also, reduced visibility in silicon, generally limited to latches and architectural state, poses a significant debugging challenge. Therefore, it is critical to devise a robust pre-silicon verification strategy from the start of a project. Verification strategy starts with the definition of a verification plan derived from the design specification, as shown in FIGURE 7-1.

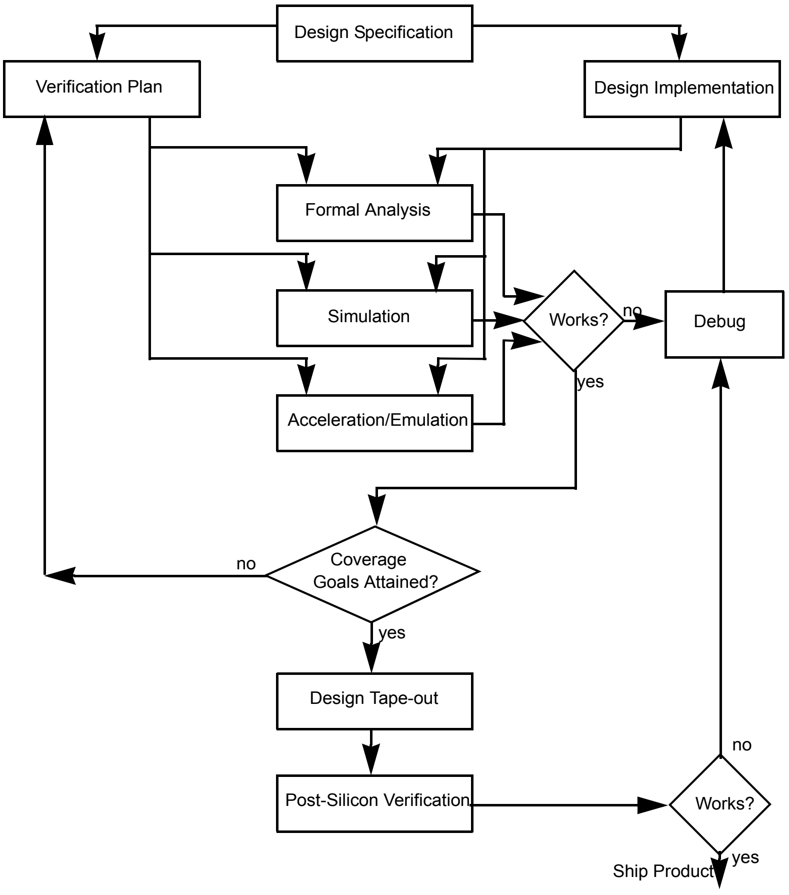


FIGURE 7-1 Verification Strategy Flow Chart

At a high level, the verification plan details different testing objectives, related metrics, and coverage goals, along with details of what, how-to, and how much of this is covered. Basically, the plan mirrors the requirements specification.

The OpenSPARC processors feature a unique multicore, multithreaded design that supports up to 64 simultaneous threads, posing new verification challenges. The processor design along with its system-level testbench is unusually large (approximately 35 million gates), pushing the capacity and performance envelopes on all tools.

To battle these challenges, the OpenSPARC verification strategy employs both divide-and-conquer and then merge-and-conquer tactics. Sun's OpenSPARC designs are composed of units that interact with each other in the larger system. Due to economies of scale, it is cost effective to start by testing at the unit level, with specially designed testbenches for the unit under test. To augment unit-level verification, formal analysis tools are deployed early in design phase to conquer contained complexities such as arbiters, fairness protocols, FIFO structures, equivalency, and clock-domain crossings. As the unit designs mature, they are integrated into large simulation models, eventually leading to simulation of the entire design.

Testing the entire design requires tremendous compute resources. At some point, traditional software-based simulators come to a crawl. These simulations typically run at single-digit cycles per second (CPS) and lack the bandwidth to run system-level simulations. This is where hardware-based acceleration or emulation technologies are deployed to speed up the simulation by six orders of magnitude. To extend this coverage further, a virtual prototype of design can also be obtained by mapping the design to FPGAs, which could be handed out as "virtual silicon" to software teams to facilitate software development.

Since the design state space is extremely large (almost incalculable), it is simply impossible to traverse all states to obtain exhaustive coverage even with the best-of-breed simulation, emulation, and formal analysis tools. At best, these tools only provide a statistical level of certainty about the correctness of design—which is why bugs can slip past pre-silicon verification into post-silicon verification. Therefore, a robust post-silicon verification strategy is devised to capitalize on increased simulation bandwidth while addressing limited design visibility. A number of systems with the SoC are assembled and deployed 24x7 to run variety of random generators and real-world applications to root out the last few bug escapes in the design.

7.2 Models

As discussed in the previous section, design development warrants creation of multiple descriptions of design that model different levels of abstraction along with related simulation environments to support testing. These models are created as the design specifications are transformed to an implementation that can be sent to the foundry for manufacturing, as depicted in FIGURE 7-2.

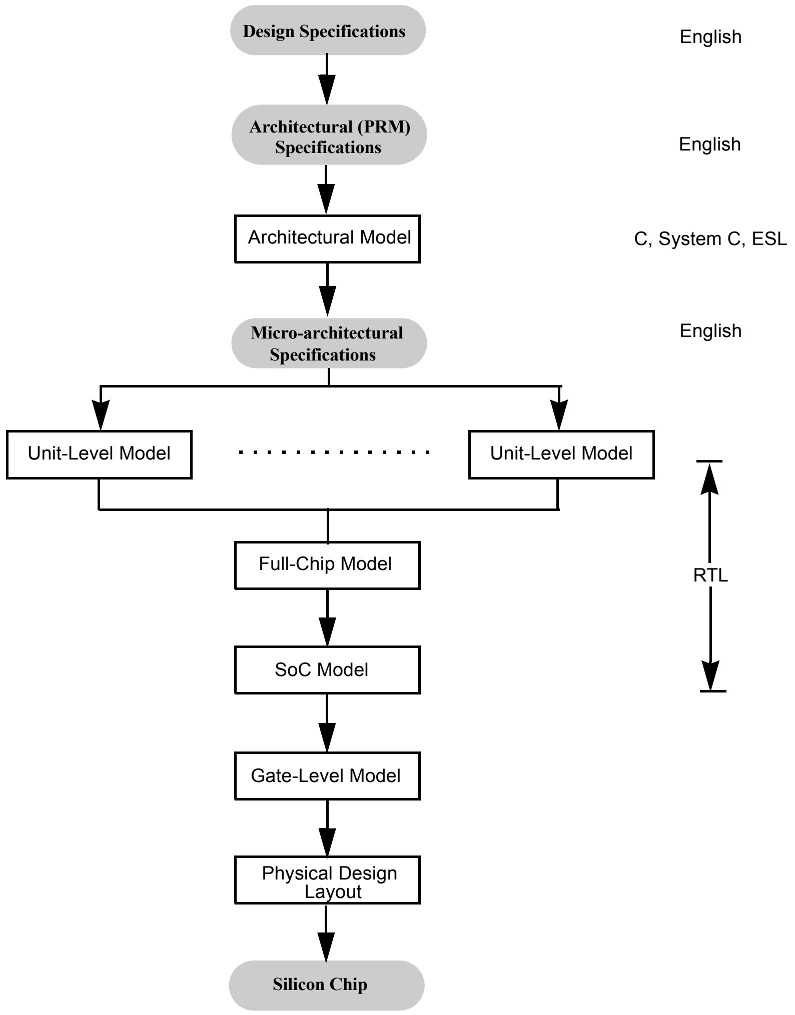


FIGURE 7-2 Different Design Abstraction Levels and Related Verification Models

7.2.1 Architectural Model

To begin, the product specification or requirement is gathered by consultation with potential customers and by compatibility with previously existing related products. The specification is usually in a nonformal, English-like description. From these specifications, a set of architectural specification is derived. This specification, also known as Programmer Reference Manual (PRM), describes the high-level functionality of the machine from a user's perspective. A high-level model that simulates the architecture (behavior) of the design is developed in a high-level programming language such as C or C++ and is called the Instruction-Set Simulator (ISS). This model includes only the programmer-visible registers, simulates the effects of the execution of an instruction sequence at high speeds, and serves as a golden reference model during different stages of design verification. This model is only instruction accurate and not clock-cycle accurate (does not implement an accurate timing model). On OpenSPARC we deployed the internally developed ISS called SPARC Architecture Model (SAM), which includes a core instruction-set simulator, referred to as "Riesling." Both of these tools have been open-sourced and are described in detail in Chapter 10, *System Simulation, Bringup, and Verification*.

The OpenSPARC design is composed of a number of functional units, each with its own area of responsibilities. Many design trade-offs come into play when defining what a unit is, how a unit is implemented, and what the communication protocols at unit interfaces are. Breaking the design into units is a necessary part of the design process and allows the formation of teams for work on individual unit design and implementation in parallel with other units. The high-level architectural model promotes exploration of possible unit designs and configurations that eventually lead to a final solution. Functionality of these individual units is described in detail in English, in a document called a microarchitectural specification.

7.2.2 Unit-Level Models

With the microarchitectural specification (MAS) completed, the unit-level interfaces are defined, and the functionality described in the RTL is defined in Verilog language. Stand-alone testbench (SAT) environments are developed to test these block-level designs. A SAT model comprises the device (or unit) under test (DUT) and a testbench that applies stimulus and checks results. This SAT-level environment is verified with software simulators. SAT environments require an investment of engineering time and resources but are cost effective because of economies of scale. SATs are the most cost-effective way to find bugs.

SAT environments contain moderately small amounts of RTL that are observed with great detail by logic-behavior checkers. These environments are small enough that the simulation speed is quite high. Testing starts with hand-written diagnostics called directed tests. Basic functionality is tested first, followed by complex behavior, and then what is called corner-case testing. After this, pseudo-random numbers are often used to create viable stimuli in combinations that directed coverage tests might not have considered.

For example, tests at Sun had more than 15 SAT-level environments focused on testing specific functionality of blocks such as SPC, FGU, IFU, TLU, LSU, SPU, PMU, MCU, PEU, TCU, NIU. These SAT environments sport a high degree of controllability and observability required to quickly identify and fix functional errors related to those blocks.

7.2.3 Full-Chip Model

When it is deemed that sufficient testing has been accomplished at unit-level SAT environments, the units are put together to make up the full-chip processor model. An environment much more elaborate than that in a SAT is required to exercise the full-chip functionality. The full-chip model also includes the main system memory, which is primarily a sparse memory model for software simulation environments. The testbench provides the required infrastructure to read in assembly language programs, execute them on the full-chip model, and then check results in lock-step with the architectural model, which serves as the golden reference. This model helps identify errors due to interface assumptions, incorrect interpretation of specifications, and issues related to functionality of the processor design.

Where possible, parts of the full design are left out of this model, including clock trees, scan chains, hardware error protection, and detection logic. Also, as in the SAT environments, large latch and memory arrays are implemented behaviorally, with protocol monitors at their interfaces to be sure the arrays are used correctly.

Although software-based simulators can simulate a model of this scale, they do not provide enough simulation cycles in time to meet the schedule deadlines a competitive product requires. High-speed simulation engines such as accelerators, emulators, and FPGA prototyping (discussed in subsequent sections) are deployed to increase simulation bandwidth on full-chip models.

7.2.4 SoC-Level Model

An OpenSPARC processor is truly a server-on-a-chip or system-on-a-chip (SoC): It not only incorporates the multicore, multithreaded processor but also integrates several I/Os such as Gigabit Ethernet and PCI-Express on the same silicon die. This model also includes a number of Design For Test (DFT)-related capabilities instrumented into silicon to facilitate manufacturing and test of the part. Therefore, the SoC-level model incorporates all of these along with an elaborate testbench that facilitates testing at this level. This model ends up being unusually large, posing a challenge to all verification tools.

This level of modeling is simulated most cost effectively with hardware-based simulators—accelerator/emulators, to achieve the bandwidth to meet verification objectives defined for this model.

7.3 Verification Methods

A number of design verification methods have evolved to aid in the accomplishment of defined verification objectives. All of these methods can be broadly categorized as simulation, formal, and emulation verification technologies and are described in subsequent sections.

A wide selection of tools is available from different EDA vendors. However, verification tools have not been able to keep pace with the growing design size and complexity of today's processors. No one vendor can provide a complete end-to-end tool suite that can meet the verification objectives of a large SoC such as OpenSPARC. Instead, most users develop a number of internal tools and utilities to fill the void. This is no different for Sun. For effective verification of OpenSPARC SoC and to ensure first-silicon success, state-of-the-art external and internal tools reinforced the three methods of simulation, formal, and emulation verification.

FIGURE 7-3 shows a typical verification time line and the different classes of verification tools that play a dominant role as the design progresses through block, full-chip, system, and productization (silicon validation) phases. Subsequent sections in this chapter describe these methods, related tools, and the way they play a significant role in ensuring a high-quality silicon.

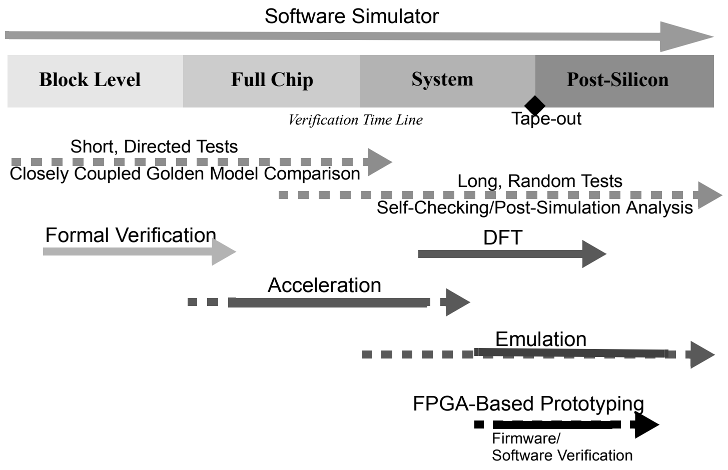


FIGURE 7-3 Typical Verification Time Line and Tools

7.4 Simulation Verification

Simulation verification is probably the oldest verification method but has undergone minimal evolution as designs have rapidly grown in size and complexity. It also continues to be the main workhorse for verification. FIGURE 7-4 shows currently available simulation engines along with their relative modeling effort required and their expected simulation performance.

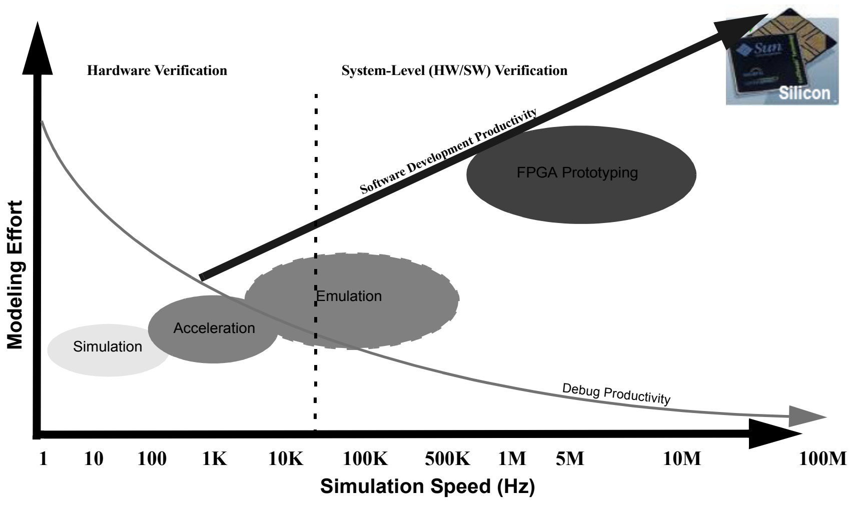


FIGURE 7-4 RTL Simulation Engines, Modeling Effort, & Simulation Performance

At Sun, varying combinations of these platforms are used to achieve specific hardware and software verification objectives. It is evident that with an increased modeling effort greater speed-ups can be obtained. However, the debug complexity increases significantly—silicon with its limited visibility is the hardest to debug. Simulation acceleration, while well suited to RTL verification, lacks the performance to enable software development and test.

Simulation verification involves compiling the design under test and a supporting testbench along with simulator software that can read and execute this design description. The resulting program is then run on general-purpose computers to produce the output results. The simulated output results are compared to results derived from an architectural-level instruction-set simulator (ISS) that serves as a golden reference model. Any deviations in the simulated results are then debugged to identify offending design constructs.

A number of times the simulation environment contains the following: the design under test; any behavioral models that can be substituted for large parts of the OpenSPARC design that are not under test (this increases simulation performance); design libraries; hardware assertion libraries; coverage measurement instrumentation; and verification infrastructure like hardware behavior monitors as well as the test stimuli (directed or random). These components must be compiled together to build an executable simulation model. These components are described in detail in following sections.

7.4.1 Testbench

Additional infrastructure or verification components are often required to facilitate testing the device under test (DUT). The primary purpose of this infrastructure that comprises the testbench is to stimulate the design and capture results. Testbenches also facilitate comparison of output results with expected results either in-sync with a simulation run or offline. Other infrastructure, such as clock generation, monitors to watch and verify transactions, or bus protocols, are also part of the testbench. FIGURE 7-5 shows various components of a typical testbench.

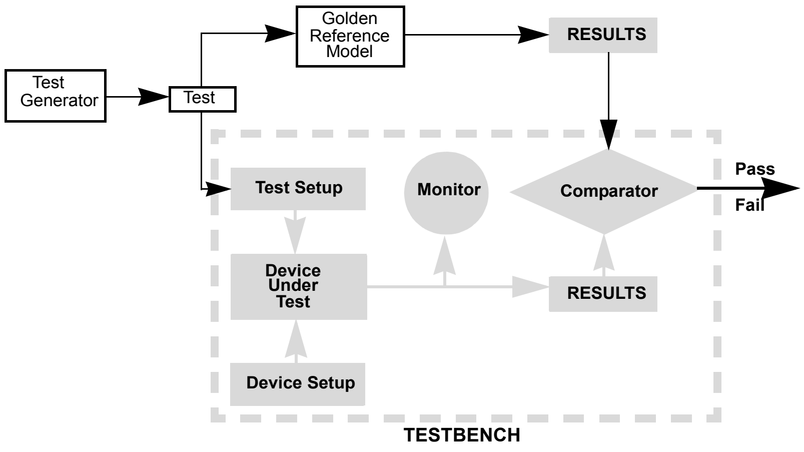


FIGURE 7-5 Simple Testbench Part of Simulation Environment

Sometimes, depending on the functionality required, testbenches are more complex and larger than the DUT itself. Therefore, complex testbenches are relatively easier to design in higher-level languages than Verilog. Languages such as Vera, Native Test Bench (NTB), System Verilog, and C are generally popular in the industry for design of testbenches. For the OpenSPARC design, Vera (from Synopsys) was extensively used in the simulation environment. Since these testbench languages afford a high degree of flexibility in the development of complex testing infrastructure, they are not synthesizable.

Full-chip software simulation testbenches can also include the use of an architectural-level instruction-set simulator (ISS) (typically written in C or C++, so it is very fast). The ISS runs the same program as the simulated design. This ISS (OpenSPARC uses an in-house simulator named Riesling) serves as a reference to compare all architecturally visible registers such as program counters, and integer and floating point registers.

Building a testbench for emulation verification involves making it synthesizable. The primary motivation for emulation verification is to run as fast as possible. This prevents the inclusion or implementation of many of the features available in simulation testbenches that cannot be synthesized and placed in the emulation hardware.

Note For emulation verification, the testbenches must be synthesizable. The fallback in this case is to use the Verilog RTL to design simpler testbenches. This warrants creation and maintenance of multiple testbenches to support multiple models and verification environments.

Non-synthesizable testbenches can also be interfaced to emulation hardware in co-simulation mode, but this approach slows down the emulator. Therefore, this mode is used early on to boost debug productivity.

For the T2 SoC, a number of testbenches were designed and built to facilitate target verification at different levels of design hierarchy. FIGURE 7-6 depicts a hierarchy of testbenches developed.

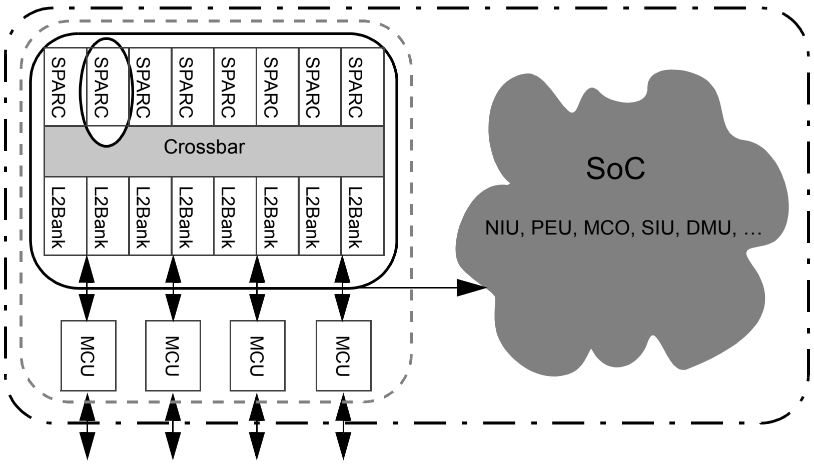


FIGURE 7-6 Hierarchy of Testbenches on OpenSPARC T2

7.4.2 Assertions

Assertions are formal properties that describe design functionality or temporal relationships in a concise, unambiguous way and that are machine executable. Such formal semantics makes it easier to express low-level signal behaviors such as interface protocols. Assertions can be embedded inline along with the RTL of the design by designers or they can reside in separate files typically written by verification engineers. Inline assertions enhance readability, improve observability in the design, and help detect and diagnose design issues efficiently.

Designers write assertions to capture critical assumptions and design intentions during the implementation phase, whereas verification engineers focus on functional checks on external interfaces based on the design specifications. Assertions can also be leveraged as coverage metrics that in combination with other forms of coverage such as code coverage (line, toggle, branch, etc.) provide data about the effectiveness of verification performed on the design. Thus, assertions not only contribute a mechanism to verify design correctness but also afford a technique to simplify and ease debugging and a measure of verification quality.

A number of assertion languages such as System Verilog Assertions (SVA), Synopsys Vera - Open-Vera Assertions (OVA), and Mentor's Zero-in exist. For OpenSPARC T1 and T2 development, the Zero-in tool was benchmarked and selected. (SVA support was not available in the majority of tools at that time.) Zero-in was chosen for its simplistic, easy-to-learn techniques of writing assertions as comment pragmas and for its large suite of tools to leverage the value of assertions. A longer-term strategy was to migrate to use of System Verilog Assertions.

A number of open source assertion libraries can be used to simplify the writing of assertions. The OpenSPARC RTL source available for download contains thousands of assertions. Assertions are embedded with the RTL code and start with `//0in` or `/*0in`. The Vera code resides in testbench files that have a suffix `*.vr`; this is true for both T1 and T2 packages.

While assertions are predominantly used in a simulation environment, they also bring significant value in formal and emulation verification. In fact, assertions can be first verified by formal verification even before the testbench needed for simulation is ready. These assertions can also be reused from a SAT model to an SoC model. Both emulation and formal tools require synthesizable versions of the assertions.

7.4.3 Coverage

There is no generally accepted measure of verification coverage, which makes it difficult to determine how much verification is enough. Several standard approaches, however, attempt to measure verification coverage. Coverage metrics are widely used to automatically record information and analyze it to determine whether a particular test verified a specific feature. Coverage metrics can quantify the verification plan accomplished so far and can pinpoint areas that require additional verification effort.

This coupling between coverage collection and test generation, also known as Coverage Driven Verification, is either manual or automatic. In this case, the random stimulus generation or directed tests are tuned through coverage measurements collected. These metrics also accord a measure of confidence for the quality of verification.

Note It should be made clear that coverage statistic instrumentation and collection come at a cost of additional effort and slowdown in simulation speed.

Coverage metrics can be broadly classified as structural coverage and functional coverage.

Structural coverage, also known as code coverage, provides insight into how well the code (RTL, gate) of the design is stimulated by tests during simulation. There are a number of different code coverage types based on different entities of the design structure:

- **Toggle coverage** — Measures the number of times variables, signals, ports and buses toggle.
- **Statement coverage** — Tracks different statements exercised in the design.
- **Conditional (path or expression) coverage** —Measures different paths of conditional statements covered during simulation
- **State (transition and sequence) coverage** — Measures the number of states visited of the total number of states encoded in a finite state machine.

The state space for structural coverage is well defined, and a number of mature commercial solutions are therefore available in this space.

Functional coverage measures how well the operational features or functions in the design are exercised during simulation. Unlike code coverage, which requires minimal effort from the designer, functional coverage requires coding additional coverage objects. These coverage objects specify functions of the device and are derived from the functional specifications of the design. It is important to emphasize that the quality of functional coverage depends on completeness of the functional specifications and completeness of coverage objects enumerated from them. No automated means to determine this completeness is available. More often it is not possible to enumerate all coverage objects, because specifications can be ambiguous. Functional objects can be defined with System Verilog or vendor-specific languages such as Vera or Zero-in.

Coverage collection mechanism are built into simulators. Coverage data can be extracted and aggregated for analysis and reporting by simulator-specific Application Programming Interfaces (APIs). These APIs are vendor specific and so pose an issue of interfacing different tools. Availability of a common set of standardized APIs will greatly help simplify ease of use.

7.4.4 Directed Testing

Directed tests that express a specific testing goal are manually written. Typical directed tests consist of a set of test cases, each stimulating and possibly checking a specific functional behavior. Directed testing is done in phases, starting with basic testing, leading to testing of complex behaviors and corner-case testing.

The checking of correct hardware behavior can sometimes be implemented as part of the directed test, but typically much of the hardware function is inaccessible to directed tests. Instead, other software, including possibly assertions or monitors, observes the hardware cycle-by-cycle to check for incorrect behavior.

Simple checks can be implemented with the use of assertions. However, complex checking can be accomplished by the use of monitors in the implementation of hardware behavior. Monitors can sometimes be implemented as state machines that are much like the state machine under test. Monitor state can then be used to check hardware state.

7.4.5 Random Test Generation

Computer designs are so complicated these days that it can be impossible for humans to think through the entire testing space. At some point the people who write directed tests run out of testing concepts to implement. This rarely means that all the design problems have been found and fixed. Pseudo random number generators can often push the hardware designs harder by stimulating the logic under test in scenarios beyond the test developer’s imagination. This testing technique depends heavily on monitors and assertions to detect incorrect hardware behavior. Problems uncovered by random testing can engender new concepts for increasing the directed test base.

Random test generation is first implemented in SAT unit-test environments, in which the stimuli can be the most diverse and have the best checking. In the full-chip environment, there are typically different random test generators that focus on particular architectural features of a design. Some random test generators focus on conditional branch testing, others focus on memory coherency, yet others might focus on virtual memory address translations. For OpenSPARC SoC, all of these test generators were implemented to effectively verify these targeted functionalities.

Most of these generators run on the Solaris OS and produce random tests in files. These files are then used in the simulation environment to test the hardware. In other cases, particularly in SATs, the random numbers are calculated and used to produce stimuli on the fly. Some random generators can do both. One of Sun's most powerful pre-silicon random testing environments enables templatized random testing, which is a combination of directed testing and random testing. This testing vehicle is flexible and can be targeted at different simulation environments.

Compared with random testing, directed testing is highly quantifiable; the number of directed tests is finite and the tests pass. With random testing, it becomes hard to know when enough testing is done; coverage measurements can aid in understanding if enough random testing has been done so that suitable measures to augment verification can be taken.

7.4.6 Result Checking

Tests are run and output results are produced. This output response must be compared with expected data to determine if the test passed or failed, as depicted in FIGURE 7-3 on page 93. A number of mechanisms accomplish the comparison. For directed testing, the results are embedded in the test or complex monitors are implemented to monitor hardware behavior as the test is run. A number of times, checking objects are spawned along with "data" objects that check results at the output and are terminated on successful checks.

In a system-level environment, one common strategy involves comparison with the output from a golden reference model. The golden reference model used in the full-chip simulation environment is an architectural-level, multithreaded SPARC simulator developed in a high-level programming language such as C. This is the Riesling SPARC Architectural Simulator, available as part of the OpenSPARC download package. This architectural model, also referred to as Instruction-Set Simulator (ISS), compares the full architectural state after every instruction execution. The model provides instruction-level accuracy, but since it does not model microarchitecture, it is not cycle accurate. It does enable event synchronization.

The reference model can be closely coupled with the testbench and simulator in a cosimulation environment. In this configuration, the results are compared after execution of every instruction and simulation stops at the first failure. The data logged from this simulation forms the initial basis of failure triage.

Self-checking tests embed expected result data in the tests and are compared during test execution. Some test generators can "boot" (load themselves on the CPU under test), generate tests, run them, and check results.

Simulation environments in which self-checking tests either lack strict data checking or cannot cosimulate with the reference model, the simulation results are logged into testbench memory, which is dumped to a file and analyzed after simulation completes. This is known as the post-simulation analysis method.

7.5 Formal Verification

Formal verification is the use of mathematical techniques to gain 100 percent accuracy and coverage in checking properties of functional models and in comparing two different models. Formal methods have long been advertised as a means to produce provably correct implementations. Formal verification is nearing maturity to a point where it is much easier to adopt. The notion of a Ph.D. degree being required in order for the formal tools to be deployed in verification is long gone.

Formal tools also enhance productivity by integrating and leveraging simulation. They promise to find tough corner-case functional issues while providing an exhaustive proof earlier in design phase.

Formal tools don't require a supporting testbench or test vectors to verify design, which means they can be deployed much earlier in the design phase. As soon as block-level RTL is ready, formal tools can be used to conform behavior to specifications. Unfortunately, at present, formal verification methodologies cannot prove correctness of the entire OpenSPARC chip. The possible combinations of the state of all the memory elements in the design make such proof incalculable.

But while commercial tools have come a long way, they still raise capacity and performance concerns. So a generic awareness of where formal functional verification applies is useful. Unlike other tools, it cannot be applied to all blocks or to a full-chip model. Therefore, understanding the characteristics of areas with high formal applicability (such as blocks, portions, or functionalities) will yield the greatest return on the time and effort invested.

Formal tools efficiently verify a protocol, find deadlock or livelock conditions, make sure that an arbiter never misses, ensure that priorities aren't violated, or make sure that a FIFO is never driven when full. They are the only automated ways to conclusively establish the absence of bugs in design or protocol. Design equivalency comparison and property checking are the main applications of formal techniques.

7.5.1 Design Comparison

With so many levels of design abstractions created during design development, determining functional equivalence is fundamental to the entire verification process. For example, RTL and gate-level models of the design must be checked against each other for functional equivalence. Formal equivalency tools are now widely used, for example,

- To compare RTL and gate-level netlists after synthesis
- To compare two gate-level designs after technology mapping or after manual or automated optimization
- To compare designs before and after scan insertion
- To check the correctness of EDA tools such as synthesis and language translators

Functional equivalency tools can be classified as logic checking or sequential design comparison. Logical equivalency can only be established when two designs have the same storage elements. The corresponding inputs to the storage elements and outputs of the design are compared in both designs. Both the RTL and gate models have sequential (FF/latch) elements, and all the logic that fans into a point of comparison is referred to as a fan-in cone or cone-of-logic. A correspondence between the storage elements of the design is either provided by the user or inferred by the tool. Then for each corresponding pair of storage elements and primary outputs, the respective logic functions are compared. If the comparison is proved to be equivalent, then the designs will behave the same.

A presupposition for logic checking is that the storage elements of the circuit are in the same positions of both the designs, or in other words, that the circuit state is encoded the same way. This may not always be the case, as, for example, with retimed logic. The same circuit state encoding is achieved by partitioning of the design such that many sequentially equivalent points in the design could be used to compare sequential circuits. Property checking tools are helpful in this area.

7.5.2 Property, or Model, Checking

Property checking, also referred to as model checking, proves whether the behavior of a design conforms to the specifications, which are expressed as a set of property descriptions. Tools increasingly support the standard languages such as System Verilog Assertions (SVA), which can also be leveraged in simulation.

There are two main approaches to property checking: Theorem Proving and Model Checking. The Theorem Proving approach lets users state theorems in a general way. A highly trained user then attempts to synthesize a proof-of-theorem by using a combination of decision procedures and heuristics. This approach is strictly limited to verification of custom specifications.

On the other hand, the Model Checking approach, which is more popular, is based on moderately expressible logic dealing with finite state machines and is reasonably automated. Traditionally, most model checkers use a static formal analysis approach. Static property-checking tools give exhaustive proofs; they prove that a specified property is either valid or invalid. They sometimes don't reach a conclusion in a reasonable time and end up with an incomplete proof, in which case the user intervenes to modify the property, increase the constraints, or repartition the design. Static checking tools are generally severely constrained by capacity and performance.

The advent of a new class of model-checking tools that employ static analysis in conjunction with simulation (also called dynamic or hybrid engines) has broadened the adoption of property-checking tools. To use these tools, engineers write properties expressed as assertions such as SVA. These dynamic formal tools generally don't provide exhaustive proofs but greatly improve overall efficiency in bug hunting. Users predominantly use this tool to track down bugs and to augment coverage by reaching state space where simulation alone fails.

7.5.3 Symbolic Simulation

Symbolic simulation (also called symbolic trajectory evaluation) allows simulation with three types of values: Boolean values (logic 0 or logic 1), symbolic variables (like variables in an algebraic formula), and X values. Such values are initially assigned to internal storage nodes of a circuit. Then, for a finite number of clock cycles, symbolic inputs are introduced into the circuit. Instead of just logic 0, logic 1, X being on a node (as in a traditional simulator), a formula is computed for the node when the circuit stabilizes. Such nodes can then be checked for correctness.

A distinguishable strength of this approach is that it does not require the design to be synthesizable; it can therefore be used with switch-level models as with gates or the behavioral RTL model. This technique has effectively been used to verify the correctness of memory arrays.

7.6 Emulation Verification

A system-level design encompassing the RTL design of the processor, memory sub-system, system components, and a supporting testbench results in a massively large design that poses a limitation to all tools, particularly simulation and formal verification. For example, the system-level model of OpenSPARC T1 exceeded 35M gates. Traditional Verilog software simulators come to a crawl (down to under 10 simulated cycles per second), and for formal verification tools, the design state-space explosion prevents completion of formal proofs. FIGURE 7-7 depicts a simplified bug-find rate graph.

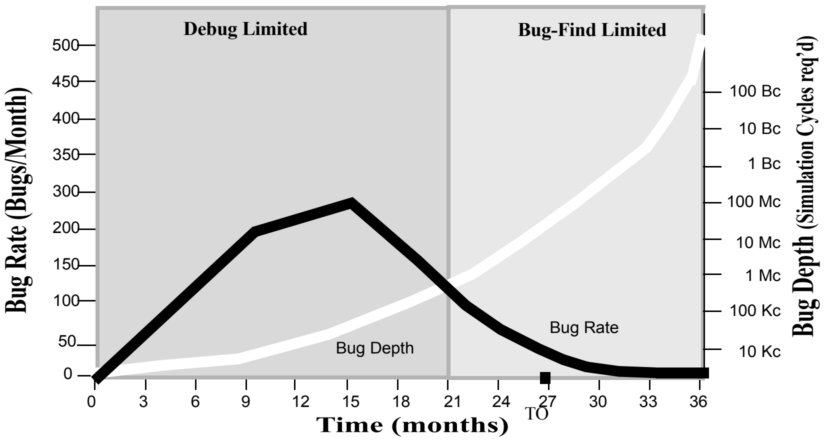


FIGURE 7-7 Simplified Bug-Find Rate and Bug Depth Relative to Design Maturity

Early on in the design phase as functionally is added, the number of errors introduced into the design rises. But since these errors can easily be found with a short simulation, they are quickly fixed. As the design matures, however, a significantly higher number of cycles must be run in order to find the few remaining complex corner-case errors. There is an exponential increase in bug depth. At this time, bug-finding is limited because design errors cannot be found easily. Emulation technology steps in here to facilitate prototyping design to produce virtual silicon. This virtual silicon, though much slower than real silicon, runs at least 10,000 times faster than a design in a software simulator.

Emulation facilitates early system integration even before the design is taped out, thereby allowing concurrent hardware and software development and testing. Traditionally, designers must wait for silicon before they can integrate the system (hardware-software), as shown in FIGURE 7-8. High-speed verification bandwidth made available by prototyping the design with emulation enables concurrent engineering, as shown in FIGURE 7-8. Concurrent engineering results in significant savings in reduced design development time by integrating and testing hardware and software early and in cost savings by reducing the number of silicon respins required before the final product is shipped.

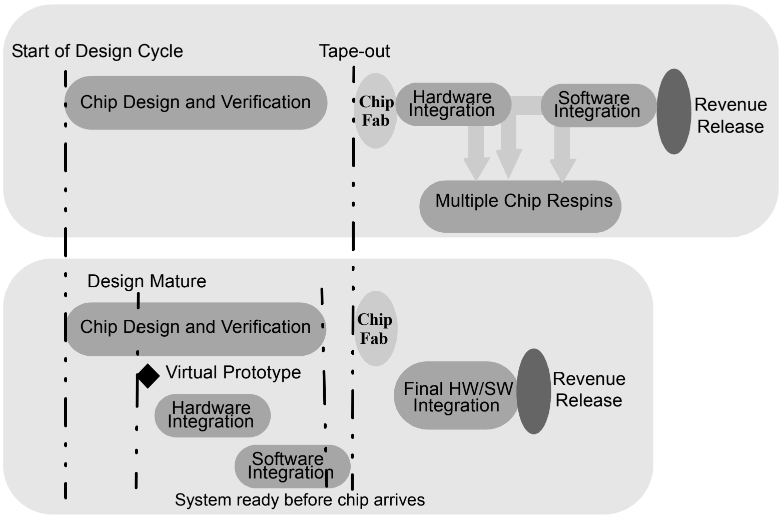


FIGURE 7-8 Concurrent Engineering Enabled by Design Prototyping With Emulation
Top: Traditional Sequential Verification Flow. *Bottom:* Concurrent Verification Flow

Acceleration and emulation are similar in concept and therefore, the terms are often used interchangeably. However, the two are subtly different. Emulation provides an ICE (in-circuit emulation) interface to a target system, whereas acceleration does not and functions only as a high-speed simulator. An accelerator also allows close coupling of a software simulation engine to the hardware platform, so modules that are not synthesizable can be placed and simulated outside the hardware box (this, of course, causes a slowdown). This use mode facilitates initial model bringup and also aids design debug.

These days, however, hardware platforms are becoming more versatile and function as both accelerator and emulator. This is especially useful because one platform can meet varying verification needs as the design progresses. In this chapter, emulation is used to mean acceleration as well. There are various applications of the hardware acceleration or emulation technology, but the primary application is to prevent functional bug escapes in design.

Note: The difference between accelerators and emulators is this: Accelerators are simply high-speed simulators, providing a close-coupled software-simulator engine along with a hardware-simulation engine. Thus, non-synthesizable modules or reference models can simulate in sync with the hardware design. Emulators, on the other hand, require design and related verification infrastructure to be fully synthesizable. Emulators also provide an interface to in-circuit-emulation in real-world target systems.

At Sun, as previously noted, the design is tested with both directed and random tests. Hardware emulation for test cases, such as RAS (reliability, availability, and serviceability), I/O testing, on-chip debug verification, DFT (design for test), scan testing, reset testing, and gate-level simulation, adds another layer of confidence about the design. To facilitate early system integration and software testing, Hypervisor, Open Boot PROM (OBP), and even Solaris OS are booted even before the design tapes out. These software applications are good candidates to ensure that performance benchmarks are met and that user applications run on the virtual silicon. Hardware acceleration technology also aids post-silicon bring-up, rapid debug analysis, and fix validation. Thus, emulation technology plays a key role in ensuring functional first silicon success, thereby greatly reducing number of silicon respins.

The adoption of an emulation-based technology definitely involves a few changes in the verification flow—changes in RTL design, testbench design, assertion strategy, monitor implementation, model release, regression, and debug processes. It pays to make the team aware of requirements up front so that everything intended for this platform is emulation friendly.

Emulation requires that the model to be emulated in the hardware platform must be synthesizable. A typical processor design has big custom-design blocks in addition to standard synthesizable blocks. The custom blocks are usually modeled as behavioral RTL in a standard simulation environment. An equivalent synthesizable RTL model must be developed for these custom blocks and for all the custom memory in the design.

The testbench, which is never written to be synthesized in a standard simulation environment, presents one of the biggest challenges in developing a synthesizable equivalent. In most cases, the end result is a simplified testbench that is made synthesizable and that is targeted for the emulation environment.

Some acceleration environments allow for non-synthesizable components (design blocks or testbench) to be simulated in a closely coupled software simulator that works in tandem with the hardware platform. Obviously, in this case, the slowest component limits the speed (Amdahl's Law), usually in the range of 5 to 100 times faster, relative to a software simulator.

7.6.1 Emulation Platforms

A number of emulation platforms in the marketplace enable virtual prototyping of design. FIGURE 7-9 shows some of them along with their relative performance data.

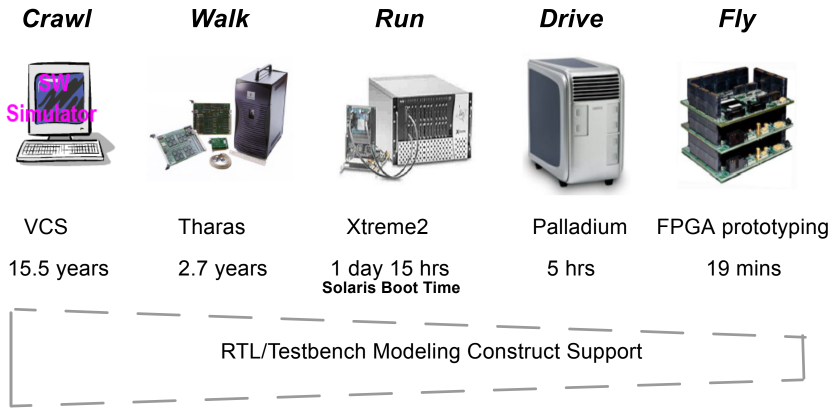


FIGURE 7-9 Simulation Platforms

Also shown in FIGURE 7-9 is the relative time required to boot a typical operating system such as Solaris (assuming ~8 billion simulation cycles are required to boot Solaris). This speed gain is at a cost of lost flexibility in the RTL, testbench, and verification infrastructure (monitors, assertions, coverage) constructs supported in these platforms.

Many of the platforms illustrated in the figure are expensive and may not be a viable prototyping option for all projects. Sometimes when the RTL model is not available or when a high degree of accuracy afforded by the hardware model is not required, hardware functionality can be modeled with high-level languages such as C, System C, or ESL (Electronic System Level). These hardware models can be cosimulated with rest of the hardware design. TABLE 7-2 shows the trade-off analysis with respect to mainstream RTL prototyping alternatives such as modeling in C, emulation, and FPGA-based prototyping.

TABLE 7-2 Hardware and Software Cosimulation Options

	Abstraction in C	**FPGA Prototyping**	**Hardware Emulation**
RTL verification	No	Yes	Yes
Debugging features	Good	Limited	Good
Bringup effort	Days	Weeks	Days
Model turnaround	Hours	Days	Hours
Timing concept	No	Yes	Yes
Modeling accuracy	Low	High	High
Runtime speed	High	High	Low
Capacity	Unlimited	Low	High
Real target system	No	Yes	Yes
Cost	Low	Medium	Very High

The marketplace reveals two distinct categories of high-performance acceleration/emulation platforms. The first category is based on field-programmable gate array (FPGA) technology. As the name suggests, FPGAs are used as a medium to accelerate model simulation, and the design under test needs to be synthesized and mapped onto FPGAs before simulation. The simulation performance is limited primarily by the FPGA speed. The other category is processor-based technology, by which hundreds and thousands of custom mini-processors concurrently simulate the model. At Sun, both platforms are used for different projects. Because of major consolidations, only a few companies, namely, Cadence, Mentor, and Eve, remain to provide acceleration and emulation hardware in EDA.

7.6.2 Emulation Deployment

Emulation can begin as early as the RTL is available, though it pays to wait until the model is mature enough to run long simulation cycles without too many quick failure interruptions.

As the full-chip model matures, acceleration is begun in cosimulation mode. In this mode, the synthesizable part of the RTL model is mapped to the emulation platform, while the non-synthesizable components, such as the behavioral blocks and the testbench, reside in the software simulator that works in tandem with the hardware engine. The software simulator is also coupled to the ISS so that cycle-by-cycle result checking is performed as the test is run.

Once the model matures and developers are no longer debug limited for the shorter tests, a fully synthesizable model (including the testbench) is targeted to the emulation platform. For highest speed-up, also called targetless emulation, everything is placed in the hardware box. Here, long random tests generated offline are run. These tests have some form of built-in result checking or are analyzed post-simulation for failures.

Once the bug depth (cycles-to-failure) goes up drastically, random SPARC instruction-based generators that self-boot on the emulator generate different random perturbations, run them, and finally self-check the results. These types of generators run literally trillions of simulation cycles, targeting the deep corner-case and its hard-to-find bugs. The majority of hardware verification applications are deployed in this targetless emulation mode.

In-Circuit-Emulation (ICE) mode is also used to successfully debug and bring up new motherboards even before silicon is received from the foundry. In this mode, the entire design and key components are modeled in the emulation box, which is then interfaced to an existing motherboard.

The speed gap between the fast side on the motherboard (or a target system) and the slow side, the emulator, must be addressed, so the ICE setup described above can be exploited to verify early firmware and the software stack in a completely new system motherboard. This emulator can also be interfaced to the parallel port of a workstation to run Perl scripts to manipulate the JTAG interface on the emulator for scan verification and other DFT-style verification tasks.

It is no surprise that emulation platforms are expensive. Therefore, to be able to successfully deploy this technology, one must plan early to ensure its success. Here are a few factors to which we attribute our successful emulation deployment on OpenSPARC:

- **Set up the emulation environment early.** Create awareness of technology benefits and limitations. Scope capacity and usage modes. Minimize the learning curve by integrating into an existing simulation flow. Not everything can be run on the emulator; therefore, prioritize and plan usage.
- **Minimize modeling issues.** Array/memory models need to be abstracted to a higher level. Eliminate nonfunctional models (for example, PLL, SerDes); abstract low-level circuit details to optimize for capacity and performance.

- **Reduce the time-to-model build.** Emulation models have to stay in sync with the mainstream model to be of value. Enable emulation-based RTL checks for release enforcement.
- **Simplify debugging.** Implement a critical set of debug monitors. Invest in debugging tools to improve debugging productivity in the project cycle.
- **Influence paradigm change.** Don't use it as just another simulator. Explore innovative uses of emulation models to enable concurrent verification of software and hardware. Deploy the emulation platform to the team at large, rather than to a selective few experts.

7.7 Debugging

Debugging is the process of identifying the root cause of a test failure. With verification being the long-pole in the process of designing and delivering OpenSPARC, within verification, debugging failures is the long pole. Debugging is one of the most manually intensive processes in design development, and not everyone is skilled in debugging a test from start to finish. Usually, a verification team is more debug limited than bug-find limited.

An error in the specification, RTL design, module interface, testbench, assertions, monitors, test, other simulation artifacts and supporting tools in the simulation environment can show up as a test failure. In a software-based simulation environment, all wires, state elements (FF, latches) and memories in design can be seen. Furthermore, signal trace mechanisms can track the state of wires over the entire simulation, so signals can be viewed and analyzed post-simulation, as with a logic analyzer. Even with powerful tracing tools, the human ability to absorb enough information about the design and state changes that uncover the bug will limit how quickly the root cause of these bugs can be determined.

System-level debugging in general presents a big challenge. Many variables related to hardware and software that play a role in the system-level verification can go wrong. Couple this with the fact that billions of cycles must be simulated to expose deep corner-case functional issues, and the reasons for the complexity of system-level debugging become obvious. As the design matures, the bug depth (which increases with design complexity) exponentially increases. Bug depth is typically a measure of simulation cycles required to set up and saturate design state and to expose deep corner-case bugs.

Debugging presents an even greater challenge in emulation. In part, this challenge stems from the lack of all monitors, from assertions, and from limited time on the emulation platform. Not all monitors or assertions can be ported from the simulation environment, either because emulation capacity is constrained or because non-synthesizable constructs are used in the simulation environment. Since many tests take several days and fail after multimillion cycles in the simulation, rerunning them is not an option. Also, generating waveforms, a primary debugging aid, would be a time-consuming proposition since most emulators greatly slow down during waveform generation. Emulation users must change their perspective and adapt to innovative techniques that improve debug efficiency.

A select class of assertions and critical debug monitors, also known as heartbeat monitors, can either be ported from simulation or specially crafted for the emulation environment. A limited set of `$display` commands should be used to log model activity in the emulation environment, since too many of these commands would simply slow the emulator. Instead of using `$display` to print often, some users adapt the monitors to store relevant information in a local memory and periodically dump the memory.

A debug engineer starts from a simulation log file to identify a large time window of failure. With the waveforms on-demand feature (there is no need to rerun the failing test), the simulation can quickly be restored to the nearest available checkpoint and then run to a specified time to generate waveforms for preidentified probe signals. This process is 100 times faster than generating full-hierarchical waveforms.

The waveforms for probe signal are analyzed to identify a more narrow time window for which a full-hierarchical signal waveform of design is obtained by means of a process similar to probe-waveform generation. These waveforms are then analyzed to pinpoint the failing RTL constructs. Expert emulation users also frequently write a set of complex logic-analyzer triggers to stop the emulation at the point of failure for interactive debugging.

All in all, debugging and uncovering design flaws require thorough overall understanding of the complete design and the ability to visualize how the design state changes over time.

One of the hardest debugging problems is attributing the symptoms of a failure to software or to hardware. In a system-level environment when the focus is a software issue, such as a Solaris OS boot debugging, debug information from the emulator must be relied on and then used in conjunction with the ISS to analyze the failure. Software debugging tools such as `gdb` or `dbx` are also used to debug software code. Close collaboration across cross-functional groups is the key to speedy analysis and resolution of a test failure.

7.8 Post-Silicon Verification

In general, considerable time and resources are spent during the pre-silicon verification phase to minimize functional issues before first silicon. For OpenSPARC T2, simulation, formal, and emulation technologies coupled with solid methodology covered all bases, ensuring the functional success of first silicon.

Nevertheless, a robust post-silicon verification methodology is critical to speeding up the time-to-ramp phase to prevent loss of product revenue. At Sun, not only are major post-silicon issues preempted before design tape-out, but planning and preparation deal with silicon issues that revolve around limited debug visibility, bug reproducibility, and bug-fix validation.

Reducing the time and complexity of verification during post-silicon verification can be managed by pulling in numerous silicon bring-up and system software and hardware verification tasks, made feasible by rapid prototyping with hardware emulation. Hardware emulation can reproduce silicon failures and can help debug them with higher productivity by providing high visibility and ease of waveform generation. Hardware emulation can also provide much-needed bandwidth (typically where traditional simulation runs out of gas) to run billions of cycles in a matter of hours to quickly boost confidence in the bug-fix validation. It also serves as a training vehicle for silicon bring-up, resulting in a greatly minimized post-silicon validation phase.

Formal verification has been used as one of the means to determine the root cause of silicon failure due to functional error. Since the RTL fix *must* be verified, it is vital that a failure observed in silicon test can effectively be reproduced and recreated in an RTL environment. One way of quickly recreating the RTL bug is to write the failure scenario in terms of property and use formal tools to generate traces that lead to the failure. Random test generators are important both in pre-silicon and post-silicon testing of chips. A variety of random test generators are used extensively in both areas for the verification of SPARC microprocessors and microprocessor-based systems. Fault isolation is a crucial capability built into some of these test generators from the ground up.

Once silicon arrives, the verification void, resulting from the range of tools used during pre-silicon verification, can be filled. Real-world applications can now be run on the actual server system. Reduced visibility, generally limited to latches and architectural state, poses a significant debugging challenge. Repeatability of failures is also an issue. Debugging of these complex failures involves considerable time and detailed system know-how.

7.8.1 Silicon Validation

After the chip manufacturer returns the silicon, tests are run to see if the silicon works. First the chip testers that use scan and built-in self-test (BIST) must confirm that the chip passes the tests. Low-level capabilities of the chip are verified here, including the various kinds of resets, scan chains, BISTs, JTAG access to the chip, and clocking.

Once things look good on the tester, chips are put into computer systems. The first software that runs on the new systems is power-on self-test (POST). POST performs cursory, as well as detailed, testing of the chip. With the use of functional and so-called diagnostic access, the integer registers, floating-point registers, caches, Translation Lookaside Buffers (TLBs), and other assorted arrays should work correctly. Diagnostic access provides ways to examine hardware state directly in order to explicitly test hardware function. For example, from a purely application-programming level, the fact that the system has instruction, data, and level-2 caches is completely invisible to software. Attempts to explore cache function with very specific code and hardware timing mechanisms can be made, but application-level software cannot directly examine cache state.

Using diagnostic accesses, which require special software permissions to use, POST can read cache tag valids, tags, data (or instructions), and any parity or ECC the cache provides.

After POST runs successfully, another layer of software, called OpenBoot, is run. OpenBoot abstracts hardware-specific implementations of memory configuration, I/O device configuration, and other hardware system resources an operating system would need to know about. The purpose of this level of abstraction is to make widely different computer system implementations look the same from the operating system perspective. This technique limits the changes needed when moving Solaris OS from system to system.

Between POST and OpenBoot, Hypervisor (HV) software is started. HV is code that runs at the highest permission level (hyperprivilege) the hardware allows. The purpose of the Hypervisor is to allow an abstraction layer of hardware behavior to again buffer Solaris OS from specific system configurations. Unlike OpenBoot though, the Hypervisor software intercepts real-time hardware events and maps them from the hardware implementation to an interface that Solaris OS knows about and that is common on all Sun computer systems.

One example of this kind of abstraction is how the system handles parity or ECC errors. The implementation of a specific hardware-state-protection mechanism, such as an I-cache having parity protection, is quite likely different on different Sun processor designs. When hardware state errors are detected, Hypervisor code takes a trap and executes code to examine and, where possible, recover from the error. Hypervisor then converts this error to an error event that it delivers to Solaris OS. The error event and its communication mechanism are the same on all Sun systems. Like OpenBoot, the Hypervisor code manages all hardware implementation differences on different systems through a common interface for Solaris OS.

OpenBoot and Hypervisor exist to make Solaris OS easier to move onto new Sun computer system designs. This also has the positive affect of minimizing the need for engineers that are adept at both software and hardware. Once Hypervisor and OpenBoot run, Solaris or verification software can be booted and run on the system.

Given all the verification done in the pre-silicon phase, it might be expected that all the system bugs had been found in that phase. While few test escapes occur on first silicon, bugs still exist in the chips and must be exposed, analyzed, and fixed in this post-silicon system validation phase.

Even if trillions of verification cycles were run over years of pre-silicon verification effort, this compares to only minutes of real-time testing on an actual server machine. Much more verification has to be performed to achieve high-quality products. Therefore, a number of systems could be assembled with the new chips and made available for real-time testing. These real-server hardware platforms would open up testing opportunities and allow new testing methods to a degree that is not possible in pre-silicon by utilizing simulation platforms alone.

As described earlier, at Sun extensive testing of POST, OpenBoot, Hypervisor, and Solaris OS was done in pre-silicon, but most of this verification was done on abstracted versions of the OpenSPARC processor. These four levels of software were run, and Solaris OS successfully booted on emulators. But even the emulation model is an abstraction of the final chip, in that it assumes perfectly functioning transistors providing perfectly functioning digital function.

A physical design team at Sun mapped the well-tested logical design to transistors and metal interconnect to produce the real chip implementation. Much work was done to confirm that the physical representation implemented the logical design, yet there can be unknown electrical side effects in the physical design that show up only when the chip returns from fabrication.

Until software that can run for long periods of time can be booted, no one can be really sure that the chip works. Booting Solaris OS and running applications, however, is one way to confirm that the chip works. Also, Sun's verification software runs as applications under Solaris OS to confirm that the chip functions correctly. Also, like POST, OpenBoot, and Hypervisor, some Sun tests run directly on the hardware (rather than as an application under Solaris OS). These tests generally use random numbers to invoke various architectural features for testing, as shown in FIGURE 7-10, which describes the silicon validation platform used for the OpenSPARC SoC.

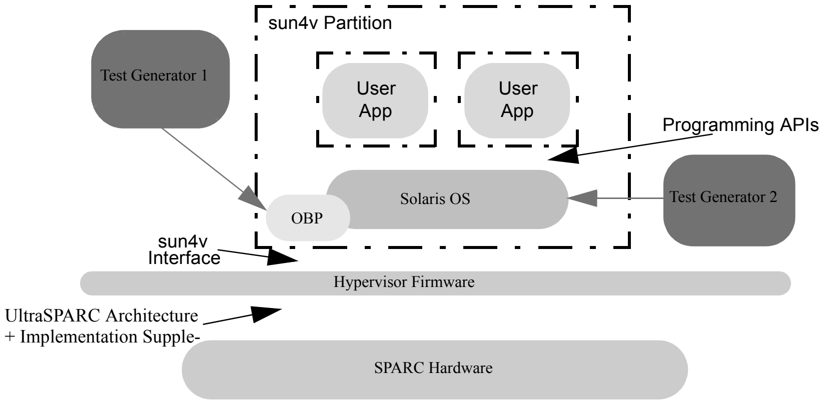


FIGURE 7-10 Silicon Validation Platform for OpenSPARC SoC

Solaris, its applications and tests, combined with tests that run without Solaris OS must be run for months to be sure that new computer systems are of the high quality that customers demand. When pre-silicon test escapes are found, the root cause of the escape must be found and determination as to whether it is fixable or whether it can be worked around must be made. If the root cause can be remedied, then the fix is implemented and tested in pre-silicon. Proper function of the fix is checked by formal methods, as well as by the classic simulation-based methods, with new tests devised to uncover the original bug.

7.8.2 Silicon Debugging

Post-silicon debugging is needed when the combination of software and hardware does not behave correctly. In pre-silicon simulation, every wire in the system can be queried at any time during the simulation to analyze problems and to determine their root cause.

This perfect visibility is not available post-silicon. Visibility into the hardware state is extremely limited in post-silicon debugging. Mechanisms can stop clocks and scan out the state of the chip, but in reality these mechanisms are difficult to use in the search for a problem's root cause. This biggest single issue is that unlike simulators, debuggers cannot stop time. Often this means that by the time software recognizes a problem, the specific state of the machine that uncovers the problem is long gone.

Often the first sign of a potential problem is visible only through the side affects of the bug. Rarely is the problem simple enough for even an expert to know directly what is wrong with the hardware. Some hardware support implemented in the chip can aid debugging. One primary mechanism is access through JTAG. JTAG access allow querying of specific hardware registers. It is not cost effective to have all hardware state accessible through JTAG, so only registers needed for system configuration, reset, error detection enabling, and error detection reporting are typically visible through JTAG.

Another debugging mechanism is a logic analyzer interface. Through experience, developers probably know what units in the design can be problematic. The interfaces of these units can be selected and routed to the logic analyzer interface. If a problem is encountered and the units of interest can be observed through the logic analyzer interface, then the developers are in luck.

The most ubiquitous debugging mechanism is software itself. Typically, software is affected by some misunderstanding of how the hardware works and how the software is supposed to use it. Making changes to the software to record more information in and around the issue and making the results visible to the human debugger can resolve most of the issues.

The reality is that most of the time, issues are software problems, not hardware problems. The reason is that often the documentation that software coders use does not adequately describe the hardware functions. Analysis should be done to verify how the hardware is designed to work and how software should use it. The documentation should then be updated to express this new understanding. At the same time, software must be modified to work properly with the hardware design.

A much less frequent case is that a hardware problem really does exist. Whenever possible, software must work around hardware problems. Sometimes the workarounds are simple, but often they are not.

If a hardware design issue is one that could be visible to the customer and software cannot work around it, then a hardware fix must be designed and verified and new chips made. This process can take a long time compared to software changes and software workarounds.

One of the biggest factors contributing to the time required for post-silicon verification is how long it takes to identify a bug and to understand the root cause. Whenever possible, software should be designed to help recreate the problems so that the issue can be resolved more quickly. Any design issues discovered this late in the product development cycle can greatly impact schedules and revenue. A reverse example is the OpenSPARC designs; they were well engineered and effectively verified before design tape-out, so silicon issues were minimized, resulting in faster time-to-market for the Sun Fire servers.

7.8.3 Silicon Bug-Fix Verification

Once a hardware design flaw is analyzed and understood, potential fixes must be explored. In some cases, fixes can be implemented with only a change to the way in which the transistors are wired together. When there is spare area on a chip, unused transistors often fill the spare space. Sometimes these spare transistors, combined with new metal interconnects, can be used to fix a problem. Fixes of this nature are one of the least expensive ways to fix the problem.

When “metal only” fixes are not achievable, the expense of the fix increases dramatically. Nonmetal fixes require new physical layout of the transistors on the chip and new metal interconnects to hook them together. Furthermore, the chip foundry must create new silicon wafers with the new transistor layouts, which is a costly endeavor.

No matter what the form of the final fix is, a number of precautions must be taken before the fix is implemented. First it must be confirmed that the fix actually solves the root problem, and that is not always easy to determine. The confirmation comes in at least two forms: detailed formal verification analysis of the problem area; and proof by the fix that it works. Simulation of the fix against newly written tests can also verify the fix. Precautions must be taken to ensure that fix does not have a functional or performance impact on the overall design.

Once a fix is chosen and implemented in the logical design, a full regression of simulation is run to confirm the fix and its impacts on the system. Additionally, high-speed emulation of billions of verification cycles must be run in a matter of hours to establish the necessary confidence. At the same time, the physical design team should implement the fix so that it can be communicated to the chip manufacturer.

Often these fixes are held by the organization so that potential problems and their fixes can be grouped into one new chip revision. Once the schedule or problem severity mandates that a new chip version be made, the new chip specification is communicated to the chip manufacturer. When the new chip version is ready, the chips are shipped to the organization, and post-silicon verification starts again.

When post-silicon verification is complete, the system is ready to be manufactured and sold to customers.

7.9 Summary

The revolutionary OpenSPARC T1 and T2 processors implement the industry's most aggressive chip-level multithreading (CMT) to exploit the processor-memory speed gap in order to provide world-class performance. Traditional verification methodologies are not enough to meet the daunting verification complexity of these new OpenSPARC SoCs. Implementation of a sound verification strategy with a well-orchestrated suite of state-of-art commercial and internal tools are critical to ensure verification success of these designs.

It is important that the primary emphasis should be on detecting and fixing the majority of functional verification issues, thus preempting as many post-silicon issues as possible before design tape-out. Since it is nearly impossible to exhaustively verify, the key is to root out enough bugs in the first level of pre-silicon verification so as to enhance the value of first silicon and enable a progressive second level of silicon validation in an actual target system environment. A secondary emphasis should be on preparing to deal with limited visibility and the debugging challenges of issues found in silicon. This is key to reducing the time-to-ramp and the general release of the servers in which these SoCs will function.

Because of finite resources and time available to ensure the highest-quality products, focus must be on verification efficiency throughout the design phase. This is even more important because verification tools have not kept pace with the exponential increase in design size and complexity. Sheer size and the complexity of these SoCs push the capacity and performance envelope on these tools.

Simulation, formal verification, and emulation technologies play a major role. Use of assertions improves debugging productivity, while use of coverage objects helps improve overall verification efficiency. Both simulation and formal tools play a major role at the block and full-chip level, but they simply run out of gas during system integration, a critical phase in the product development. Here, emulation technology can generate a virtual prototype that provides the much-needed simulation bandwidth to accomplish such cycle-intensive tasks as software and hardware integration. Emulation technology can also perform many tasks usually possible only after the arrival of silicon. Once silicon arrives, it is deployed in numerous systems to run real-world applications in order to root out last few bugs before shipping a quality product.

Numerous innovative techniques that span multiple verification platforms were deployed to enable concurrent engineering throughout the project. The paradigm of concurrent verification greatly reduced development cost and shortened the product development cycle, thus helping developers get hardware and software right the first time on our Sun Fire servers driven by OpenSPARC Server-on-a-Chip (SoC).

CHAPTER 8

Operating Systems for OpenSPARC T1

As low-end hardware systems become more powerful, underutilization of the hardware system is an issue for some applications. Virtualization technology allows consolidation of applications running on multiple underutilized hardware systems onto one hardware system. The multiple hardware systems that are consolidated could be running different versions of the same operating system or different operating systems. Virtualization technology decouples the operating system (OS) from the hardware system and allows multiple OS environments (Solaris, Linux, Windows, etc.) to run on one host hardware system.

This chapter presents details about virtualization in the OpenSPARC T1 OS in the following sections:

8.1 Virtualization

A thin layer of software running on the hardware system presents a virtual system to each guest OS running on the host hardware system. The software presenting the virtual system view to the guest OS is called the Hypervisor. The virtual system is defined by the hardware registers and a software API to access system services provided by the Hypervisor. The Hypervisor uses hardware virtualization extensions to provide protection between the virtual systems. The hardware resources in the system like memory, CPUs, and I/O devices are partitioned and allocated to virtual systems.

8.2 sun4v Architecture

OpenSPARC T1 is the first SPARC processor to implement hardware extensions to support the Hypervisor. The virtual system model presented to the guest OS is referred to as the sun4v architecture. The hardware resources in the virtual system are specified with a data structure called machine description.

8.3 SPARC Processor Extensions

OpenSPARC, in addition to privileged mode and user mode execution, supports hyperprivileged mode execution. Only software running in hyperprivileged mode can access registers that manage hardware resources like memory, CPUs, and I/O devices. The Hypervisor software runs in hyperprivileged mode. The OS runs in privileged mode and manages hardware resources in its virtual system by making Hypervisor API calls. The OS cannot directly manage hardware resources bypassing the Hypervisor.

OpenSPARC extends Tcc trap instructions to allow software trap numbers 80_{16} and above. Similar to user-mode processes trapping into the OS to request OS services, the OS running in privileged mode traps into the Hypervisor to make Hypervisor API calls using the Tcc trap instruction extensions. Only software running in privileged mode can execute the new Tcc trap instruction extensions. Processes running on top of the OS cannot trap into the Hypervisor from user mode.

Each virtual system is identified by a partition identifier value (PID). Each CPU in the system has a 3-bit partition identifier register, and the Hypervisor initializes the partition identifier register of a CPU with the PID of the virtual system to which the CPU belongs. Therefore, at most eight virtual systems can be supported.

OpenSPARC, in addition to supporting virtual addresses (VA) and physical addresses (PA), supports real addresses (RA). The MMU supports RA-to-PA translation in addition to VA-to-PA translation. The MMU does not support VA-to-RA hardware translation.

The physical memory in the virtual system is described with real addresses. The guest OS accesses memory either directly with real addresses or through virtual addresses mapped to real addresses. When the guest OS requests the Hypervisor to set up a VA-to-RA MMU mapping (through a Hypervisor API call), the Hypervisor translates the RA to the physical address of the memory allocated to the virtual system. The Hypervisor then sets up a VA-to-PA MMU mapping. The MMU stores the PID of the virtual system in the TLB entry along with the VA, PA, and other information. This TLB entry is now associated with the virtual system that has inserted the mapping. When software running in a virtual system does memory operations, the MMU translations match only those TLB entries that have the same PID as the PID stored in the partition identifier register of the CPU. Therefore, a virtual system cannot access resources (memory, memory-mapped I/O device registers, etc.) mapped by TLB entries of another virtual system.

When a virtual system becomes active, one of the CPUs allocated to the virtual system is designated as the boot CPU and starts bringing up the guest OS. Eventually the guest OS reads the machine description of its virtual system and starts the other CPUs in its virtual system. The guest OS has to request the Hypervisor through Hypervisor API calls to start a CPU allocated to its virtual system. Before the CPU starts executing software in the virtual system, the Hypervisor initializes the PID register of the CPU with the PID of the virtual system. The Hypervisor API also supports a call for sending an interrupt from one CPU to another CPU within a virtual system.

The OS can access I/O device registers by requesting the Hypervisor to set up an MMU mapping from the virtual address of the guest OS to the physical address of the I/O device. A device driver written to the DDI/DDK interface need not be modified to run in an OpenSPARC system.

8.4 Operating System Porting

Porting an OS that already runs on non-sun4v SPARC systems requires making changes to the low-level software that reads hardware system configuration, accesses MMU registers to set up VA-to-PA translations, and enables and disables CPUs. Currently, the Solaris, OpenSolaris, Linux, and FreeBSD operating systems have been ported to OpenSPARC systems.

CHAPTER 9

Tools for Developers

This chapter presents an overview of some of the tools that are available for developing applications on SPARC platforms. The chapter reveals how to use these tools to tune performance and to debug applications. The chapter ends with a discussion on how CMT changes the game plan for developers.

These ideas are contained in the following sections:

9.1 Compiling Code

In this section you will learn about the following:

9.1.1 Compiling Applications With Sun Studio

The Sun Studio suite is a free download that includes compilers for C, C++, and Fortran. It also includes the debugger and Performance Analyzer, as well as various libraries (such as optimized mathematical libraries).

The Sun Studio compiler supports a wide range of features and optimization. In this text we focus on a small set of commonly used options. There are three common scenarios for developers: compiling an application for maximal debug, compiling an application with some optimization, and compiling an application with aggressive optimization. The flags representing these three different options are shown in TABLE 9-1.

TABLE 9-1 Compiler Flags for Three Developer Scenarios

Scenario	Debug Flag	Optimiza-tion Flag	Architecture	Address Space Size
Debug	`-g`	None	`-xtarget=generic`	`-m32` for 32-bit code `-m64` for 64-bit code
Optimized	`-g` (`-g0` for C++)	`-O`	`-xtarget=generic`	
Aggressively optimized	`-g` (`-g0` for C++)	`-fast`	`-xtarget=generic`	

The first thing to discuss is the flags that enable the generation of debug information. The debug information is used both when debugging the application and when the application is profiled. The debug information enables the tools to attribute runtime to individual lines of source code.

It is recommended that debug information is always generated. The flag that causes the compiler to generated debug information is `-g`. This flag will cause only minimal difference to the generated code, except in two situations. For code compiled with `-g` and no optimization flags, the compiler will produce code that has the maximal amount of debug information; this will included disabling some optimizations so that the resulting code is clearer. The other situation where `-g` may cause a significant difference to the generated code is for C++ applications. For C++ source files, the `-g` flag will disable some function inlining. For C++ codes, it is recommended that the `-g0` flag be used instead of `-g`; this flag generates debug information, but also enables the inlining optimizations.

When no optimization flags are specified, the compiler performs no optimization; this usually results in code that runs slower than might be expected. The `-O` optimization flag usually represents a good trade-off between compile time and the performance of the new code, so is normally a good initial level of optimization.

The `-fast` macro-flag enables a range of compiler optimizations that have been found to deliver good performance on a wide range of applications. However, this flag may not be appropriate for all applications since it makes some assumptions about the behavior of the application:

- The compiler can use floating-point simplification. The flags that enable this are `-fsimple=2`, which allows the compiler to reorder floating-point expressions, and `-fns`, which allows the application to flush subnormal numbers to zero. The flag will also cause the generated code not to set the `errno` and `matherr` variables for some mathematical function calls; typically, this happens when the compiler replaces the function call with either an equivalent instruction (for example, replacing the call to `sqrt` with the SPARC `fsqrt` instruction) or with a call to an optimized version of the library. Some applications are sensitive to floating-point optimizations, and for such, the `-fast` compiler flag may not be suitable.
- The `-fast` flag for C code includes the flag `-xalias_level=basic`, which allows the compiler to assume that pointers to different variable types (for example, pointers to ints and to floats) do not point to the same location in memory. Code that adheres to the C standard will also conform to this criterion, but there will be some codes for which this is not true.
- The `-fast` macro-flag also includes the flag `-xtarget=native`, which tells the compiler to assume that the system building the application is also the system on which the application will be run. In some cases, this assumption can produce an executable that will run poorly (or perhaps not at all) on other systems. This behavior is the reason for following the `-fast` flag with the flag `-xtarget=generic`, which undoes this and tells the compiler to build code that will run on the widest range of platforms. The compiler evaluates flags from left to right, so flags placed later on the command line will overrule flags that are earlier.

One way of using the `-fast` flag is to estimate the performance gain that might be obtained from aggressive optimization, and then select from the options enabled by `-fast` only those that are appropriate for the code and that contribute an improvement in performance.

For more on the selection of compiler flags, see the article at `http://developers.sun.com/solaris/articles/options.html`.

9.1.2 Compiling Applications With GCC for SPARC Systems

GCC for SPARC Systems is available as a free download from `http://cooltools.sunsource.net/gcc/`. It enables the GCC compiler to use the code generator from the Sun Studio compiler. This feature enables developers to use GCC-specific extensions in their code while still getting the features and performance of the Sun Studio compiler. TABLE 9-2 shows the equivalent flags for GCC for SPARC Systems.

TABLE 9-2 GCC Compiler Flags for Three Developer Scenarios

Scenario	Debug Flag	Optimization Flag	Architecture	Address Space Size
Debug	`-g`	None	`-xtarget=generic`	`-m32` for 32-bit code `-m64` for 64-bit code
Optimized	`-g`	`-O`	`-xtarget=generic`	
Aggressively optimized	`-g`	`-fast`	`-xtarget=generic`	

GCC for SPARC Systems supports optimization features of the Sun Studio compiler such as profile feedback, cross-file optimization, autoparallelization, binary optimization, data race detection, and others. These features are discussed later in this chapter.

9.1.3 Improving Performance With Profile Feedback

One of the techniques that can improve application performance is profile feedback. This technique is particularly appropriate for codes that have complex flow control. When the compiler encounters source code which has a branch that can be taken or not taken, the compiler either assumes that the branch will be taken and untaken with equal probability, or in some cases may be able to use information about the branch statement to estimate whether the branch will be taken or untaken with greater frequency. If the compiler can correctly determine the behavior of the branch, then it is often possible to reduce instruction count and perform other optimizations that result in improved performance.

The other situation in which knowledge of the control flow of an application helps the compiler is in deciding about the inlining of routines. The advantage of inlining routines is that it reduces the cost of calling the code; in some cases it may even expose further opportunity for optimizations. However, the disadvantage is that it can increase code size and therefore reduce the amount of code that fits into the cache.

Compiling with profile feedback enables the compiler to gather data about the typical code paths that an application executes and therefore give the compiler information that helps it to schedule the code optimally.

Using profile feedback is a three-step process. The first step is to produce an instrumented version of the application. The instrumented version is then run on a workload that represents the actual workload on which the application will usually be run. This training run gathers a set of data that is used in the third step, the final compilation to produce the release version of the application. An example is shown in CODE EXAMPLE 9-1.

CODE EXAMPLE 9-1 Process of Building an Application With Profile Feedback

```
$ cc -O -xprofile=collect:./profile -o test test.c
$ test training-workload
$ cc -O -xprofile=use:./profile -o test test.c
```

Profile feedback requires that the same compiler flags (with the exception of the `-xprofile` flag) are used for both the instrumented and final builds of the application. The `-xprofile` flag also takes as a parameter the location of the file where the data from the training workload is stored. It is recommended that this file be specified; otherwise, the compiler may not be able to locate the data file.

Several concerns about using profile feedback should be mentioned at this point:

- The process of compiling with profile feedback means two passes through the compiler plus a run of the application on training data. This can add to the complexity and time required to build the application. It is best to evaluate this cost against the performance gains realized through the process. It may be necessary to use profile feedback only on final rather than interim builds of the application.
- There is sometimes a question of whether the data used to train the application really represents the work that the application usually does. This topic is addressed in more detail in *Evaluating Training Data Quality* on page 142.

- A similar concern is whether training the application will result in code that performs better on one workload at the expense of a different workload. This concern is also addressed in the later section.

For more information on using profile feedback, see `http://developers.sun.com/solaris/articles/profeedback.html`.

9.1.4 Inlining for Cross-File Optimization

Cross-file optimization is the procedure by which routines from one source file are inlined into routines that are located in a different source file. This optimization is controlled with the compiler flag `-xipo`. The flag needs to be used for both the initial compilation of the source files and also at link time in order for the compiler to do cross-file optimization.

Cross-file optimization has the potential to effect performance gains in three ways:

- Inlining removes the cost of the call to the inlined routine, removing a control transfer instruction and resulting in more straight-line code.
- The compiler ends up with a larger block of instructions, and it may be possible to better schedule these instructions.
- When a routine is inlined it may expose opportunities for further optimizations. For example, one of the parameters passed into the routine may be a constant at the calling site, resulting in a simplification of the inlined code.

There are some downsides to inlining:

- The size of the code to be scheduled increases, possibly resulting in a larger number of active registers, which may result in registers being spilled to memory.
- The memory footprint of the routine will increase, and this may place more pressure on the caches. If it turns out that the inlined code was not necessary, a routine that once fitted into cache may no longer be able to fit.

Cross-file optimization is a natural fit with profile feedback since the training data set will help the compiler identify frequently executed routines that can appropriately be inlined.

9.1.5 Choosing TLB Page Sizes

The compiler has options that allow the developer to select the page size for the application at compile time. If these options are not used, Solaris will pick what it considers to be appropriate sizes. The page size for stack and heap can be set with the flag `-xpagesize=`*value*, where *value* is one of `8K`, `64K`, `512K`, or `4M`.

The rules determining appropriate the appropriate page size to select are relatively simple:

- Larger page sizes reduce the number of TLB misses because more memory will be mapped by a single TLB entry.
- However, it can be hard for Solaris to find the contiguous memory necessary to allocate a large page. So although the application may request large pages, the operating system may be unable to provide them.

The `cputrack` command can be used on an executable to count how many TLB misses are occurring for a given application. This information can then be used to see if changing the page size has any impact. An example of counting both the instructions executed and the number of data TLB misses is shown in CODE EXAMPLE 9-2. In this example, a total of 115 data TLB misses occurred in over 100 million instructions.

CODE EXAMPLE 9-2 Counting Data TLB Misses With `cputrack`

```
% cputrack -c DTLB_miss,Instr_cnt sub
   time  lwp      event       pic0      pic1
   1.021   1       tick         98  72823692
   1.726   1       exit        115 123562685
```

The command `pmap -xs` *pid* can be used to examine a process and check whether it obtained large pages or not. CODE EXAMPLE 9-3 shows the output from this command for the executable `a.out`. The heap for the target executable is mapped onto both 8-Kbyte and 64-Kbyte pages.

CODE EXAMPLE 9-3 Output From `pmap` Showing the Page Sizes for a Process

```
$ pmap -xs 6772
   6772: a.out
 Address   Kbytes  RSS   Anon  Locked Pgsz  Mode  Mapped File
00010000      8      8      -    -      8K   r-x--   a.out
00020000       8      8     8    -      8K   rwx--   a.out
00022000      56     56    56    -      8K   rwx--   [heap]
00030000  24512  24512  24512 -      64K   rwx--   [head]
...
```

9.2 Exploring Program Execution

The following subsections explore aspects of program execution:

- *Profiling With Performance Analyzer* on page 132
- *Gathering Instruction Counts With BIT* on page 137
- *Evaluating Training Data Quality* on page 142
- *Profiling With SPOT* on page 146
- *Debugging With `dbx`* on page 148
- *Using Discover to Locate Memory Access Errors* on page 151

9.2.1 Profiling With Performance Analyzer

One of the most important tools for developers is Performance Analyzer, included with Sun Studio, which generates profiles of applications showing where the runtime is being spent. Consider the code shown in CODE EXAMPLE 9-4.

CODE EXAMPLE 9-4 Example Program

```
#include <stdio.h>

double a[10*1024*1024];

void clear()
{
  for (int i=0;i<10*1024*1024;i++)
  {
    a[i]=0.0;
  }
}

double sum()
{
  double t=0;
  for (int i=0; i<10*1024*1024;i++)
  {
    t+=a[i];
  }
  return t;
}

void main()
{
  for (int i=0; i<10; i++)
```

CODE EXAMPLE 9-4 Example Program *(Continued)*

```
    {
      clear();
      sum();
    }
}
```

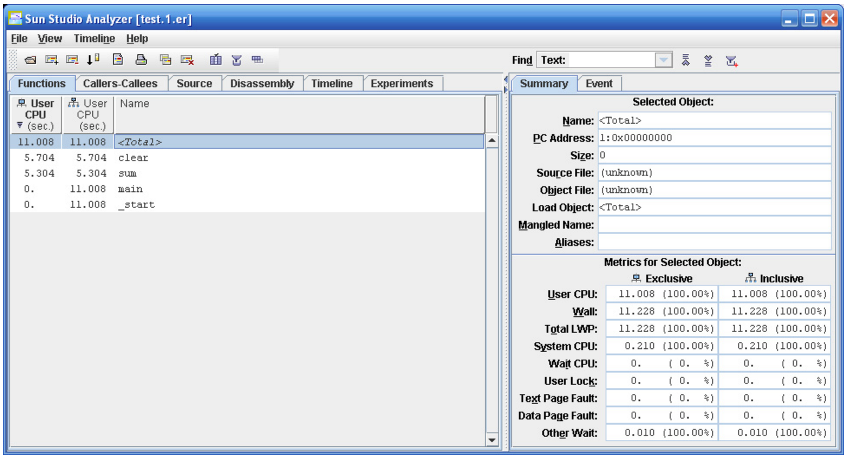
The tool that gathers the profile information is called `collect`. This tool can either be attached to a running process by `collect -P` *pid* or can follow the entire run of an application with `collect` *application parameters*. The profile data is gathered in a directory which is by default given the name `test.1.er`. Two tools can display the gathered data: the command-line tool called `er_print` or the GUI tool called `analyzer`. Either tool should be invoked with the name of the experiment that is to be loaded. The command-line version of the tool can also be invoked with commands to be executed or with a script to run.

The steps necessary to build, run, and view the profile in the GUI for the code from CODE EXAMPLE 9-4 are shown in CODE EXAMPLE 9-5.

CODE EXAMPLE 9-5 Compiling and Profiling Example Application

```
$ cc -g ex.9.4.c
$ collect a.out
Creating experiment database test.1.er ...
$ analyzer test.1.er
```

The initial view of the profile is shown in FIGURE 9-1.


Sun Studio Analyzer [test.1.er]
File View Timeline Help
Find Text:
Functions Callers-Callees Source Disassembly Timeline Experiments
Summary Event

User CPU (sec.)	User CPU (sec.)	Name
11.008	11.008	*<Total>*
5.704	5.704	clear
5.304	5.304	sum
0.	11.008	main
0.	11.008	_start

Selected Object:

Name:	<Total>
PC Address:	1:0x00000000
Size:	0
Source File:	(unknown)
Object File:	(unknown)
Load Object:	<Total>
Mangled Name:	
Aliases:	

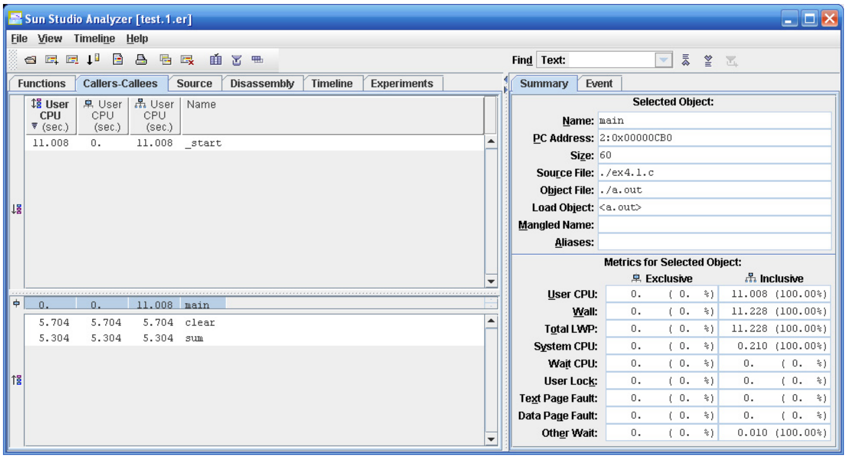
Metrics for Selected Object:

	Exclusive	Inclusive
User CPU:	11.008 (100.00%)	11.008 (100.00%)
Wall:	11.228 (100.00%)	11.228 (100.00%)
Total LWP:	11.228 (100.00%)	11.228 (100.00%)
System CPU:	0.210 (100.00%)	0.210 (100.00%)
Wait CPU:	0. (0. %)	0. (0. %)
User Lock:	0. (0. %)	0. (0. %)
Text Page Fault:	0. (0. %)	0. (0. %)
Data Page Fault:	0. (0. %)	0. (0. %)
Other Wait:	0.010 (100.00%)	0.010 (100.00%)

FIGURE 9-1 Profile From Performance Analyzer

The first two columns in the profile show the *exclusive* and *inclusive* user time. Exclusive time for a function is the time spent solely in that function. Inclusive time for a function is the time spent in that function plus any functions that it calls. For example, the function `main` has zero exclusive time—meaning that negligible time was spent in that function, but it has 11 seconds of inclusive time, meaning time that was spent in routines called by the `main` function. The routine `sum` accumulated 5.3 seconds of exclusive time, or time spent in that routine. The inclusive time for the routine `sum` is the same as the exclusive time because the routine called no other routines.

The tool also gathers call-stack information. This information allows the tool to calculate the inclusive time. FIGURE 9-2 shows the call stack for the routine `main`. The routine `main` is called by the routine `_start`, and it calls the routines `clear` and `sum`.


Sun Studio Analyzer [test.1.er]
File View Timeline Help
Functions Callers-Callees Source Disassembly Timeline Experiments
Find Text:
Summary Event

User CPU (sec.)	User CPU (sec.)	User CPU (sec.)	Name
11.008	0.	11.008	_start
0.	0.	11.008	main
5.704	5.704	5.704	clear
5.304	5.304	5.304	sum

Selected Object:

Name:	main
PC Address:	2:0x00000CB0
Size:	60
Source File:	./ex4.1.c
Object File:	./a.out
Load Object:	<a.out>
Mangled Name:	
Aliases:	

Metrics for Selected Object:

	Exclusive	Inclusive
User CPU:	0. (0. %)	11.008 (100.00%)
Wall:	0. (0. %)	11.228 (100.00%)
Total LWP:	0. (0. %)	11.228 (100.00%)
System CPU:	0. (0. %)	0.210 (100.00%)
Wait CPU:	0. (0. %)	0. (0. %)
User Lock:	0. (0. %)	0. (0. %)
Text Page Fault:	0. (0. %)	0. (0. %)
Data Page Fault:	0. (0. %)	0. (0. %)
Other Wait:	0. (0. %)	0.010 (100.00%)

FIGURE 9-2 Call Stack Information

The caller-callee chart introduces a metric called *attributed* time, which breaks down the time spent in a particular routine by the routines that call it. The attributed time, shown in the first column, represents the amount of time attributed to a particular routine when the selected routine is in the call stack. It is clearer to explain this concept using the data from FIGURE 9-2. The attributed time for the routine `_start` is 11 seconds, meaning that the entire 11 seconds of inclusive time for the `main` routines is attributable to its being called by `_start`. The attributed time for the routine `clear` is 5.7 seconds, which is the amount of time attributable to the `clear` routine from being called by the `main` routine. If the program were more complex and the routine `clear` were called from multiple locations, the inclusive and exclusive time for the `clear` routine would increase but the time attributable to its call from the `main` routine would remain the same.

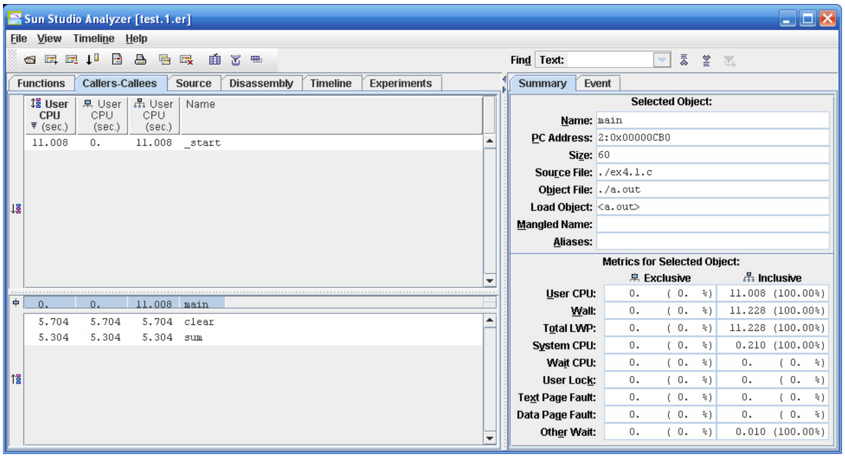
It is also possible to describe the attributed time as a formula—the sum of the attributed times for the calling routines will equal the inclusive time for the selected routine, as will the sum of the attributed time for the selected routine and the routines that it calls.

On hardware that supports it, Performance Analyzer can profile an application to show where the hardware counter events occur. For example, it can show the point in the code where the largest number of data cache misses occur. This feature is not supported on UltraSPARC T1 but is available on UltraSPARC T2. The option is supported by passing the flag `-h`, together with the list of counters to use, to `collect`. CODE EXAMPLE 9-6 shows the instructions necessary to profile the application to locate the places in the code where there are data cache misses; the data cache misses are recorded by the `DC_miss` counter.

CODE EXAMPLE 9-6 Profiling With the Hardware Performance Counters

```
$ collect -h DC_miss a.out
Creating experiment database test.2.er ...
$ analyzer test.2.er
```

FIGURE 9-3 shows the results of profiling data cache misses for the example code on an UltraSPARC T2 system. All the data cache misses occur in the `sum` routine; this is not surprising since the `sum` routine is streaming through memory, loading data. The `clear` routine is storing data to memory, so has no need to perform loads and incur data cache misses.


Sun Studio Analyzer [test.1.er]
File View Timeline Help
Find Text:
Functions Callers-Callees Source Disassembly Timeline Experiments
Summary Event

User CPU (sec.)	User CPU (sec.)	User CPU (sec.)	Name
11.008	0.	11.008	_start
0.	0.	11.008	main
5.704	5.704	5.704	clear
5.304	5.304	5.304	sum

Selected Object:

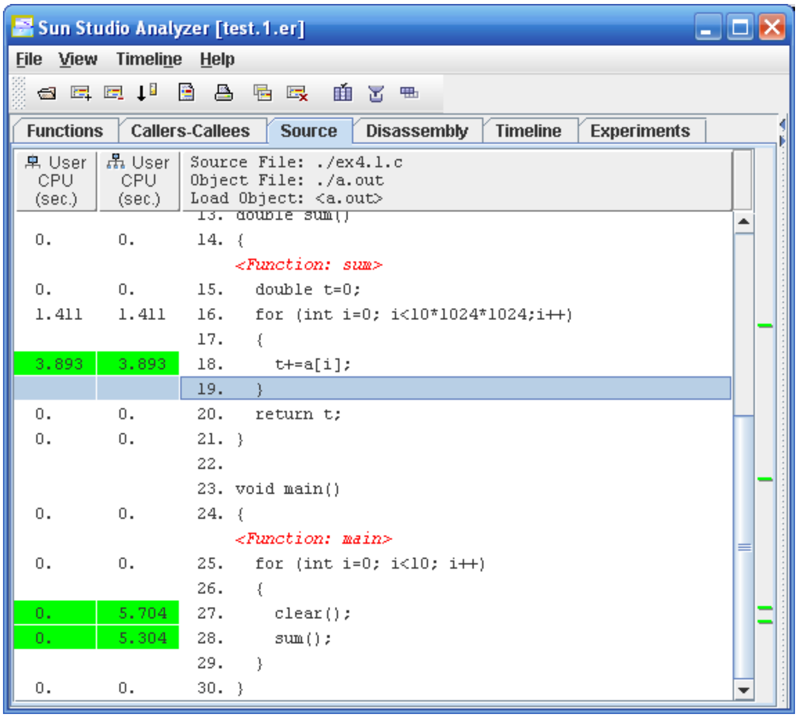
Name:	main
PC Address:	2:0x00000CB0
Size:	60
Source File:	./ex4.1.c
Object File:	./a.out
Load Object:	<a.out>
Mangled Name:	
Aliases:	

Metrics for Selected Object:

	Exclusive	Inclusive
User CPU:	0. (0. %)	11.008 (100.00%)
Wall:	0. (0. %)	11.228 (100.00%)
Total LWP:	0. (0. %)	11.228 (100.00%)
System CPU:	0. (0. %)	0.210 (100.00%)
Wait CPU:	0. (0. %)	0. (0. %)
User Lock:	0. (0. %)	0. (0. %)
Text Page Fault:	0. (0. %)	0. (0. %)
Data Page Fault:	0. (0. %)	0. (0. %)
Other Wait:	0. (0. %)	0.010 (100.00%)

FIGURE 9-3 Application Profiled by Location of Data Cache Misses

Source and disassembly tabs in the tool show the time attributed at the source code level (if the code has been compiled with `-g`) and at the disassembly level. FIGURE 9-4 shows the source-level view of time attributed to the routines `sum` and `main`. Once again the two columns show exclusive and inclusive user time. The routine `main` has two calls to other routines, and these calls have inclusive time attributed to them, but no exclusive time (since the call instruction takes effectively zero time).

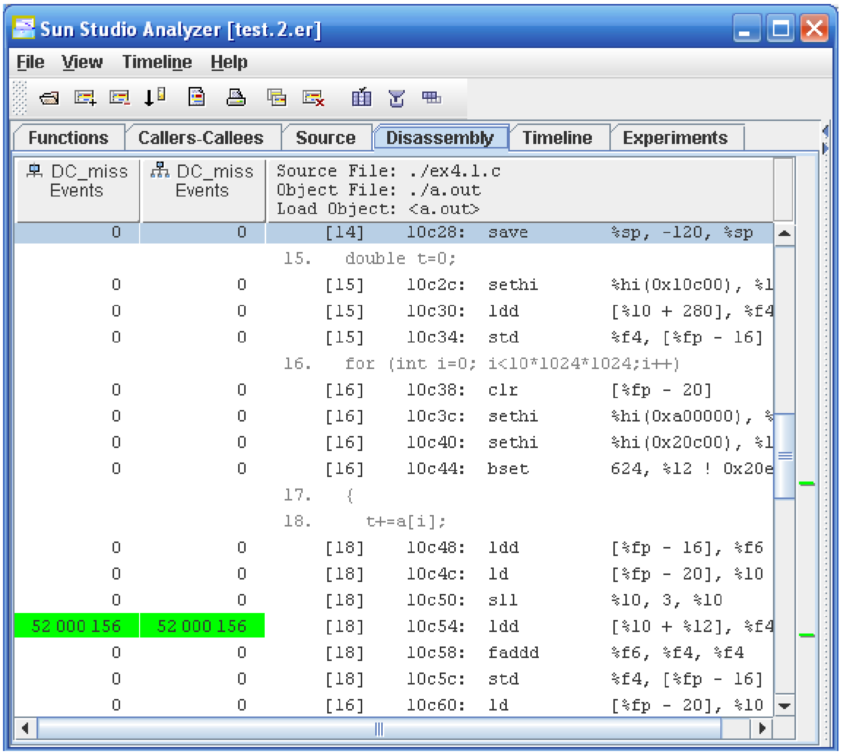


```
 User     User     Source File: ./ex4.1.c
 CPU      CPU      Object File: ./a.out
 (sec.)   (sec.)   Load Object: <a.out>
                   13. double sum()
 0.       0.       14. {
                       <Function: sum>
 0.       0.       15.   double t=0;
 1.411    1.411    16.   for (int i=0; i<10*1024*1024;i++)
                   17.   {
 3.893    3.893    18.     t+=a[i];
                   19.   }
 0.       0.       20.   return t;
 0.       0.       21. }
                   22.
                   23. void main()
 0.       0.       24. {
                       <Function: main>
 0.       0.       25.   for (int i=0; i<10; i++)
                   26.   {
 0.       5.704    27.     clear();
 0.       5.304    28.     sum();
                   29.   }
 0.       0.       30. }
```

FIGURE 9-4 Profile at Source-Code Level

The profile can also be viewed at the disassembly level, as shown in FIGURE 9-5. To determine where the time is being spent, Performance Analyzer inspects the address of the next instruction to be executed. The frequency with which it does this can be changed by the user, but the default is to do this inspection 100 times a second. Every time this happens, the tool records an additional quantum of time for that particular location, so a PC that is observed 100 times at the default frequency of 100 samples per second will end up with 1 second of time being attributed to it.

For each instruction, the compiler records the line number and source file for the source code that caused the instruction to be generated. The source line number is shown in square parentheses in FIGURE 9-5 next to each disassembled instruction. To determine how much time should be attributed to a particular line of source code, the tools sum the time attributed to all the disassembly instructions that were generated from that source line.


Sun Studio Analyzer [test.2.er]
File View Timeline Help
Functions Callers-Callees Source Disassembly Timeline Experiments

```
DC_miss     DC_miss     Source File: ./ex4.1.c
Events      Events      Object File: ./a.out
                        Load Object: <a.out>
         0           0       [14]    10c28:  save        %sp, -120, %sp
                          15.    double t=0;
         0           0       [15]    10c2c:  sethi       %hi(0x10c00), %l
         0           0       [15]    10c30:  ldd         [%l0 + 280], %f4
         0           0       [15]    10c34:  std         %f4, [%fp - 16]
                          16.    for (int i=0; i<10*1024*1024;i++)
         0           0       [16]    10c38:  clr         [%fp - 20]
         0           0       [16]    10c3c:  sethi       %hi(0xa00000), %
         0           0       [16]    10c40:  sethi       %hi(0x20c00), %l
         0           0       [16]    10c44:  bset        624, %l2 ! 0x20e
                          17.    {
                          18.      t+=a[i];
         0           0       [18]    10c48:  ldd         [%fp - 16], %f6
         0           0       [18]    10c4c:  ld          [%fp - 20], %l0
         0           0       [18]    10c50:  sll         %l0, 3, %l0
52 000 156  52 000 156       [18]    10c54:  ldd         [%l0 + %l2], %f4
         0           0       [18]    10c58:  faddd       %f6, %f4, %f4
         0           0       [18]    10c5c:  std         %f4, [%fp - 16]
         0           0       [16]    10c60:  ld          [%fp - 20], %l0
```

FIGURE 9-5 Profile at the Disassembly Level

UltraSPARC T2 has exact traps for hardware counter overflow, which is the mechanism that identifies where in the code the events counted by the hardware counters occur. An exact trap means that the tools can identify the exact instruction that caused the event. In FIGURE 9-5, the data cache misses are identified as being due to the load at 0x10c54. Unsurprisingly, this load corresponds to the load of the value of `a[i]` in the source code.

9.2.2 Gathering Instruction Counts With BIT

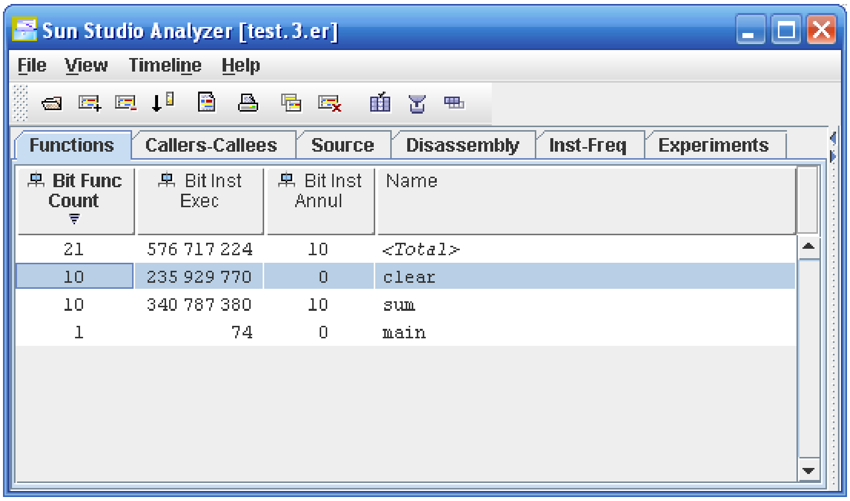
The BIT tool (`http://cooltools.sunsource.net/bit`) enables the user to gather data about the number of times an instruction is executed, the number of times a routine is called, the probability of a branch being taken, and other useful metrics.

Once BIT is installed, it can either be invoked directly or, more conveniently, through `collect -c on`. For BIT to work, the application must be compiled with optimization and compiled and linked with the compiler flag `-xbinopt=prepare` for the compiler to record the necessary annotations. CODE EXAMPLE 9-7 shows an example of compiling and gathering the execution counts for the example program.

CODE EXAMPLE 9-7 Gathering Instruction Count Data With BIT

```
% cc -O -xbinopt=prepare -g ex.9.4.c
% collect -c on a.out
Creating experiment database test.3.er ...
% analyzer test.3.er
```

When invoked by `collect`, BIT outputs the results as an Analyzer experiment, as shown in FIGURE 9-6.

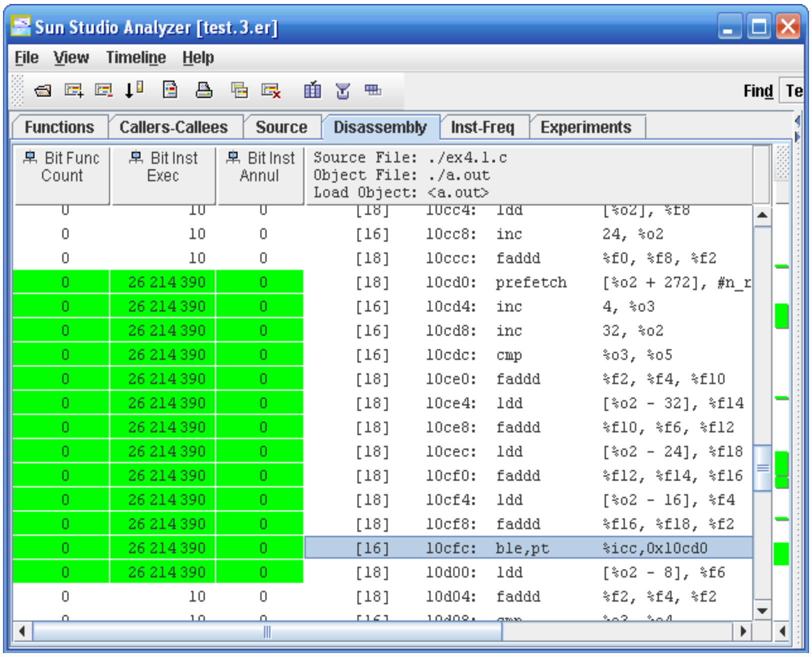


Bit Func Count	Bit Inst Exec	Bit Inst Annul	Name
21	576 717 224	10	<Total>
10	235 929 770	0	clear
10	340 787 380	10	sum
1	74	0	main

FIGURE 9-6 Analyzer Shown Count Data From BIT

The first of the three columns of data shows the number of times that each function was entered: the routine `main` was entered once, the routines `sum` and `clear` were both entered ten times. The second column shows the total number of instructions that were executed over the run of the code, in this case, nearly 600 million dynamic instructions. The final column of data shows the annulled instructions. Annulled instructions are instructions that have been placed in the delay slot of the branch. They are executed if the branch is taken, but if the branch is not taken, the instructions are annulled.

The instruction count data is also available at the disassembly level, as shown in FIGURE 9-7. You can see that the loop at addresses 0x10cd0 to 0c10d00 is entered 10 times and has an average trip count of about two and a half million iterations every time it is entered. Since the code was compiled with optimization, the compiler has unrolled the loop four times, as can be seen from the four `faddd` instructions in the loop.

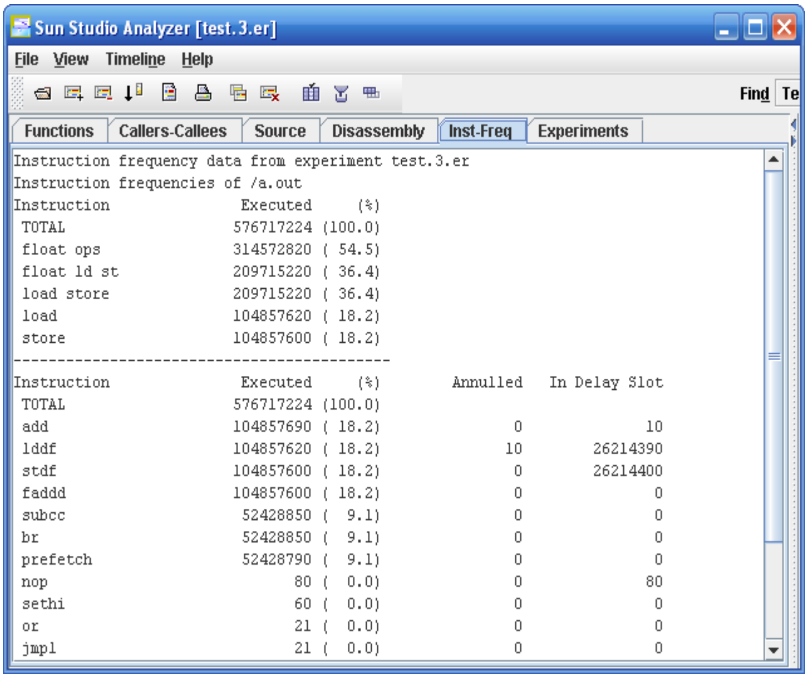


```
Source File: ./ex4.1.c
Object File: ./a.out
Load Object: <a.out>
```

Bit Func Count	Bit Inst Exec	Bit Inst Annul				
0	10	0	[18]	10cc4:	ldd	[%o2], %f8
0	10	0	[16]	10cc8:	inc	24, %o2
0	10	0	[18]	10ccc:	faddd	%f0, %f8, %f2
0	26 214 390	0	[18]	10cd0:	prefetch	[%o2 + 272], #n_r
0	26 214 390	0	[16]	10cd4:	inc	4, %o3
0	26 214 390	0	[16]	10cd8:	inc	32, %o2
0	26 214 390	0	[16]	10cdc:	cmp	%o3, %o5
0	26 214 390	0	[18]	10ce0:	faddd	%f2, %f4, %f10
0	26 214 390	0	[18]	10ce4:	ldd	[%o2 - 32], %f14
0	26 214 390	0	[18]	10ce8:	faddd	%f10, %f6, %f12
0	26 214 390	0	[18]	10cec:	ldd	[%o2 - 24], %f18
0	26 214 390	0	[18]	10cf0:	faddd	%f12, %f14, %f16
0	26 214 390	0	[18]	10cf4:	ldd	[%o2 - 16], %f4
0	26 214 390	0	[18]	10cf8:	faddd	%f16, %f18, %f2
0	26 214 390	0	[16]	10cfc:	ble,pt	%icc,0x10cd0
0	26 214 390	0	[18]	10d00:	ldd	[%o2 - 8], %f6
0	10	0	[18]	10d04:	faddd	%f2, %f4, %f2
0	10	0	[16]	10d08:	cmp	%o3, %o4

FIGURE 9-7 Instruction Count Data at the Disassembly Level

Since the tool has data for the execution frequency for each PC, it is also able to calculate the execution frequency of each different type of assembly language instruction. This data is shown on a separate tab in the Analyzer GUI, as shown in FIGURE 9-8. The summary data shows that about a third of the total instruction count is loads and stores, and these are all loads and stores of floating-point data. Over half of the total instruction count is concerned with manipulating floating-point data.


Sun Studio Analyzer [test.3.er]
File View Timeline Help
Find Te
Functions Callers-Callees Source Disassembly Inst-Freq Experiments

```
Instruction frequency data from experiment test.3.er
Instruction frequencies of /a.out
Instruction                  Executed     (%)
 TOTAL                      576717224 (100.0)
 float ops                  314572820 ( 54.5)
 float ld st                209715220 ( 36.4)
 load store                 209715220 ( 36.4)
 load                       104857620 ( 18.2)
 store                      104857600 ( 18.2)
-------------------------------------------
Instruction                  Executed     (%)        Annulled   In Delay Slot
 TOTAL                      576717224 (100.0)
 add                        104857690 ( 18.2)               0              10
 lddf                       104857620 ( 18.2)              10        26214390
 stdf                       104857600 ( 18.2)               0        26214400
 faddd                      104857600 ( 18.2)               0               0
 subcc                       52428850 (  9.1)               0               0
 br                          52428850 (  9.1)               0               0
 prefetch                    52428790 (  9.1)               0               0
 nop                               80 (  0.0)               0              80
 sethi                             60 (  0.0)               0               0
 or                                21 (  0.0)               0               0
 jmpl                              21 (  0.0)               0               0
```

FIGURE 9-8 Instruction Type Frequency Information From BIT

BIT can also output reports showing the probability and frequency that branches are taken and the execution counts for basic blocks. To get this data, invoke BIT at the command line, as shown in CODE EXAMPLE 9-8.

CODE EXAMPLE 9-8 Invoking BIT at the Command-Line to Generate Basic Block Execution Counts

```
$ cc -O -xbinopt=prepare -g ex.9.4.c
$ bit instrument a.out
$ a.out.instr
$ bit analyze -a bbc a.out
Basic block counts of /a.out
=================
              Count              PC   #Instrs  Function name
                 10         0x10c00        11  clear
                 10         0x10c2c         1
           26214400         0x10c30         9
                 10         0x10c54         3
                  0         0x10c60         1
                  0         0x10c64         5
                 10         0x10c78         2
                 10         0x10c88        11  sum
...
$ bit analyze -a branch a.out
Branch taken/not taken report of /a.out
============================
  PC  Dir %Taken  %Not  Compiler       Trip    Taken     Not  Instr.
                 Taken Prediction     Count            Taken
                       Correct?
10c24 F     0.0%  100.0%   Y             10        0      10  br,pn
10c4c B   100.0%    0.0%   Y       26214400 26214390      10  br,pt
10c58 F   100.0%    0.0%   N             10       10       0  br,pn
...
```

As can be seen in CODE EXAMPLE 9-8, BIT is first invoked to generate an instrumented version of the application. BIT uses the annotations recorded under `-xbinopt=prepare` to disassemble the application and generate a new version of the binary containing the instrumented code. This instrumented application is the one that must be run. At the end of the run of the instrumented binary, a data file is created containing the results of the instrumented run. BIT is then invoked to analyze this data file.

BIT can instrument only the parts of the code that were compiled with `-xbinopt=prepare`. If an application has a mix of files with and without this option, then the data reported by BIT is gathered only from the instrumented parts. If the instrumented code calls into uninstrumented libraries, then those libraries are also excluded from the analysis.

BIT can also generate coverage reports showing the code that has been executed. It can also generate a report called an uncoverage report. The idea of uncoverage is to examine the call stack of the routines as well as to determine whether they are executed or not. The tool can then identify particular routines, which if executed, also cause other unexecuted routines to be used. With this analysis, it is possible to rapidly develop tests that will hit particular parts of the code, knowing that those tests will inevitably execute other uncovered routines. The coverage report can be generated by BIT, as shown in CODE EXAMPLE 9-9. An analyzer experiment, which contains both the coverage and uncoverage reports, is also generated.

CODE EXAMPLE 9-9 Generating a Coverage Report From BIT

```
$ bit coverage a.out
Creating experiment database test.5.er ...
BIT Code Coverage
Total Functions: 3
Covered Functions: 3
Function Coverage: 100.0%
Total Basic Blocks: 17
Covered Basic Blocks: 15
Basic Block Coverage: 88.2%
Total Basic Block Executions: 52,428,902
Average Executions per Basic Block: 3,084,053.06
Total Instructions: 88
Covered Instructions: 81
Instruction Coverage: 92.0%
Total Instruction Executions: 576,717,224
Average Executions per Instruction: 6,553,604.82
```

9.2.3 Evaluating Training Data Quality

When an application is compiled with profile feedback, the compiler gathers information about branch probabilities and execution frequencies from the training data used. It is, therefore, important that this training data be representative of the workload that will usually be run with the production binary. The BIT tool can generate reports on the execution frequencies of the basic blocks of code and also on the probabilities of each branch instruction being taken or untaken.

Reports from the training and actual workloads can be compared to see whether individual branch instructions are taken with the same frequency and whether the same basic blocks are important. Gove and Spracklen (2006, 2007)[1] used this analysis on the SPEC CPU2006 benchmarks. One of the important observations from that paper is that most of the training workloads were already representative of the workloads used in the runs of the benchmarks. The weakest agreement was seen when the training workload was of a different problem type than the workload used in the benchmark run.

[1] ACM SIGARCH Computer Architecture News, Vol. 35, No. 1 - March 2007

http://www.spec.org/cpu2006/publications/SIGARCH-2007-03/10_cpu2006_training.pdf

http://www.spec.org/workshops/2006/papers/10_Darryl_Gove.pdf

The methodology is described in detail at `http://developers.sun.com/solaris/articles/coverage.html`. An overview is as follows. The binary is compiled with the flag `-xbinopt=prepare`. The compiled binary is instrumented with BIT, and then reports for both the training workload and an actual workload are generated. A simple example program that demonstrates this is shown in CODE EXAMPLE 9-10.

CODE EXAMPLE 9-10 Example Program for Demonstrating Training Workload Quality

```
#include <stdio.h>

void main(int argc, char** argv)
{
  for (int i=1; i<argc; i++)
  {
    printf("%s\n",argv[i]);
  }
}
```

The steps necessary to generate the report showing the branch probabilities are shown in CODE EXAMPLE 9-11, together with a sample of the output from the report. The report indicates the number of times each branch was taken and untaken, as well as whether the compiler set the prediction bit on the branch instruction correctly.

CODE EXAMPLE 9-11 Branch Probabilities

```
$ cc -O -xbinopt=prepare -o profile ex.9.10.c
$ bit instrument profile
$ profile.instr 1
$ bit analyze -a branch profile
Branch taken/not taken report of profile
========================================
 PC   Dir %Taken  %Not  Compiler  Trip  Taken Not    Instruction
                 Taken Prediction Count       Taken
                         Correct?
10bec F    0.0% 100.0%     Y        1     0    1   br,pn@(le),%icc
10c1c B    0.0% 100.0%     N        1     0    1  br,a,pt@(le),%icc
```

The steps necessary to generate the basic block execution frequency report are shown in CODE EXAMPLE 9-12, together with a sample of the output from that report. The report includes counts of the number of times each basic block was executed.

CODE EXAMPLE 9-12 Basic Block Counts

```
$ bit analyze -a bbc profile
Basic block counts of profile
=============================
               Count              PC    #Instrs  Function name
                   1         0x10be4          4  main
                   1         0x10bf4          5
                   1         0x10c08          7
                   1         0x10c24          2
```

Next, the test code is executed with multiple command-line parameters. The branch and block reports for this are shown in CODE EXAMPLE 9-13.

CODE EXAMPLE 9-13 Branch and Block Data for Actual Workload

```
$ bit instrument profile
$ profile.instr 1 2 3 4
1
2
3
4
$ bit analyze -a branch profile
Branch taken/not taken report of profile
========================================
 PC   Dir %Taken   %Not Compiler   Trip  Taken Not   Instruction
                  Taken Prediction Count      Taken
                          Correct?
10bec F   0.0% 100.0%     Y       1     0    1  br,pn@(le),%icc
10c1c B  75.0%  25.0%     Y       4     3    1   br,a,pt@(le),%icc
$ bit analyze -a bbc profile
Basic block counts of profile
=============================
               Count              PC    #Instrs  Function name
                   1         0x10be4          4  main
                   1         0x10bf4          5
                   4         0x10c08          7
                   1         0x10c24          2
```

The branch probabilities data can be used to examine to determine whether there is a good fit between the training and real workloads. Each branch can be classified as usually taken or usually untaken. If the branch has the same prediction for both the training and reference workloads, then the branch is predicted correctly. If the branch has different behavior for the training and real workloads, then the branch is predicted incorrectly.

For a single metric that describes the quality of the training workload, the dynamic execution count of the correctly predicted branches during the real workload is divided by the dynamic execution count for all branches during the real workload. This results in a value between 0 and 1. A value near 1 means that all the frequently executed branches were correctly predicted; a value near 0 means that none of the frequently executed branches were correctly predicted.

For the example, there are two branches. The first branch is predicted not taken by the training workload and is not taken by the reference workload; it is executed once by the reference workload. The second branch is also predicted not taken by the training workload but is taken by the reference workload. The second branch is encountered four times by the reference workload. So the calculation is one correctly predicted branch executed once by the reference workload and four incorrectly predicted branches executed by the reference workload, so a branch prediction quality of $(1 \times 1 + 0x4) \div 5 = 0.2$. So the training workload is a poor match for the branch behavior of the reference workload.

The calculation for the agreement between the dynamic execution frequency of the basic blocks is slightly different. The probability of a branch being taken has a range between 0 and 1, but there is no equivalent upper limit on the execution frequency for a basic block. However, one metric that can be clearly defined for basic blocks is whether the training workload covered all the basic blocks that are important for the real workload; after all, if the workload does not execute the block, then the compiler can obtain no information about the block's runtime behavior. The block is considered covered so long as the training workload executed the block at least once. If a block was never executed by the training workload, then the block is considered to be uncovered.

A value between 0 and 1 can be calculated by summing the dynamic execution count of those basic blocks in the real workload that were covered by the training workload, and dividing this by the dynamic execution count of all the basic blocks in the real workload. A value near 1 means that all the frequently executed basic blocks in the real workload were also executed at least once by the training workload. A value near 0 means that none of the basic blocks that were frequently executed in the real workload were covered by the training workload. In the example, the training workload executes all four basic blocks in the reference workload, so gets a basic block coverage quality of $(1 + 1 + 4 + 1) \div 7 = 1.0$. The training workload is a good coverage match for the reference workload.

It is worth observing that although these metrics are a proxy for the quality of the training workload, it is also possible to plot the agreement between the training and real workloads on a graph. Examples of graphs together with scripts that process the results from BIT are available in the article referred to at
`http://developers.sun.com/solaris/articles/coverage.html`.

9.2.4 Profiling With SPOT

SPOT (`http://cooltools.sunsource.net/spot/`) is a tool that makes the process of locating and identifying performance issues as easy as possible. The tool runs an application under multiple probes and generates an HTML report showing the results of these probes plus the profile of the application. The probes that are available depend on the features of the hardware and operating system, so not all probes will be available on all machines. Below are descriptions of the probes that SPOT uses:

- System information is recorded, making it easy to track the configuration of the system on which the application was run.
- Compiler flags are extracted from the application if this information is recorded. It also examines the libraries that are linked into the application.
- Information about the performance counter events encountered by the application is gathered by `ripc` and used to estimate the amount of time lost to various processor stall states. This information may be used by SPOT to profile the application under those hardware performance counters that contribute the most to the total stall time.
- Profile information that shows where the time is being spent in the application is gathered by Performance Analyzer.
- Instruction count data is gathered by BIT if the application has been compiled with appropriate flags.
- System-wide bandwidth utilization data and system-wide trap data are gathered over the run of the application. Collecting this information requires superuser privileges.

The report generated by SPOT has several advantages over running the various tools stand-alone. The report saves a significant amount of context information, so you can go back to a report and find out the compiler flags used, the name of the system, as well as the source and disassembly of the time-consuming parts of the application. The other significant advantage of using SPOT is that you can view HTML reports remotely using a browser. Moreover, it is trivial to share a URL indicating data of interest or a hot-point in the source code with remote colleagues, and you can expect them to see exactly the data you are looking at.

An application can be run under SPOT in the same way that the application is run under `collect`, as shown in CODE EXAMPLE 9-14. Alternatively, SPOT can attach to a running process with `spot -P` *pid*, although this approach is not recommended for production systems as it potentially involves stopping the process multiple times. SPOT defaults to gathering a short report on the performance of an application; for situations in which there is more time, it can gather a longer report under the `-X` command-line option, as shown in CODE EXAMPLE 9-14.

CODE EXAMPLE 9-14 Gathering Extended Information for `a.out` With SPOT

```
$ spot -X a.out
```

The profile information SPOT gathers will already be familiar from the previous results in this chapter. One part of the report that might be less familiar is the hardware counter data reported by `ripc`. Part of this data for a short running application is shown in CODE EXAMPLE 9-15.

CODE EXAMPLE 9-15 `ripc` Output as Displayed by SPOT

```
========================================================
Analysis Of Application Stall Behavior
========================================================
  Stall                     Ticks         Sec       %
========================================================
ITLB-miss                   7,426        0.000    0.0%
DTLB-miss                 412,696        0.000    0.6%
Instr. Issue            3,894,132        0.003    5.3%
D-Cache                25,512,190        0.021   34.9%
L2-DC-miss                642,128        0.001    0.9%
L2-IC-miss                108,004        0.000    0.1%
StoreQ                    421,840        0.000    0.6%
--------------------------------------------------------
Total Stalltime        30,998,416        0.026   42.3%
--------------------------------------------------------
Total CPU Time                          0 Sec
Total Elapsed Time                      0 Sec
Total Cycle Count              73,200,008
Total Instr. Count             12,301,788
FP Instructions                  3,964   0.0% of Total Instr.
MIPS                              201.669
--------------------------------------------------------
Unfinished FPops                        0
```

The section at the start of the `ripc` output estimates the number of cycles lost to each type of processor event. For this code, about a third of the runtime is lost to data cache miss events. About 40% of the total runtime is spent in stall cycles. `ripc` also tracks total times, instruction counts, and the floating-point operations. It also tracks the number of system-wide floating-point operation traps that occurred while the application was running. These traps are caused when the processor needs a floating-point operation to complete in software. The tool also tracks memory size and read bandwidth utilization.

9.2.5 Debugging With dbx

The Sun Studio debugger, called dbx, in common with most debuggers supports the examination of core files as well as the running of applications. The use of dbx to examine a core file is shown in CODE EXAMPLE 9-16. The core file usually records the name of the executable that generated it, so dbx can be started with a minus sign as the name of the executable. If this is not the case, then the executable can be specified on the command line.

CODE EXAMPLE 9-16 Using dbx to Examine a Core File

```
$ dbx application core
$ dbx - core
```

Consider the code in CODE EXAMPLE 9-17. This code has the error of memory being used without having been allocated.

CODE EXAMPLE 9-17 Program With Memory Access Error

```
#include <stdlib.h>

void f(double *a, int b)
{
  a[b]=0;
}

void main()
{
  double *a;
  f(a,1);
  f(a,1000);
  f(a,2);
}
```

The compiler flag -g causes the compiler to generate debug information. In Sun Studio 12, and subsequent versions, this debug information is recorded in the executable. Prior versions of Sun Studio recorded this information in the object files but not the binary. The program in CODE EXAMPLE 9-17 generates a core file when run, and dbx can be used to examine what happened, as shown in CODE EXAMPLE 9-18. The application has been compiled without optimization in order to obtain the most debug information. dbx correctly identifies the application from the core file and reports the error that caused the core file to be generated.

CODE EXAMPLE 9-18 Using dbx on a Core File

```
% cc -g ex.9.17.c
% a.out
Segmentation Fault (core dumped)
% dbx - core
Corefile specified executable: "/a.out"
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
program terminated by signal SEGV (access to address exceeded
protections)
Current function is f
    5     a[b]=0;
(dbx) where
=>[1] f(a = 0x1070c, b = 1), line 5 in "ex.9.17.c"
  [2] main(), line 11 in "ex.9.17.c"
(dbx) print b
b = 1
(dbx) print a
a = 0x1070c
```

The initial display shows the source line at which the error occurred. The command `where` shows the call stack at the point of error, reporting that the routine `f` was called by the routine `main`. It also shows the parameters passed into the routine `f`. You can also print the values of the parameters, using the `print` statement.

If the code had been compiled with optimization, the debugger would have been able to present less information, and it may be necessary to look at the disassembly code to identify what happened. The process can be slightly more tricky, as shown in CODE EXAMPLE 9-19.

CODE EXAMPLE 9-19 Using dbx to Explore a Core File From Optimized Code

```
% cc -g -O ex.9.17.c
% a.out
Segmentation Fault (core dumped)
% dbx - core
Corefile specified executable: "/a.out"
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
program terminated by signal SEGV (no mapping at the fault address)
Current function is f (optimized)
    4   {
(dbx) where
=>[1] f(a = ?, b = ?) (optimized), at 0x10ba8 (line ~4) in "ex.9.17.c"
  [2] main() (optimized), at 0x10bb4 (line ~11) in "ex.9.17.c"
(dbx) dis f
0x00010ba0: f       :   sll      %o1, 3, %o5
0x00010ba4: f+0x0004:   retl
0x00010ba8: f+0x0008:   clrx     [%o0 + %o5]
...
(dbx) regs
current frame:  [1]
g0-g1    0x00000000 0x00000000 0x00000000 0x0000f000
...
o0-o1    0x00000000 0x0001070c 0x00000000 0x00000001
o2-o3    0x00000000 0x00014bd1 0x00000000 0x00014bd1
o4-o5    0x00000000 0x00018331 0x00000000 0x00000008
...
pc       0x00000000 0x00010ba8:f+0x8     clrx     [%o0 + %o5]
npc      0x00000000 0x00010bbc:main+0x10 ld       [%sp + 92], %o0
```

In this case, the compiler is unable to identify the exact line of source code that causes the problem. It does identify the routine correctly. Looking at the disassembly of the routine with the `dis` command shows that the problem instruction is the `clr` (clear) instruction at 0x10ba8. Since the routine is short, it is trivial to see that `%o0` contains the value of `a` that was passed into the routine and that `%o5` contains the index into that array. The `regs` command prints the current values of the registers and shows that `%o1` contains the value 1, `%o5` contains 8 (which is 1 multiplied by the size of the array element, each double taking 8 bytes), and `%o0` contains 0x1070c—coincidentally the same value as in the unoptimized situation.

It is also possible to run a program under `dbx`. In that case, `dbx` must be passed the name of the executable as a command-line parameter, as shown for the unoptimized code in CODE EXAMPLE 9-20. If the application requires any command-line parameters, these can be set with the `runargs` command to `dbx`.

CODE EXAMPLE 9-20 Running a Problem Application Under `dbx`

```
$ dbx a.out
Reading a.out
Reading ld.so.1
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
(dbx) run
Running: a.out
(process id 28372)
signal SEGV (access to address exceeded protections)
in f at line 5 in file "ex.9.17.c"
    5     a[b]=0;
```

9.2.6 Using Discover to Locate Memory Access Errors

Memory access errors are bugs caused when memory is accessed without being allocated, after it has been freed, before it has been initialized, etc. Bugs of this kind can be hard to locate because the results of the error often manifest themselves an arbitrarily long time after execution of the code containing the problem. The tool Discover (`http://cooltools.sunsource.net/discover/`) is designed to detect these kinds of problems. For Discover to be used, the application must be built with optimization and the compiler option `-xbinopt=prepare`. Then the `discover` tool can make an instrumented version of the application, which is then run. CODE EXAMPLE 9-21 shows an example run of the tool.

Discover can also generate the results as an HTML report. You enable this by setting the environment variable `DISCOVER_HTML` before running the instrumented binary.

CODE EXAMPLE 9-21 Using Discover to Locate Memory Access Errors

```
% cc -g -O -xbinopt=prepare ex.9.17.c
% /opt/SUNWspro/extra/bin/discover a.out
% a.out
ERROR (UMR): accessing uninitialized data from address
0xffbffa74
(4 bytes) at:
        main() + 0x4c [/a.out:0x30088]
          <ex.9.17.c:11>:
                 8:    void main()
                 9:    {
                10:      double *a;
                11:=>    f(a,1);
                12:      f(a,1000);
                13:      f(a,2);
                14:    }
        _start() + 0x108 [/a.out:0x107a8]
Segmentation Fault (core dumped)
```

9.3 Throughput Computing

The OpenSPARC T1 and OpenSPARC T2 processors have multiple threads sharing a core, and as such are designed for workloads that require high throughput rather than short response time. The common way of illustrating why multiple threads can share a single core shows the gaps that appear in the instruction stream from events like cache misses and demonstrates that these gaps could be used by other threads to make progress. This can be expressed as *all threads will have stall cycles, and these stall cycles are sufficient to enable multiple threads to interleave*. These subsections look at how to develop applications that use multiple threads:

- *Measuring Processor Utilization* on page 153
- *Using Performance Counters to Estimate Instruction and Stall Budget Use* on page 156
- *Collecting Instruction Count Data* on page 159
- *Strategies for Parallelization* on page 159
- *Parallelizing Applications With POSIX Threads* on page 160
- *Parallelizing Applications With OpenMP* on page 162
- *Using Autoparallelization to Produce Parallel Applications* on page 165
- *Detecting Data Races With the Thread Analyzer* on page 165
- *Avoiding Data Races* on page 169
- *Considering Microparallelization* on page 174
- *Programming for Throughput* on page 177

9.3.1 Measuring Processor Utilization

With each core running multiple threads, the maximum throughput that can be achieved is one instruction per core for OpenSPARC T1 and two instructions per core for OpenSPARC T2. In this situation, each core would be achieving the maximum number of instructions that it can achieve. Obviously, not all workloads will manage this rate of instruction issue.

The instruction issue rate is not visible through the normal Solaris tools such as `prstat` (which shows how much time the operating system has scheduled a thread onto a virtual processor). The rate of instruction issue can be measured for an individual process, using `cputrack` to read the performance counters, or it can be measured for the entire system with `cpustat`. A convenient way of viewing the system-wide issue rate is with `corestat` (`http://cooltools.sunsource.net/corestat/`), which reports the utilization of the integer and floating-point pipes of the processor cores.

`corestat` works by reporting the number of issue instructions out of the total maximum possible number of instructions that could be issued. This produces a value showing how much of the "instruction budget" for a processor core is actually used. Some sample output showing an idle UltraSPARC T2-based system is shown in CODE EXAMPLE 9-22. Only pipeline zero of core 2 shows any significant activity.

CODE EXAMPLE 9-22 UltraSPARC T2 Integer Pipe Utilization From `corestat`

```
  Core Utilization for Integer pipeline
 Core,Int-pipe      %Usr     %Sys    %Usr+Sys
 -------------      -----    -----   --------
     0,0            0.00     0.19      0.20
     0,1            0.00     0.01      0.01
     1,0            0.00     0.03      0.03
     1,1            0.00     0.01      0.01
     2,0            1.15     0.02      1.16
     2,1            0.00     0.01      0.01
     3,0            0.02     0.02      0.04
     3,1            0.00     0.01      0.01
     4,0            0.00     0.02      0.03
     4,1            0.00     0.01      0.01
     5,0            0.02     0.01      0.03
     5,1            0.00     0.01      0.01
     6,0            0.05     0.03      0.08
     6,1            0.00     0.01      0.01
     7,0            0.00     0.03      0.03
     7,1            0.00     0.01      0.01
 -------------      -----    -----   ------
     Avg            0.08     0.03      0.10
```

corestat is essentially a convenient wrapper for cpustat, which tracks processor events system-wide. Various hardware events can be counted, and it is useful to discuss the information that the events can provide. CODE EXAMPLE 9-23 shows a multithreaded program that can be used to examine instruction issue rate.

CODE EXAMPLE 9-23 Example Multithreaded Code

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
  int* array=calloc(500*1000*1000,sizeof(int));
  int total=0;
  for (int j=0; j<20; j++)
  {
    #pragma omp parallel for default(__auto)
    for (int i=0; i<500*1000*1000; i++)
    {
      if (array[i]==1) {total++;}
    }
  }
  printf ("Total = %i\n",total);
}
```

The code uses OpenMP to parallelize a simple loop; OpenMP is introduced in *Parallelizing Applications With OpenMP* on page 162. The environment variable OMP_NUM_THREADS sets the number of threads that the application will use in the parallel region. CODE EXAMPLE 9-24 shows compiling and running this program on an UltraSPARC T1-based system. The compiler flag -xopenmp causes the compiler to recognize the OpenMP directive in the code. The flags -xloopinfo and -xvpara cause the compiler to report more information about the parallelization of the code.

CODE EXAMPLE 9-24 Compiling and Running an OpenMP Parallelized Code

```
$ cc -g -O -xopenmp -xvpara -xloopinfo ex.9.23.c
"ex.9.23.c", line 8: not parallelized, loop contains pragma loop
"ex.9.23.c", line 11: PARALLELIZED, user pragma used
$ setenv OMP_NUM_THREADS 32
$ timex a.out
Total = 0

real       15.97
user     5:28.28
sys         3.15
```

While the program is running, cpustat can be used to track the number of instructions issued by each thread, as shown in CODE EXAMPLE 9-25. The command requests that the instruction count be captured every second for ten seconds. The columns show the time at which the sample was captured, the virtual CPU, the type of line of text being reported (in this case, "tick"), and the count from the hardware performance counter.

CODE EXAMPLE 9-25 Using `cpustat` to Capture Instruction Issue Rates

```
$ cpustat -c Instr_cnt,sys 1 10
...
  5.010   8  tick 186368297
  5.010   9  tick 187145974
  5.010  16  tick 185612254
  5.010  26  tick 186933907
  5.010   2  tick 186923129
  5.010  25  tick 186640326
  5.010  18  tick 186967737
  5.010  10  tick 186894549
  5.010   1  tick 186611175
  5.010   3  tick 187320071
  5.010  17  tick 186536875
  5.010  27  tick 187316280
  5.010  19  tick 187204290
  5.010  24  tick 188030654
  5.010   4  tick 186343850
  5.010  11  tick 187312586
  5.010  12  tick 188903145
  5.010   5  tick 190709642
  5.010  13  tick 186474558
  5.010  28  tick 188546808
  5.010  21  tick 187561252
  5.010  20  tick 187388082
  5.010  29  tick 185464662
  5.010   6  tick 189905445
  5.010  14  tick 187142919
  5.010  22  tick 186514943
  5.010   7  tick 186904827
  5.010  15  tick 177650912
  5.010   0  tick 189722028
  5.010  23  tick 186926542
  5.020  31  tick 187765561
  5.020  30  tick 187851684
...
```

The output shows that each thread executed about 188M instructions every second. The UltraSPARC T1-based system that generated this data has four threads on each core. Taking virtual CPU IDs 0, 1, 2, and 3, which all reside on the first core, the total number of instructions issued by that core is about 750M; for a 1.2 GHz processor this is 750 ÷ 1200 = 65% utilization.

For both the UltraSPARC T1 and UltraSPARC T2 processors, in ideal circumstances one of four threads gets to issue an instruction at each cycle. The other three threads either are waiting to issue an instruction or are stalled on some processor event. Under an even distribution of opportunities to issue an instruction, each thread will have three cycles during which it cannot issue an instruction for every cycle at which it can issue an instruction. During those three cycles it does not matter whether the thread is stalled on an event or ready to issue, since other threads will use the instruction issue opportunity before the current thread gets another opportunity to issue an instruction. Consequently, each thread has what might be called a "*stall budget*" that is three times the "*instruction budget*." Until the number of cycles that the thread is stalled exceeds this budget, the thread will continue issuing instructions at its peak rate.

9.3.2 Using Performance Counters to Estimate Instruction and Stall Budget Use

The hardware performance counters on UltraSPARC T1 and UltraSPARC T2 can be used to capture some information about what stall events a thread has encountered. The interesting performance counters for the two processors are listed in TABLE 9-3.

TABLE 9-3 UltraSPARC T1 and UltraSPARC T2 Performance Counters

Event	UltraSPARC T1 Performance Counter	UltraSPARC T2 Performance Counter
Cycles store buffer full	SB_full	Not available
Floating-point instructions issued	FP_instr_cnt	Instr_FGU_ arithmetic
Instruction cache misses	IC_miss	IC_miss
Data cache misses	DC_miss	DC_miss
Instruction TLB misses	ITLB_miss	ITLB_miss
Data TLB misses	DTLB_miss	DTLB_miss

TABLE 9-3 UltraSPARC T1 and UltraSPARC T2 Performance Counters

Event	UltraSPARC T1 Performance Counter	UltraSPARC T2 Performance Counter
Instruction cache misses that also miss L2 cache	L2_imiss	L2_imiss
Data cache load misses that also miss the L2 cache	L2_dmiss_ld	L2_dmiss_ld
Instructions issued	Instr_cnt	Instr_cnt

With the exception of the store buffer full counter on UltraSPARC T1, the counters collect data on the number of events, not the number of cycles that an event consumed. For example, a data cache miss that is satisfied by data from the second-level cache will take about 18 cycles to complete. The way to estimate the number of cycles lost by the condition is to multiply the number of events by the cost of that type of event.

It is relatively easy to use `cpustat` or `cputrack` to capture the frequency of the various events during the run of an application. These can then be combined to produce an estimate of the total number of cycles spent in stall conditions. The data is summarized for the example code, run on a 1.2 GHz UltraSPARC T1-based system, in TABLE 9-4. It is important to appreciate that the estimates for the costs of the various events are just estimates, not exact measurements. However, this does not undermine the exercise of estimating the costs for the various events—order-of-magnitude estimates are as useful in this context as exact values.

TABLE 9-4 Total Cycles Spent in Stall Events

Event	UltraSPARC T1 Performance Counter	Est. Cost in Cycles Per Event	Events	Est. Cost in Sec
Cycles store buffer full	SB_full	1	185,264	0
Floating-point instructions issued	FP_instr_cnt	30	0	0
Instruction cache misses	IC_miss	20	1,683,889	0
Data cache misses	DC_miss	20	5,059,778,329	84
Instruction TLB misses	ITLB_miss	100	217	0
Data TLB misses	DTLB_miss	100	686,278	0

TABLE 9-4 Total Cycles Spent in Stall Events

Event	UltraSPARC T1 Performance Counter	Est. Cost in Cycles Per Event	Events	Est. Cost in Sec
Instruction cache misses that also miss L2 cache	L2_imiss	100	1,262,433	0
Data cache load misses that also miss L2 cache	L2_dmiss_ld	100	1,015,218,197	84
Instruction count	Instr_cnt	4	65,284,655,747	217s

Given the number of events, the cost of each event, and the clock speed, the number of seconds consumed by each type of event can be estimated.

When run with 32 threads, the code uses about 320 seconds of user time and about 15 seconds of real time. Of those 320 seconds, it is estimated that 84 seconds was spent waiting for data from the second-level cache, and about the same time again was spent waiting on data from memory. So about half the total user time is spent stalled waiting on memory

In 320 seconds a single thread issuing an instruction every four cycles would have issued about 96 billion instructions. The estimated runtime of a code executing 65 billion instructions is 217 seconds. The code used about 68% of the available instruction budget.

A single thread running for 320 seconds would have three cycles of every four spent stalled while three other threads use the core. This represents 288 billion cycles of potential stall, or 240 seconds. The code has ~170 seconds of total stall due to cache misses, which corresponds to a utilization of about 71% of the stall budget.

The conclusion for this particular code is that it is not issuing as many instructions as it could potentially issue, but it is also not suffering from as much memory stall as it could be. Looking at it another way, the number of stall cycles could be increased beyond what it currently is before it would have an impact on the runtime of the application.

In terms of optimization, reducing the stall cycles on a single thread will cause that thread to potentially execute faster. The thread will be more frequently ready to take available instruction slots. However, the thread will only be able to take these instruction slots if the other threads are not using them. So the thread will not gain the entire benefit from the reduction in stall cycles.

Probably the most important metric to examine is the total number of instructions executed by all threads per cycle. It will be possible to improve throughput by avoiding stall events only if instruction slots are available. If no instruction slots are available, then any performance improvements must come from a reduction in instruction count.

9.3.3 Collecting Instruction Count Data

Since instruction count is probably the main metric that determines the throughput of an application, it is important to be able to measure the instruction frequencies for the various parts of the application. There are multiple ways to do this:

- Use Performance Analyzer to profile the instruction count hardware performance counter. This option is easy to perform but relies on the hardware performance counter generating an overflow event. This mechanism is available on UltraSPARC T2, but not on UltraSPARC T1.
- Use the BIT tool (`http://cooltools.sunsource.net/bit/`) to extend Performance Analyzer in order to gather instruction count data. This is discussed in *Gathering Instruction Counts With BIT* on page 137.
- Use the `ifreq` tool, which is shipped with the SHADE emulation library (`http://cooltools.sunsource.net/shade/`). This tool can also capture instruction frequency, but it is currently not possible to import the output from the tool into Performance Analyzer.
- Use `cpustat` and `cputrack` to get instruction count information. The `corestat` tool uses instruction counts to report the utilization of the virtual cores.

9.3.4 Strategies for Parallelization

An application can be distributed over multiple cores in a number of ways:

- **Multiprocess.** A multiprocess application has a number of separate executables that are run simultaneously to form a single “application.” The executables typically communicate through mechanisms like signals or messages, or perhaps through the network stack. The advantage of using multiple processes is that each process is independent and can potentially be restarted if it fails, so a single failure does not necessarily bring down the whole application.
- **Multithread.** A multithreaded application is a single executable that spawns a number of threads that do the work. In a Web server, each thread might be responsible for handling a new page request that comes in. In a more complex system, each thread may be assigned a specific task. In a

computationally intense workload, each thread might compute part of the problem. Since all the threads share the same memory, one concern is that an error in a single one of those threads may cause the entire application to crash. There are two common ways of coding multithreaded applications, PThreads and OpenMP, both of which are discussed later.

- **Multiple systems.** It is possible to spread one application over multiple systems. In some situations this can be achieved by replicating the one executable and installing on multiple systems, then placing these systems behind some kind of load balancing mechanism—this is typically how a large Web site might be hosted—with many systems running an identical software stack.

  Another way of using multiple systems is an extension of multiprocess, with the processes being placed on different machines. This is commonly achieved with TCP/IP as an interconnect. In the high-performance computing domain, it is usual to use the message-passing interface (MPI) to produce an application that partitions work over multiple systems. MPI is beyond the scope of this text. The Sun product that supports it is named ClusterTools (`http://www.sun.com/software/products/clustertools/`).

9.3.5 Parallelizing Applications With POSIX Threads

POSIX Threads, or PThreads, is often used to produce multithreaded applications. It has a rich API both to control the threads and to provide various synchronization mechanisms (such as mutex locks) that are necessary in order to develop applications that produce the correct results. A simple example PThread application is shown in CODE EXAMPLE 9-26.

CODE EXAMPLE 9-26 Simple PThread Example

```
#include <stdio.h>
#include <pthread.h>

void *do_work(void *var)
{
  printf("Thread number %i\n",(int)var);
}

void main()
{
  pthread_t threads[5];
  for (int i=0; i<5; i++)
  {
     pthread_create(&threads[i],0,do_work,(void*)i);
  }
  for (int i=0; i<5; i++)
  {
     pthread_join(threads[i],0);
  }
  printf("All threads completed\n");
}
```

In the application, each call to `pthread_create` creates a new thread. The new thread executes the `do_work` routine. A single parameter is passed into the routine. This parameter contains the value of the variable `i` when the thread was created. Although it's tempting to pass the address of the variable `i`, unfortunately this does not work since the variable may have changed value before the thread starts.

The only thing that each thread does is print its identifier. After that, the thread exits. The `main` thread calls `pthread_join` to wait until each thread has completed its work. Once all the threads have completed, the master thread prints a message before exiting.

To compile the application, you must include the compiler flags `-mt` and `-lpthread` for versions of Solaris prior to version 10. With Solaris 10, the threading library was combined with `libc`, so you no longer need to explicitly link in `libpthread`. The steps of compiling and running the application on a Solaris 10 platform are shown in CODE EXAMPLE 9-27.

CODE EXAMPLE 9-27 Compiling and Running PThread Example

```
$ cc ex.9.26.c -mt
$ a.out
Thread number 0
Thread number 2
Thread number 4
Thread number 3
Thread number 1
All threads completed
```

When the application is run, the threads will not run in a deterministic order. In the case shown, the threads clearly do not run in the order in which they were created.

9.3.6 Parallelizing Applications With OpenMP

OpenMP is a way of parallelizing an application, using directives that are inserted into the source code of an application. The OpenMP approach has a number of advantages:

- The directives are relatively easy to use and hide the complexity of managing the threads that are necessary for parallelization.
- The compiler will recognize the directives only when a particular flag (`-xopenmp`) is specified. If the flag is not present, the original source is preserved, so a single source base can produce both a serial and parallel version of the application. This facility is very helpful in determining whether a bug is due to the serial logic of the application or due to something introduced during parallelization.
- The directives need be applied only to the parts of the code that are to be parallelized, so the rest of the code can be left unchanged. That way, developers can incrementally parallelize the application, picking only the parts of the application that will benefit from parallelization and leaving the rest of the application unchanged.

The principal constraint with using OpenMP to parallelize applications is that it is most effective at being used to parallelize code containing loops. However, the recently released OpenMP 3.0 specification does provide a task-based approach.

An example of using OpenMP directives to parallelize a simple application is shown in CODE EXAMPLE 9-28

CODE EXAMPLE 9-28 Example OpenMP Code

```
#include <stdio.h>

void main()
{
  #pragma omp parallel for
  for (int i=0; i<4; i++)
  {
    printf("Iteration %i\n",i);
  }
}
```

The OpenMP directive `parallel for` tells the compiler to produce a parallel version of the following loop. Each thread takes a separate part of the loop; the number of threads is controlled by the environment variable `OMP_NUM_THREADS`, although there also exist API functions for the program to discover or change the number of threads at runtime.

To compile the application, you must use the flag `-xopenmp`. The flags `-xvpara` and `-xloopinfo` can be used to request the compiler to output information about the parallelization that has been achieved. The sequence of compiling and running the example code is shown in CODE EXAMPLE 9-29

CODE EXAMPLE 9-29 Compiling and Running Simple OpenMP Example

```
$ cc -xopenmp -xvpara -xloopinfo -O ex.9.28.c
"ex.9.28.c", line 6: PARALLELIZED, user pragma used
$ export OMP_NUM_THREADS=4
$ a.out
Iteration 1
Iteration 0
Iteration 3
Iteration 2
```

A more complex example is shown in CODE EXAMPLE 9-30. This code contains two parallel regions: the first is a straightforward loop that sets the contents of the array `values`; the second parallel region is a reduction.

CODE EXAMPLE 9-30 Example of a Reduction With OpenMP

```
#include <stdio.h>

void main()
{
  double total=0;
  double values[1000000];
  int i;
  #pragma omp parallel for
  for (i=0; i<10000000; i++)
  {
     values[i]=i;
  }
  #pragma omp parallel for reduction(+: total)
  for (i=0; i<10000000; i++)
  {
    total += values[i];
  }
}
```

A reduction is a situation in the code when a set of values is reduced to a single value. Addition and subtraction are examples of this, but other operations like finding the maximum value are also reductions. The problem with reductions like addition and subtraction is that the order in which the computation is performed in the parallel case may be different from the order in which the computation is performed in the serial case.

For example, imagine that a program has to add a list of floating-point numbers that happens to be sorted in order from the largest number to the smallest number. With floating-point computation, when a very small number is added to another number that is of much greater magnitude, the resulting number may be identical to the original large number. So when the series of numbers is summed, the result will not reflect the values of the small numbers. Now, imagine in the parallel case that one thread gets to add up all the large numbers and another thread gets to add up all the small numbers. When all the small numbers are added together, they may become sufficiently large to actually have an impact when added to the sum of all the large numbers. Although this example is entirely contrived, a similar scenario can lead to small differences in the results.

9.3.7 Using Autoparallelization to Produce Parallel Applications

One final approach to parallelization is to let the compiler parallelize the application. Two flags control this. The flag `-xautopar` enables automatic parallelization. Under this flag, the compiler tries to identify blocks of code that can be parallelized. The compiler can also parallelize code that contains reductions. As discussed in *Parallelizing Applications With OpenMP* on page 162, reductions can alter the results of some floating-point computations. Consequently, the compiler requires explicit permission from the user to generate reduction code. The user gives permission with the flag `-xreduction`. An example of code that contains a reduction is shown in CODE EXAMPLE 9-31.

CODE EXAMPLE 9-31 Example of Code Containing Reduction

```
double total(double *values, int size)
{
  double t=0;
  for (int i=0; i<size; i++)
  {
    t+=values[i];
  }
  return t;
}
```

The sequence of compiling is shown in CODE EXAMPLE 9-32.

CODE EXAMPLE 9-32 Compiling With Automatic Parallelization

```
$ cc -c -O -xautopar -xreduction -xvpara -xloopinfo ex.9.31.c
"ex.9.31.c", line 4: PARALLELIZED, reduction, and serial version generated
```

The number of threads used by an autoparallelized application is set in the same way as for an OpenMP application, using the environment variable `OMP_NUM_THREADS`.

9.3.8 Detecting Data Races With the Thread Analyzer

A data race occurs when two (or more) threads attempt to access the same memory location at the same time and one or more of those accesses is a write. As a simple example, consider two threads. Both threads are going to read, increment, and write back the same variable `total`. The sequence of operations is shown in CODE EXAMPLE 9-33. Note that the variable `total` is declared as `volatile`, which stops the compiler from holding the variable in a register and ensures that the variable is loaded from memory before the increment and stored back afterwards.

CODE EXAMPLE 9-33 Example of Code Containing a Data Race

```
#include <stdio.h>
#include <pthread.h>

volatile int total=0;

void *do_work(void *var)
{
  for (int i=0; i<100000; i++)
  {
    total+=1;
  }
}

void main()
{
  pthread_t threads[2];
  pthread_create(&threads[0],0,do_work,0);
  pthread_create(&threads[1],0,do_work,0);
  pthread_join(threads[0],0);
  pthread_join(threads[1],0);
  printf("total = %i\n",total);
}
```

The results of compiling and running the program twice are shown in CODE EXAMPLE 9-34. The first time the program runs, it reports a result of 63,165, but the second time the same code reports a result of 132,400. The correct answer would be for each thread to increment the variable 100,000 times, yielding a value of 200,000.

CODE EXAMPLE 9-34 Compiling and Running Code Containing a Data Race

```
$ cc -O ex.9.33.c -mt -o ex.9.33
$ ex.9.33
total = 63165
$ ex.9.33
total = 132400
```

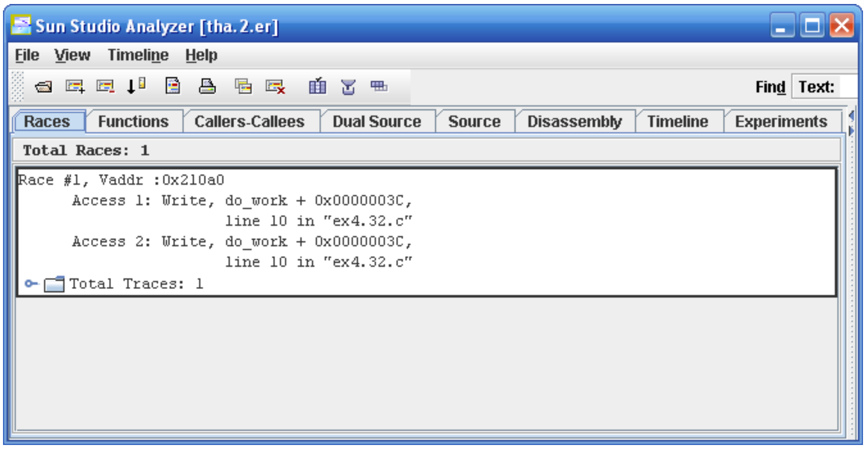
Although both threads attempted to increment the variable, only one of the increments was actually recorded. In common with many data corruption bugs, this problem can be hard to detect because the result of the error will probably be detected far from the location of the error in the code.

Fortunately, Sun Studio 12 includes the Thread Analyzer, which detects and reports errors in multithreaded code. To use the tool, you must compile the program with the compiler flag `-xinstrument=datarace`, which tells the compiler to include the necessary instrumentation to record the errors. It is recommended that you also include the debug flag `-g`. The code is then run under `collect` with the option `-r on`. These steps are shown in CODE EXAMPLE 9-35.

CODE EXAMPLE 9-35 Compiling for Data Race Detection

```
$ cc -g -O -xinstrument=datarace ex.9.33.c -mt -o ex.9.33
$ collect -r on ex.9.33
Creating experiment database tha.2.er ...
total = 167277
```

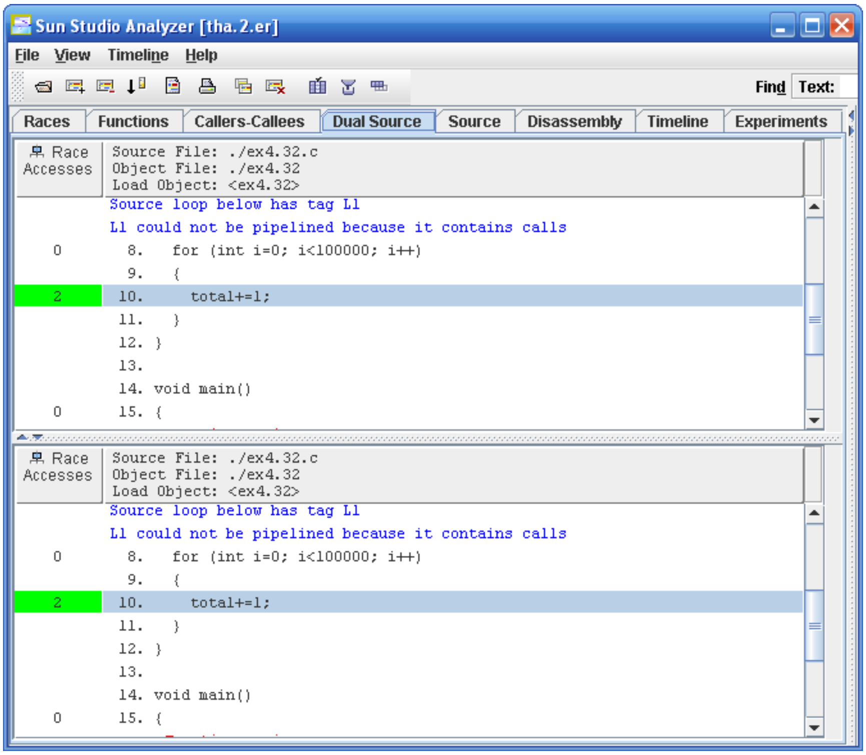
Once the run has completed, the resulting experiment can be examined with the Thread Analyzer GUI, as shown in FIGURE 9-9. The initial display is a list of all the potential data races in the code.


Sun Studio Analyzer [tha.2.er]
File View Timeline Help
Find Text:
Races Functions Callers-Callees Dual Source Source Disassembly Timeline Experiments

```
Total Races: 1
Race #1, Vaddr :0x210a0
      Access 1: Write, do_work + 0x0000003C,
                       line 10 in "ex4.32.c"
      Access 2: Write, do_work + 0x0000003C,
                       line 10 in "ex4.32.c"
   Total Traces: 1
```

FIGURE 9-9 Data Races Shown in the Thread Analyzer GUI

The user can select any one of the data races and see which lines of source are causing the problem, as shown in FIGURE 9-10. The dual-pane display shows the source code from the two places where the same memory location is being accessed. In this particular example, the two accesses come from the same line of source being executed by two different threads.


Sun Studio Analyzer [tha.2.er]
File View Timeline Help
Find Text:
Races Functions Callers-Callees Dual Source Source Disassembly Timeline Experiments

```
Race      Source File: ./ex4.32.c
Accesses  Object File: ./ex4.32
          Load Object: <ex4.32>
          Source loop below has tag L1
          L1 could not be pipelined because it contains calls
   0         for (int i=0; i<100000; i++)
             {
   2           total+=1;
             }
          }

          void main()
   0      {
```

```
Race      Source File: ./ex4.32.c
Accesses  Object File: ./ex4.32
          Load Object: <ex4.32>
          Source loop below has tag L1
          L1 could not be pipelined because it contains calls
   0         for (int i=0; i<100000; i++)
             {
   2           total+=1;
             }
          }

          void main()
   0      {
```

FIGURE 9-10 Lines of Source Containing Data Race

There are two typical solutions to data races: locking or rearchitecting.

The most obvious solution is to place some form of locking around the problem code so that only one thread can update the memory at a time. In code parallelized with PThreads the common form of locking is a mutex lock. For OpenMP there exists a critical section directive that ensures that only one thread executes a region of code at any one time.

An alternative to mutexes is the use of atomic operations. An atomic operation is one that completes as if it were a single operation; that is, no other thread can interrupt and corrupt the operation. Solaris 10 has code for a number of atomic operations included in libc. The range of supported operations can be found under `man atomic_ops`.

The section on *Avoiding Data Races* on page 169 explores the costs of these various methods of sharing data among threads.

The second solution to data races is to rearchitect the code so that data is not shared among threads. The advantage of this approach is that it avoids the synchronization codes altogether, but this avoidance is not always possible to achieve in practice.

9.3.9 Avoiding Data Races

The most obvious way of avoiding the data race shown in CODE EXAMPLE 9-33 is to place a mutex lock around the update of the variable. This structure ensures that only a single thread updates the variable at any one time. Because the majority of the work that the threads perform is the updating of the variable protected by the mutex lock, the performance will be slower than with the serial case. The modified code is shown in CODE EXAMPLE 9-36.

CODE EXAMPLE 9-36 Avoiding Data Races By Means of Mutex Locks

```
#include <stdio.h>
#include <pthread.h>

volatile int total=0;
pthread_mutex_t total_mutex;

void *do_work(void *var)
{
  for (int i=0; i<100000; i++)
  {
    pthread_mutex_acquire(&total_mutex);
    total+=1;
    pthread_mutex_release(&total_mutex);
  }
}

void main()
{
  pthread_t threads[2];
  pthread_mutex_init(&total_mutex,0);
  pthread_create(&threads[0],0,do_work,0);
  pthread_create(&threads[1],0,do_work,0);
  pthread_join(threads[0],0);
  pthread_join(threads[1],0);
  pthread_mutex_destroy(&total_mutex);
  printf("total = %i\n",total);
}
```

The sequence of compiling and running the code is shown in CODE EXAMPLE 9-37. The code produces the correct answer of 200,000.

CODE EXAMPLE 9-37 Compiling and Running Code Containing a Mutex Lock

```
$ cc -O -o ex.9.36 -mt -lpthread ex.9.36.c
$ ex.9.36
total = 200000
```

It is interesting to compare the performance of an SMP and a CMT system on the above code. If the code is edited so that only a single thread is created, the timing on a 900 MHz V880 and on a 1.2 GHz T2000 is roughly the same. However, when two threads are created, the performance is very different, as shown in TABLE 9-5.

TABLE 9-5 Comparing Single and Multiple Thread Access to Mutex

Platform	Single Thread	Two Threads, Twice the Work
900 MHz V880	1.8 s	10.0 s
1.2 GHz T2000	2.5 s	6.2 s

In both cases, having to contend for the mutex causes the application to run slower than in the case where the mutex is uncontended. However, the V880 takes five times longer when there are two threads than the situation in which there is a single thread, compared to the CMT T2000 which takes just over twice as long—that is, it takes twice as long to do twice the work.

The reason for this difference in performance is that on the V880 multiple threads have to share the mutex through memory. For this system, memory latency is a couple of hundred cycles. In contrast, the T2000 can share the mutex through the second-level cache, or perhaps even through the first-level cache. Cache latency is substantially less than memory latency.

An alternative way of modifying the code is to use an atomic operation. Atomic operations are included in `libc` in Solaris 10 (see `man atomic_ops`). These are actually short routines that behave as if the operation were atomic (that is, cannot be interrupted by another thread). The modified code is shown in CODE EXAMPLE 9-38.

CODE EXAMPLE 9-38 Using Atomic Operations

```
#include <stdio.h>
#include <pthread.h>
#include <atomic.h>

volatile unsigned int total=0;

void *do_work(void *var)
{
  for (int i=0; i<10000000; i++)
  {
    atomic_add_32(&total,1);
  }
}

void main()
{
  pthread_t threads[2];
  pthread_create(&threads[0],0,do_work,0);
  pthread_create(&threads[1],0,do_work,0);
  pthread_join(threads[0],0);
  pthread_join(threads[1],0);
  printf("total = %i\n",total);
}
```

The difference in performance on the T2000 is impressive. The two-threaded version of the code completes in about 0.8 s. Interestingly, the two-threaded version completes twice as much work in the same wall time, but uses twice the user time. The atomic operations have a much lower cost than acquiring and releasing the mutex, which leads to much better scaling. Of course, an SMP system would still suffer a substantial performance penalty from using atomic operations in this way because the sharing would have to take place through memory.

Two directives in OpenMP handle the same situation. CODE EXAMPLE 9-39 shows an OpenMP code containing a data race. This code has two loops that are parallelized by OpenMP. The first loop sets up an array, and the second loop calculates the sum of all the values in the array.

CODE EXAMPLE 9-39 OpenMP Code Containing a Data Race

```
#include <stdio.h>

void main()
{
  double a[100000];
  int i;
  double total=0;
  #pragma omp parallel for
  for (int i=0; i<100000; i++)
  {
   a[i]=i;
  }
  #pragma omp parallel for shared(total)
  for (int i=0; i<100000;i++)
  {
    total+=a[i];
  }
  printf("total = %f\n",total);
}
```

The process of compiling and running the code with one and two threads is shown in CODE EXAMPLE 9-40. The compiler flag `-xvpara` causes the compiler to emit a warning about the potential data race in the second loop. The compiler flag `-xloopinfo` causes the compiler to emit information about the loops it has parallelized. Unsurprisingly, the code produces incorrect results when run with more than one thread.

CODE EXAMPLE 9-40 Compiling and Running OpenMP Program Containing Data Race

```
$ cc -O -xopenmp -xvpara -xloopinfo -o ex.9.39 ex.9.39.c
"ex.9.39.c", line 13: Warning: inappropriate scoping
      variable 'total' may be scoped inappropriately as 'shared'.
      write at line 16 and write at line 16 may cause data race

"ex.9.39.c", line 9: PARALLELIZED, user pragma used
"ex.9.39.c", line 14: PARALLELIZED, user pragma used
$ export OMP_NUM_THREADS=1
$ ex.9.39
total = 4999950000.000000
$ export OMP_NUM_THREADS=2
$ ex.9.39
total = 3749975000.000000
```

Two directives can be used to avoid the data race condition. The first is the critical section directive. This directive specifies that a section of code should be executed by only one thread at a time. The modified code is shown in CODE EXAMPLE 9-41.

CODE EXAMPLE 9-41 Avoiding Data Race With the OpenMP Critical Section Directive

```
#include <stdio.h>

void main()
{
  double a[100000];
  int i;
  double total=0;
  #pragma omp parallel for
  for (int i=0; i<100000; i++)
  {
   a[i]=i;
  }
  #pragma omp parallel for shared(total)
  for (int i=0; i<100000;i++)
  {
    #pragma omp critical
    {
      total+=a[i];
    }
  }
  printf("total = %f\n",total);
}
```

When this code is run, it produces the correct answer in both the serial and parallel cases. This code is analogous to using a mutex lock to protect access to the variable `total`. However, a more efficient directive can be used in this situation. That is the `atomic` directive, which specifies that the next operation be performed atomically. The code using this directive is shown in CODE EXAMPLE 9-42.

CODE EXAMPLE 9-42 Avoiding Data Race With the OpenMP Atomic Directive

```
#include <stdio.h>

void main()
{
  double a[100000];
  int i;
  double total=0;
  #pragma omp parallel for
  for (int i=0; i<100000; i++)
  {
   a[i]=i;
  }
  #pragma omp parallel for shared(total)
  for (int i=0; i<100000;i++)
  {
    #pragma omp atomic
    total+=a[i];
  }
  printf("total = %f\n",total);
}
```

9.3.10 Considering Microparallelization

CMT processors have two very significant advantages over more traditional approaches. The first advantage is having a large number of threads. The second is that these threads typically have a low cost of synchronization—the synchronization takes place at a level of cache that is very close to the core.

The combination of these two advantages leads to the idea that it should be possible on CMT processors to parallelize sections of code where the synchronization costs on traditional processors outweigh the performance gains from using multiple threads. Microparallelization is the approach where a small quantum of work is distributed among multiple threads; the possibility of doing this relies on the synchronization costs being kept low.

A very general framework for microparallelization can be described as follows:

- Each thread should identify the next quantum of work that it will perform.
- Each thread checks that it is safe to perform its quantum of work.
- Once the work is completed, the thread updates status to enable dependent threads to start.

An advantage of this framework is that it can handle the constraint that one item of work may be unable to start until a previous item has completed. This flexibility significantly increases the number of situations to which microparallelization can be applied. CODE EXAMPLE 9-43 shows a code with potential dependencies. The danger is that same element of the `values` array could be pointed to by multiple indexes held in the `index` array. This example happens to be a type of reduction, so an atomic add operation could be used to avoid the potential data race. Microparallelization could also perform the same task, although with substantially more overhead. The objective of the example is to demonstrate the structure rather than the potential performance gains.

CODE EXAMPLE 9-43 Code Containing Possible Dependencies Between Iterations

```
double calc(int *index, double *values)
{
  for (int i=0; i<10000; i++)
  {
     values[index[i]]+=i;
  }
}
```

The first thing that each thread needs to do is identify which iteration it is responsible for calculating. One way of achieving this is shown in CODE EXAMPLE 9-44.

CODE EXAMPLE 9-44 Identifying Iteration to Compute

```
#include <atomic.h>
#include <pthread.h>

volatile unsigned int next;
volatile unsigned int completed;

void *do_work(void* value)
{
  unsigned int my_iteration=0;
  unsigned int next_iteration;
  /*signal that the thread is ready*/
  atomic_add_32(&completed,-1);
  /*wait for all threads to be ready*/
  while (completed>=0){}
  /*main loop*/
  while (next<10000)
  {
    next_iteration=atomic_cas_32(&next,my_iteration,my_iteration+1);
    if (next_iteration==my_iteration)
    {
      do_iteration(my_iteration);
    }
    my_iteration=next_iteration;
  }
}

void main()
{
  pthread_t threads[2];
  next=0;
  completed=2;
  pthread_create(&threads[0],0,do_work,0);
  pthread_create(&threads[1],0,do_work,0);
  pthread_join(threads[0],0);
  pthread_join(threads[1],0);
}
```

The example code creates two threads. It uses the variable `completed` to hold the threads until both have been created and are running. Once the two threads are running, they work through the `main` loop, each attempting to get the next iteration of work. The atomic Compare and Swap (`cas`) instruction ensures that only one thread will execute each iteration. Once a thread has acquired the right to perform a particular iteration, it calls the do_iteration routine to handle the work. The do_iteration routine is shown in CODE EXAMPLE 9-45.

CODE EXAMPLE 9-45 Code to Ensure Safety and Perform Work

```
int index[10000];
int values[10000];
...
void do_iteration(unsigned int this)
{
  int i;
  int ok;
  /*check that it is safe to start iteration*/
  do
  {
    ok=1;
    for (i=completed; i<this; i++)
    {
     if (index[i]==index[this])
     {
       ok=0;
     }
    }
  } while (ok==0);
  /*do calculation*/
  values[index[this]]+=this;
  /*Indicate work completed*/
  while (atomic_cas_32(&completed,this,this+1)!=this){}
}
```

The code that does the work must first check that no iterations in flight will use the value of the index for this iteration. Once this test passes, the code can perform the calculation safe in the knowledge that no other thread will read or modify the element being computed. Once the calculation is completed, the thread updates the variable completed, which indicates which iterations have been successfully computed.

One final snippet of code is necessary to initialize the arrays index and values. This is shown in CODE EXAMPLE 9-46. The code deliberately sets up the array index to point to only the first eight elements in the array values.

CODE EXAMPLE 9-46 Code to Initialize the Arrays

```
  for (int i=0; i<10000; i++)
  {
    values[i]=0;
    index[i]=i&7;
  }
```

This demonstration of microparallelization proves that the work can be split among multiple threads. However, it is a poor demonstration for performance gains because the amount of work performed by each thread is dwarfed by the synchronization costs. Obviously, a more complex calculation for each iteration would lead to more time being spent in performing the work and a correspondingly smaller proportion of the time lost to synchronization costs.

9.3.11 Programming for Throughput

CMT processors introduce a change in the way that developers should approach application performance. The traditional approach is to produce a serial application, tune the application by eliminating stalls such as cache misses, then once the stall conditions have been eliminated, to consider parallelizing the application to further improve performance. Unfortunately, this approach leads to the situation in which the application is already very complex by the time parallelization is considered, so parallelization is often not a trivial task.

Collecting Instruction Count Data on page 159 discussed how, for CMT processors, stall events are no longer the dominant factor that determine performance and that instruction count is more likely to be the factor to control to improve performance.

CMT processors also have the advantage, outlined in *Avoiding Data Races* on page 169, that the cost of synchronizing multiple threads is much lower than is found on a traditional SMP-type system. This observation fits well with the fact that a large number of virtual processors are ready to execute these threads.

These factors combine in the observation that for CMT processors (and by extension most future processors, since almost all future processors will have some multithread capability) the approach to obtain maximal performance is to develop multithreaded applications, which are then subsequently tuned to reduce instruction count (and possibly stall events). Fortunately, designing multithreaded applications from the start avoids the situation alluded to earlier in which the capability of using multiple threads has to be “bolted on” to an existing serial application.

The conclusion is that CMT processors promote the design of parallel applications. They reduce, or remove, the traditional barriers to parallelization, such as synchronization costs, reduce the importance of eliminating stall events, and provide a large number of threads to support the application.

CHAPTER **10**

System Simulation, Bringup, and Verification

The SPARC Architectural Model (SAM) is a flexible, reconfigurable virtual platform for system simulation. It allows users to model various SPARC-based systems. The virtual platform has an important role for system exploration, software bringup, and development when new hardware is not ready—users can run software on the simulated system even though hardware is not available. The virtual platform can boot Open Boot PROM (OBP) and the Solaris operating system, and it creates an environment very similar to that created by real hardware.

Other benefits may accrue from the use of the virtual platform: it provides much better observability than the real hardware and it creates a much better environment for debugging and system exploration.

SAM is intensively used for collecting architectural traces from benchmark workloads (TPCC, SPECWeb, ICPERF, SPECJAppServer, ICPERF, SPECJbb, SPEC CPU, and the like). It is also used for performance modeling, system bringup, system software and device driver development, and RTL verification.

This chapter contains the following sections:

10.1 SPARC Architecture Model

SAM simulates a system consisting of the following components:

- One or more SPARC CPUs
- Physical memory (RAM)
- Serial console device
- Disk subsystems (for example, SCSI, FibreChannel)
- Network cards (for example, Gigabit-Ethernet device drivers—GEM and Cassini)

Since SAM can run the Solaris OS, most of the high-level functionality built on top of the OS kernel is available in the simulation—TCP/IP, file systems, kernel modules, kernel debugger, multiple users, and so forth.

SAM is organized as a collection of dynamically loaded modules that simulate various devices. To initialize the simulator, the main module processes a special configuration file to instantiate and configure other device modules that constitute the simulated system. The configuration file contains a set of general configuration parameters, for example: CPU and system clock frequency; number of worker threads on the host machine to simulate virtual processors; platform- and environment-specific parameters; CPU; and debug level. It also provides names and configuration parameters of the CPU and device modules in device tree. The specification is somewhat similar to the Solaris device tree; each line describes a device module and contains parameters for simulated devices.

SAM can be run in multithreaded mode. Each simulated virtual processor and I/O device controller can be bound to a thread running on a separate host machine processor. SAM is optimized to simulate a multiprocessor system on a multiprocessor host machine. A configuration file maps a set of simulated virtual processors to a set of threads on the host machine. Each simulated virtual processor is allowed to advance independently for a very short period of time. However, a virtual processor is never simulated by more than one simulation thread. Device models should have an appropriate synchronization mechanism to guarantee data integrity in a multithreaded simulation environment.

SAM can manage simulated time across all host machines. It implements an event queue to keep track of when the data from the simulated devices should be processed. Typical simulation speed is approximately 6–10 MIPS.

SAM supports system dump/restore capabilities. Simulation can be stopped at any time to create a checkpoint. Later, simulation can be restored and resumed from that checkpoint. This feature is useful for a quick restart after a long warm-up run and also for debugging.

SAM has a built-in user interface (UI) based on a simple and easy-to-use command language. It has a rich set of basic commands, for example, `run` and `stop` to control simulation, `break` to set a breakpoint, `read-reg` to examine register state. The UI also supports scripts written in Python.

Note Python was chosen because it is a powerful, portable, open-source, clear scripting language, supports object-oriented programming, has access to a large number of built-in and external libraries, and has an easy interface to code written in C or C++.

Two internal interfaces allow the simulator to be reconfigurable:

- Virtual CPU (VCPU) interface to plug-in CPU models
- Module Model interface (MMI) for I/O models. MMI is used for loading libraries, creating device instances, mapping device physical I/O addresses, enabling register-module-specific user commands, and more.

Note It is important to mention that the SAM module need not represent a real device. Some other simulator components, for example, tracer module, additional UI commands, timing model, and the like could well be implemented as dynamically loaded libraries.

The next subsections explore the following SAM features in more detail:

- SPARC CPU
- VCPU interface
- Device model interface (MMI)

10.1.1 SPARC CPU Model

SAM supports various levels of SPARC CPU models. A CPU model consists of a number of CPU cores, each of which can contain a number of virtual CPU (strand) models. The structure and functionality of the CPU model tightly reflects the actual hardware. CPU and memory models could be configured in a minimal system configuration for verification and some initial system software development.

The standard CPU model included in SAM does not model time. A single step (cycle) is implemented as Fetch, Decode, Execute, Retire the architectural state in one single operation; in other words, a CPU model is a simple instruction-level simulator. More advanced timing models are not included in standard SAM distribution.

Caches are not modeled because they are not required by the SPARC specification.

10.1.2 VCPU Interface

The VCPU interface hides the details of the virtual-processor-specific implementation. It is an abstract interface to the "virtual CPU." For a CPU with multiple strands, it represents each strand; for a CPU without strands, it represents the CPU. The CPU module is built as a shared library. A few instances of CPU objects can be created, each CPU object can have a few CPU cores, and each core can have a few strands.

The interface hides the hierarchical nature of internal CPU structures and presents a flat array of VCPU objects. The model could be partitioned to run on separate threads of the host machine—interface methods should be MP safe.

The VCPU interface has these components:

- Control interface
- System interface (memory and I/O)
- Trace interface

Note The interface to a loadable module consists of two elements: exported and imported interfaces.

Exported interfaces are routines in CPU module that are called from outside the module. Pointers to the routines exposed by the CPU module are defined in the exported interface structure.

Imported interfaces are routines that the CPU module calls to gain access to the rest of the system. Imported interfaces are defined by the imported interface structure, which is a set of pointers that the CPU module uses to send requests to the system modules.

So, the Control interface is defined in a VCPU exported interface, and System and Trace interfaces are defined in an imported interface.

10.1.2.1 Control Interface

The Control interface allows users to create, destroy, reset, save, and restore VCPU instances and to access architecture-visible state for all virtual processors in the system.

Here is an example that creates a `vcpu` instance.

CODE EXAMPLE 10-1 Creating a `vcpu` Instance

```
// open shared library, lib_name is the name of cpu shared library.
void *cpu_lib_handle = dlopen (lib_name, RTLD_LAZY|RTLD_
  GLOBAL|RTLD_PARENT);


// extract cpu lib exported interface
VCPU_GetIntfFn get_interface = (VCPU_GetIntfFn)dlsym ( cpu_
  lib_handle, "get_ex_interface" );


// obtain exported cpu interface
VCPU_ExInterface g_cpu_ex_int
get_interface ( &g_cpu_ex_intf);


//Create a vcpu instance
Vcpu * vcpu = (Vcpu*)g_cpu_ex_intf.create( &config_info, imp_intf );
```

After the `vcpu` instance is created, VCPU methods can access the `vcpu` instance state.

The Control interface has methods to read and write register state, advance simulation *n* number of instructions or cycles, set and delete breakpoints, translate address in the current `vcpu` context, and so forth.

The VCPU Control interface allows users to hook up an external source-level debugger or user-interface front-end module.

10.1.2.2 System Interface

The CPU module imports a system interface to access the rest of the system—a set of pointers that are passed to the VCPU at creation time. It has methods to access memory and I/O address space.

Memory is represented by an abstract interface to model different levels of the memory hierarchy, including L2 and L3 caches. Currently, a sparse memory model and flat memory models are in use. For performance, the memory model option is defined at build time instead of runtime.

Cache models are not provided.

10.1.2.3 Trace Interface

The Vtracer interface is an abstract interface for connecting different analyzers and performance models. The CPU model calls Vtracer methods to output complete information about architecture state changes for every instruction.

The information for every instruction is collected in a special instruction structure that accumulates state value changes for each VCPU. Separate structures collect trap information and TLB update information.

10.1.3 Module Model Interface

The Module Model interface (MMI) lets users plug a device module into the SAM simulation environment. All API function and type names are prefixed with the `mmi_` prefix. A device model is built as a shared library that is dynamically loaded into SAM during initialization.

MMI provides a common framework for the following:

- Loading a module dynamic library
- Configuring a module, notifying module instances of certain events (such as the addition or deletion of another module)
- Supplying a mechanism for `dump` and `restore` operations
- Registering user-interface commands
- Handling module export/import interfaces
- Supporting the event queue that models time
- Providing module information

The protocol of how SAM modules interact with one another depends on the agreed-on interfaces they exchange through MMI.

Part of MMI is more specific to certain type of devices, for example, host bus bridges, some other devices (like the system interface unit (SIU), ROM, TOD, consoles) that are attached to the simulated system bus (interconnect). Actions of that part of MMI include the following:

- Accessing memory
- Signaling an interrupt
- Mapping the portions of I/O physical address space
- Mapping some specific ASI registers

The MMI interface has global routines available to each module to access simulated memory and to send interrupts to the destination virtual processor.

Overall, the MMI interface is generic enough to provide a foundation for building a flexible system simulator.

10.1.3.1 SAM Configuration File

A SAM configuration file describes all devices in the simulated system. The sysconf directives in the configuration file specify the device module, instance name, and its associated properties. Each sysconf line specifies the module to be loaded, if necessary, and the name of the device node to be instantiated, followed by the properties associated with the instance in the form of *name=value* pairs. A device model built as a shared object (`.so` file) typically has only one module that plugs into SAM. Internally, within the device model, there could be a hierarchy of components.

An instance is what actually represents a device in SAM. There are generally multiple instances of a given device module. Each instance can be referenced with an opaque `mmi_instance_t` handle. The handle is used by the module to refer to its own instance as well as to other instances to get an exported interface. SAM keeps a list of all available loaded device modules. The module handle can be retrieved by module name.

To properly instantiate a device node, SAM calls the instance initialization function registered by the device module.

10.1.3.2 Module Loading and Unloading

Modules can be loaded in any order as specified in the configuration file. In some cases (the tracer module, for example), modules can be unloaded at any time. Each module is notified when another module changes status. To enable interaction with other modules, each module has to obtain an exported interface—pointers to the interface routines. When a module is unloaded, other modules should receive the corresponding configuration change notification and remove their interface pointers to the unloaded module.

10.1.3.3 Module Initialization

Each module has an initialization method that should be called to create an instance of this module. So, the first thing each module should do when its library is loaded is register with the SAM simulator a pointer to the initialization method. The instance initialization function is called by SAM whenever a new instance of the module is created—usually when the corresponding line in the system configuration file is processed. An opaque instance handler is used in MMI calls to access methods for a particular module instance. Normally, the instance initialization routine will parse config parameters from the configuration line to find out which address range the device module is supposed to be mapped to. It also registers a few other routines to be called for the following purposes:

- To obtain module information. This function is called through the `modinfo` user-interface command. It prints any important information about the module.
- To get a module interface. Here, *interface* means an agreement between the client of the interface and the provider.
- To notify modules that another module is loaded or unloaded.
- To read or write when the I/O address range to which the module is mapped is accessed.

10.2 System Configuration File

A SAM module is a shared object (ELF file), typically containing the implementation of a particular device model: for example, a SAS controller; or perhaps a conceptual simulation object, for instance, a PCIE bus; or even a completely pseudo device, for instance, an LL (Local Loopback file system) device. Any dynamically loaded shared object that follows the semantics of SAM's MMI (`mmi.h`) can be called a SAM module.

`sysconf` is a module specification directive in SAM. It specifies which module to configure in a particular simulation instance and how to instantiate it. It also indirectly functions to create a device tree hierarchy in SAM (which closely resembles that in Solaris). The directives are specified in a configuration file that is processed by SAM, one per line. So, all together, the sysconf directives specify the simulated system configuration.

Once loaded, a device module cannot be unloaded at the middle of the run.

10.2.1 The `sysconf` Directive Format

`sysconf` *mod-name instance-name* `[`*arg-name*`[=`*arg-value*`]]*` `[-d`*debug-level*`]`
`sysconf -p` *mod-path*

where

- *mod-name* — The name of a SAM module. It is the identifier of the shared object without the trailing `.so`; for example, `sas` is the *mod-name* for a SAS controller whose implementation resides in `sas.so`.

- *instance-name* — The unique instance name of this module in a SAM configuration. It is possible to have multiple instances of a module in a single configuration (if allowed by module semantics). All instance names in a particular simulation instance (whether the same or a different module) must be unique; otherwise, SAM flags a fatal error. Many modules optionally support a UI command with the same name as its instance name (run **`help`** *instance-name* to verify).
- *arg-name* — The name of a configuration parameter supported by a module. A module can have zero or more such parameters. An argument can be mandatory or optional. Its value can be boolean (that is, its presence or absence sets the parameter value) or it can have an *arg_value*.
- *arg-value* — The value part of the *name=value* pair of a module's configuration parameter. The semantic and legal values of *arg-value* are defined by a particular module. For example, a console module may support a configuration parameter `bg_color`. A user may set the value of this parameter to a valid value, such as `black`. This would appear in the `sysconf` directive line as `bg_color=black`.

  The configuration parameters and their valid values vary from module to module and are part of the module-specific documentation. The configuration parameters are interpreted by individual modules.
- `-d`*debug-level* — An optional debug flag supported by a particular module. The *debug-level* can be 0, 1, or 2, with 0 being debug off (the same as when `-d` is not present), 1 (verbose), and 2 (verbose++).

  The semantics of level and printed information varies from module to module. Typically, a module would export a UI command that can modify the flag value during runtime.
- *mod-path* — In a typical SAM installation, the modules reside in *install-dir*`/lib` directory and the SAM binary resides in *install-dir*`/bin`. SAM by default looks for modules in `$ORIGIN/../lib`. However, a user can override the default by specifying `-p` *mod-path*, whereby SAM will first look into the *mod-path* directory for modules. Each `sysconf -p` adds a lookup path, searched in LIFO order, the last being the default `$ORIGIN/../lib`. It is recommended that the default search path be used but that this feature not be used under normal circumstances.

Note The `sysconf` command specification for CPUs is a little different. *mod-name* is always `cpu`. The *arg-value* of *arg-name* `cpu-type` determines the CPU library to load. For example, the T2 CPU is configured as

```
sysconf cpu cpu-type=riesling_n2_vcpu
name=cpu0 clock-frequency=1417000000 id=0
```

The CPU library is `libriesling_n2_vcpu.so` in this case, loaded from `$ORIGIN/../lib`.

10.2.2 Examples

This section presents examples of sysconf directives.

The following directive loads `ioram.so` and creates an instance of `ioram` called `obp`. (`ioram` is a module that implements ROM.) The module loads file `openboot.bin` in simulated ROM at PA FF F008 0000_{16}. The size of the ROM is 80000_{16} bytes.

```
sysconf ioram  obp start_pa=0xfff0080000 size=0x80000
 file=openboot.bin
```

The following two examples show how a device tree hierarchy is created with a sysconf directive. Here a SAS controller `sas0` is connected to a PCIE bus `pcie_a`, which is connected to host bus bridge `piu`.

The directive below creates an instance of `pcie_bus` module and names it `pcie_a`. The bus is upstream. The bridge instance name is `piu`.

```
sysconf pcie_bus    pcie_a  bridge=piu
```

The directive below creates a SAS controller with name `sas0` and connects it to PCIE bus named `pcie_a`. The controller is connected as device 0, function 0 on the PCIE bus. The `targets` parameter provides information about the target SCSI disks attached to `sas0`.

```
sysconf sas sas0 bus=pcie_a dev=0 fun=0 targets=sasdisk.init
```

10.2.3 Simulated Time in SAM

The STICK register provides a synchronized, system-wide clock that is used for high-resolution time-stamping. Earlier UltraSPARC processors that did not have the STICK register used instead a counter, called the TICK register, driven at the CPU's clock speed. Consequently, processors operating at different speeds would show their TICK registers incrementing at different rates. To model the time in the simulator, we must define a policy for how and when the value of the STICK register should be updated.

SAM has two parameters that affect how time is maintained in SAM.

- `mips` — Determines how often the CPU's STICK register is incremented. For example, if `mips` is set to *M*, SAM will increment STICK every *M* instructions. If there is more than one CPU, STICK is incremented after each CPU has executed *M* instructions. A typical value of `mips` is 1000 (1 billion instructions/CPU/sec).

  The `mips` value is configured in the `rc` file at startup, using the `conf` directive, for example, `conf mips 1000`. For benchmark runs, the `mips` value is derived from cpustat counters. The `mips` parameter can be changed at any time during simulation.

- `stickfreq` — Determines by how much the STICK register is incremented every `mips` instructions. For example, if the STICK frequency is set at 1.4 GHz (`stickfreq` = 1.4 billion) and `mips` is set at 1000, then STICK is incremented by 1400 every 1000 instructions.

  The STICK frequency is set with the `conf` directive in the `rc` file at startup, for example, `conf stickfreq 1417000000`. The STICK frequency may not be changed after initialization. It should be noted that the STICK frequency configured in SAM should match the STICK frequency specified in the machine description to the guest OS. If they do not match, then the simulated OS may see time progression that is different from the user settings.

So, the smallest unit of simulated time in SAM is 1 microsecond. For example, if `mips` = M and `stickfreq` = N, SAM executes M instructions before incrementing the STICK register by $N/10^6$. This is 1 µsec worth of simulation.

Simulated time can be seen on SAM through the time of the day (`tod`) device model at debug level 2. This should match the `uptime` and `date` command output in simulated Solaris.

Care should be taken to set `mips` and `stickfreq` with reasonable values (close to reference platforms).

10.3 SAM Huron Sim Architecture

Let’s consider an example of configuring a simulation platform called Huron. Don’t confuse it with a production system; there is no a real product with a configuration like that of Huron as described here.

It is important to notice that some I/O modules could physically reside on the processor chip. Huron is just a model consisting of a collection of components configured to form the simulated system.

FIGURE 10-1 is an overview of all major components of the T2 Huron SAM simulation platform. Details for each major module are described later.

The Huron SAM simulation platform is a symmetric multiprocessor system consisting of homogeneous multithreaded SPARC cores tightly coupled with a common memory subsystem. FIGURE 10-1 shows a CPU module attached by the VCPU interface and an I/O root complex attached by the MMI. The user interface (UI), which is an integral part of the main SAM module, allows users to control the simulated system.

The PCIE-to-PCI bridge can be replaced by a PCIE switch, with PCIE-PCI bridges connected to downstream ports. Other PCIE devices (for example, network controllers) can be connected either to the PCIE bus or directly to the downstream switch ports.

SwitchSim models a network and also synchronizes multiple SAM simulation environments. Network interface controller modules (GE and CE) are included in the diagram for completeness. SAM network modeling is not described in this chapter.

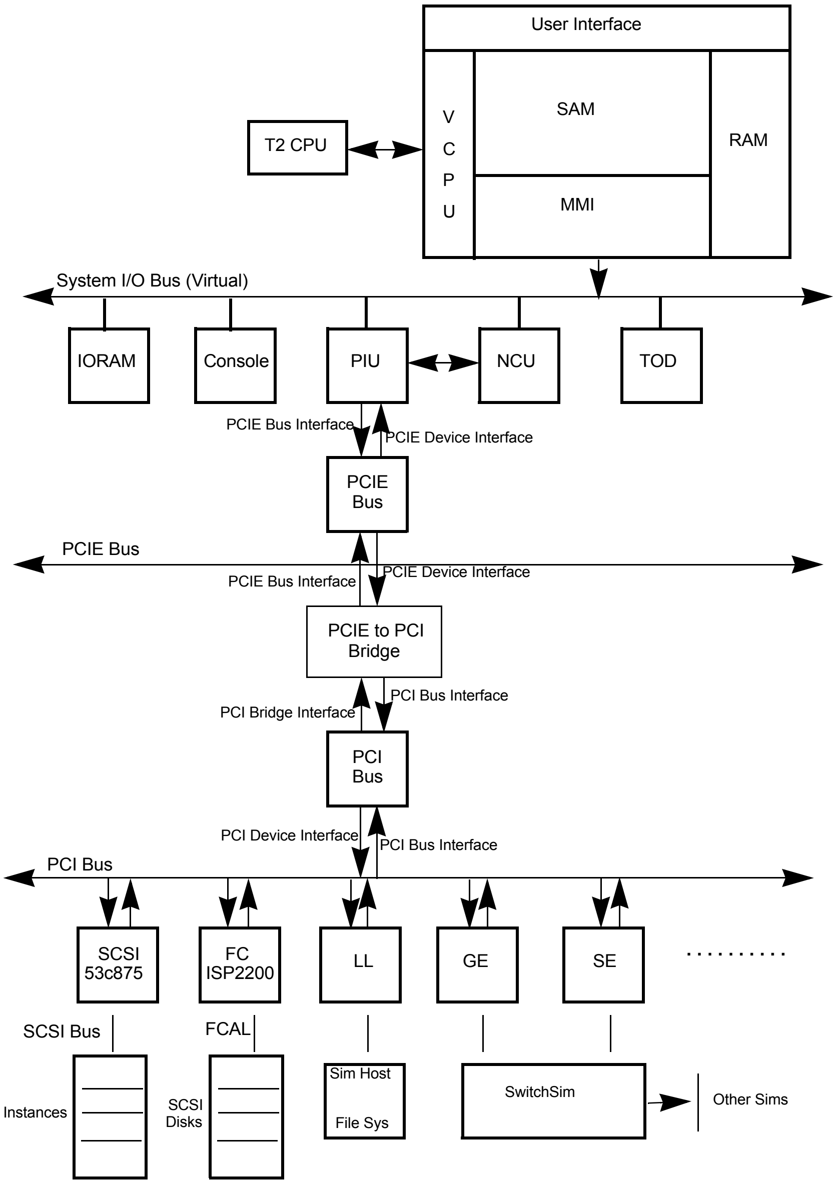


FIGURE 10-1 Overview of Major Components of the Huron SAM Simulation Platform

10.3.1 Sample Configuration File for T2 Huron on SAM

Here is a sample configuration file for T2 Huron system. The file starts with configuration parameters common to the entire system: the size of simulated memory, system performance parameters, files that should be loaded to simulated memory, and the name of the directory where the device module can be found. After that, each device module is configured by a separate sysconf line. Altogether, sysconf lines describe the Huron system device tree.

CODE EXAMPLE 10-2 Sample Configuration File for the T2 Huron System

```
conf ramsize 256M
conf mips 1000

conf stickfreq 1200000000

# number of worker-threads per physical cpu
conf cpu_per_thread 1

# hypervisor partition description, size 8k
load bin hv-md.bin 0x112080000

# machine description , size 8k
load bin guest-md.bin 0x112000000

# nvram for obp, size 8k
load bin ./nvram.bin 0x111000000

sysconf -p ./modules

sysconf cpu name=cpu0 cpu-type=SUNW,UltraSPARC-N2 clock-
  frequency=1200000000  id=0
sysconf cpu name=cpu1 cpu-type=SUNW,UltraSPARC-N2 clock-
  frequency=1200000000  id=1

sysconf ioram reset start_pa=0xfff0000000 size=0x10000 file=reset.bin
  sparse
sysconf ioram hv start_pa=0xfff0010000 size=0x70000 file=q.bin sparse
sysconf ioram obp start_pa=0xfff0080000 size=0x80000
  file=openboot.bin sparse


# hypervisor console device
sysconf serial_4v hypervisor-console base=0xfff0ca0000 size=0x100

# guest console device
sysconf serial_4v guest-console  base=0x9f10000000 size=0x51 pop
  log=guest.log

# tod device
sysconf tod_4v tod base=0xfff0c1fff8 size=0x8 tod=01010000002000 -d2
```

CODE EXAMPLE 10-2 Sample Configuration File for the T2 Huron System *(Continued)*

```
# configure NCU and PIU
sysconf n2_ncu ncu -d2
sysconf n2_piu piu bus=pciea ncu=ncu -d2

# pcie bus connected to piu
sysconf pcie_bus pciea bridge=piu

# pciea bus devices, pcie-pci bridge with two functions
sysconf bridge b0 bus=pciea dev=0 fun=0 secbus=pcia
sysconf bridge b2 bus=pciea dev=0 fun=2 secbus=pcib

# downstream pci buses A and B connected to bridges b0 and b2
sysconf pci_bus pcia bridge=b0
sysconf pci_bus pcib bridge=b2

# pciA devices
sysconf scsi scsi0 bus=pcia dev=1 fun=0 targets=scsidisk1.init
sysconf ll ll0 bus=pcia dev=2 fun=0
sysconf gem ge0 bus=pcia dev=3 fun=0

# pciB devices
sysconf scsi scsi2 bus=pcib dev=1 fun=0 targets=scsidisk.init
sysconf cassini ce1 bus=pcib dev=3 fun=0
sysconf fc fc1 bus=pcib dev=2 fun=0 targets=fcdisk.init
```

10.3.2 Serial Device Module

The Serial Device module (`serial_4v`) is a virtual console device used by OBP. The implementation in based on the `tip` utility. The module provides UI commands and logging capabilities.

Format The `sysconf` format for this module is

`sysconf serial_4v` *instance-name* `base=`*start-addr* `size=`*map* `size [log[=`*filename*`]] [pop] [fg=`*color*`] [bg=`*color*`] [font=`*font*`] [dnkxoe]`

where

- `base` is the starting address of the memory-mapped address of the device CSRs.
- `size` is the size of the CSR space for this device.

- `log` is an optional boolean argument that enables the logging of console input and output. If the argument supplies a *filename* argument, then the console is logged into *filename*. If *filename* is not provided, then the log file is *serial-instance-name*`.log`. The file is opened in append mode so that previous contents are not lost over multiple runs.
- `pop` is an optional boolean argument that, if supplied, causes an xterm window, with the `tip` utility already running in it to pop up on start. The `DISPLAY` environment variable needs to be set up properly for this to work correctly. If `pop` is not present, then the user would need to "tip" into the console from an existing xterm window.

  Note The `sam -w` option for the sun4u system is not supported by this module.

- `fg`, `bg`, `font` are optional arguments to set foreground, background, and text font of the xterm window.

  Notes Users should make sure that `fg`, `bg`, `font` are valid inputs to the xterm program. xterm should be `kill`'ed and `pop`'ed (see UI below) for the new `fg`, `bg`, `font` values to take effect.

  Default: `fg` = `black`, `bg` = `gray90`
  `font` = `-dec-terminal-medium-r-normal-*-14-140-*-75-c-80-iso8859-1`

- `dnkxoe` - optional, do not kill xterm on exit. The default is that the window is killed.

`sysconf` Format Examples

```
# hypervisor console device
sysconf serial_4v hypervisor-console base=0xfff0ca0000
 size=0x1000

# guest console device
sysconf serial_4v guest-console base=0x9f10000000
 size=0x51 pop log=guest.log fg=green bg=black
```

UI Commands The UI format for the Serial Device module is

serial-instance-name command command-args

The module supports the following UI commands:

- `send` *some-char-string* — Echo the character string appended with a new line to the console.
- `sendfile` *filename* — Echo the contents of *filename* to console.

  Note The commands above simulate input from the console; the output echoed on xterm depends on the Solaris/OBP serial driver.

- `debug [`*level*`]` — Set the debug level for `debug` to print *level*. If *level* is not provided, print current debug level.
  level = `[0|1|2]`
- `pop` — Pop up the xterm console if it is not already open.
- `kill` — Kill the xterm console if opened.
- `fg [`*color*`]` — Set the foreground color of xterm to *color*; otherwise, report the current color.
- `bg [`*color*`]` — Set the background color of xterm to *color*; otherwise, report the current color.
- `font [`*font*`]` — Set the font of xterm to *font*; otherwise, report the current font.

UI Format Example

The following example, based on the `sysconf` line in the format example above, sends "`show-devs\`*n*" to the guest console. If the instruction were typed at the OBP ok prompt, this example would show the device tree.

```
run: guest-console send show-devs
```

Mod Info

`modinfo` *serial-instance-name* provides instance-specific information, for example, about the `tip` device:

```
stop: modinfo guest-console
guest-console: sun4v serial module
guest-console: logfile guest.log
guest-console: for console tip /dev/pts/62
guest-console: xterm fg set to green
guest-console: xterm bg set to black
guest-console: xterm font set to -dec-terminal-medium-r-
normal-*-14-140-*-75-c-80-iso8859-1
```

To retrieve the above information, type `help` *serial-instance-name* at the SAM UI prompt. Example:

`help guest-console`

10.3.3 NCU Module

The NCU module (`n2_ncu`) implements the functionality of the T2 noncacheable unit (NCU) on the T2 chip. It supplies the PCI-Express I/O, Mem, and Config mappings, as well as the interrupt to the SAM VCPUs.

Format The `sysconf` format for this module is

`sysconf n2_ncu` *ncu-instance-name* `[-d[0|1|2]]`

where `-d` is the optional argument that sets the debug level to 0, 1, or 2.

`sysconf` Format Example `sysconf n2_ncu ncu -d2`

UI Commands The UI format for the NCU module is

ncu-instance-name command command-args . . .

The NCU module supports the following UI commands:

- `dump [`*filename*`]` — Dump the CSR contents to *filename*. The default is stderr. The `dump` format is
  csr-name csr-offset csr-value
- `restore` *filename* — Restore the CSR contents from *filename*. The `restore` file format is same as the `dump` file format.
- `debug [`*level*`]` — Set the debug level for debug. Print *level*. If *level* is not provided, print current debug level.
  level = `[0|1|2]`

The `dump` command can be used to obtain a snapshot of CSR contents at any time. Debug level 2 shows runtime CPU accesses to this module.

Mod Info `modinfo` *ncu-instance-name* provides information about the physical address assignment of T2 PIU's mapping of the PCIE IO/MEM/CONFIG space.

To retrieve the above information, type `help` *ncu-instance-name* at the SAM UI prompt. Example:

```
help ncu
```

10.3.4 PIU Module

The PIU module (`n2_piu`) implements the functionality of T2 PCI-Express interface unit (PIU) on the T2 chip. The model has been modularized with SAM MMI and enhanced for device interrupts. Dump and restore capability has also been added.

Format The `sysconf` format for this module is

`sysconf n2_piu` *piu-instance-name* `bus=`*pcie-bus-instance-name* `ncu=`*ncu-instance-name* `-d[0|1|2]`

where

- `bus` is the instance name of SAM PCI-Express bus module connected to the downstream port of this instance (see `pcie_bus`).
- `ncu` parameter is the name of T2 NCU model in the SAM `rc` file (see `n2_ncu`).
- `-d` is the optional argument that sets the debug level to 0, 1, or 2.

`sysconf` Format Example `sysconf n2_piu piu bus=pciea ncu=ncu`

UI Commands The UI format for the PIU module is

piu-instance-name command command-args

The module supports the following UI commands:

- `dump [`*filename*`]` — Dump the CSR contents to *filename.* Default is stderr.
- `restore` *filename* — Restore the CSR contents from *filename*. The `restore` file format is same as the `dump` file format.
- `debug [`*level*`]` — Set the debug level for `debug` to print *level*. If *level is* not provided, print current debug level.
  level = `[0|1|2]`

The `dump` command can be used to obtain a snapshot of CSR contents at any time. The debug level 2 can be set to view on-the-fly read/write accesses to CSRs, virtual-to-physical address translations, device interrupts, and so forth.

Debug Examples The following examples are based on the `sysconf` line above.

This example sets the debug message flag to 2:

```
piu debug 2
```

This example takes a CSR content snapshot:
```
piu dump
```

Mod Info `modinfo` *instance-name* provides additional instance-specific information such as physical address mapping of this instance into T2 I/O address space.

To retrieve the above information, type `help` *piu-instance-name* at the SAM UI prompt. Example:
`help piu`

10.3.5 IORAM Module

The IORAM module (`ioram`) implements RAM/ROM in the I/O address space of the processor (for example, the boot ROM can be implemented with this module). In the Huron setup, this module is used to load `reset.bin`, `q.bin` (Hypervisor), and `openboot.bin` (T2 OBP) as instances of the `ioram` device.

Format •The `sysconf` format for this module is

```
   sysconf ioram ioram-instance-name start_pa=addr size=size
[file=name [addr=load-addr]] [rw|ro|wo] [sparse|flat]
[-d[0|1|2]]
```

where

- `start_pa` is the start address of the memory segment in IO space of processor.
- `size` is the size of the memory segment.
- `file` is an optional argument that, if provided, loads the memory segment with the contents of that file. If this argument is not provided, then the segment is zero filled.

  The file formats that are supported are `.bin`, `.img`, and `.elf`. The file name must have these extensions in order for the type to recognized.
- `addr` is the intended load address in the case of `.bin` and `.elf` files.

  `.img` files should not supply this argument.
- `rw` is an optional boolean argument that makes this segment read/write.

  Only one of `rw`, `ro`, `wo` must be supplied. Default is `rw`.
- `ro` is an optional boolean argument that makes this segment read-only.
- `wo` is an optional boolean argument that makes this segment write-only.
- `sparse` is an optional boolean argument that makes this segment type sparse.

- `flat` is an optional boolean argument that makes this segment type flat. Only one of the two arguments above must be supplied.

> **Note** Currently only the sparse model is supported. Default is also `sparse`.

- `-d` is the optional argument that sets the debug level to 0, 1, or 2.

`sysconf` Format Examples Two examples of the IORAM module format are shown below.

```
sysconf ioram obp start_pa=0xfff0080000 size=0x80000
  file=openboot.bin sparse

sysconf ioram hv start_pa=0xfff0010000 size=0x70000 file=q.bin
```

UI Commands The UI format for the IORAM module is

ioram-instance-name command command-args . . .

The module supports following UI commands:

- `write` *addr value size* — Write *size* bytes to memory at address *addr* where *size* = `[1|2|4|8]`
- `read` *addr size* — Read *size* bytes from memory at address *addr* where *size* = `[1|2|4|8]`
- `dis` *addr* [*n-instr*] — Disassemble *n-instr* instructions (default 1) starting from address *addr.*
- `dump` *addr size* [*format*] — Dump *size* bytes of memory on stdout, where *format* is `[x|d|c|o]`. The default is `x`.
- `save` *filename* [*addr* [*size*]] — Save the memory contents in *filename* starting from *addr* for *size* bytes. If no *addr* and *size* arguments are specified, save all. If *addr* is supplied but *size* is not, save until the end.

All numerical values are expected to be in decimal, octal, or hex. The *addr* argument is an absolute physical address.

`dis` Example The following example, based on `sysconf` line above, prints the disassembly of 256 instructions from `openboot.bin`, starting from physical address FF F008 0000_{16}.

```
obp dis 0xfff0080000 0x100
```

Mod Info The `modinfo` *instance-name* provides additional instance-specific information such as physical address mapping of this instance into the SAM I/O address space, load file name, and so forth.

For example, `modinfo obp` prints the following information:

```
obp: SAM IOMem module
start paddr <0xfff0080000, end paddr <0xfff00fffff>
loaded with file <openboot.bin> at addr <fff0080000>
s/w access <rw>, implementation <sparse model>
for UI help type obp
```

To retrieve the above information, type `help` *ioram-instance-name* at the SAM UI prompt. Example:
`help hv`

10.3.6 Time-of-Day Module

The Time-of-Day module (`tod_4v`) implements the virtual time-of-day device. The module allows optional setting of the time of day. It also maintains correct time of day according to SAM's global time. This means that in a correctly configured setup, no Solaris error messages about the time of day being stalled or the time-of-day clock jumping ahead should occur.

At debug level 2, this module prints "simulated" system time once every simulated second (this may give an indication if the simulated system seems sluggish—the simulated time might be moving too slowly because of configuration parameters).

Format The `sysconf` format for this module is

`sysconf tod_4v` *tod-instance-name* `base_pa=`*start-addr* `size=`*map-size* `[tod=`*mmddHHMMSSyyyy*`] [-d[0|1|2]]`

where

- `base_pa` is the start address of the memory-mapped address of the device CSRs.
- `size` is the size of the CSR space.
- `tod` is the time of day initial value. If not present, the current host system time is read. TOD format string semantics are
  - *mm* – month
  - *dd* – day
  - *HHMMSS* – hours, minutes, seconds
  - *yyyy* – year, should not be before 1970
- `-d` is the optional argument that sets the debug level to 0, 1, or 2.

`sysconf` Format Example The following example sets the time to 12 A.M. Jan 1, 2000.

```
sysconf tod_4v tod base=0xfff0c1fff8 size=0x8
tod=01010000002000 -d2
```

UI Commands The UI format for the Time-of-Day module is

tod-instance-name command command-args . . .

The module supports following UI command:

- `debug [`*level*`]` — Set the debug level for `debug` to print *level*. Print current `tod` every simulated second, and print total simulated time. If *level* is not provided, print current debug level.
  level = `[0|1|2]`

  Note Solaris sim time and tod sim time match only if the Solaris STICK frequency is same as the SAM STICK frequency.

Mod Info `modinfo` *tod-instance-name* provides instance specific information such as current time of day.

To retrieve the above information, type `help` *tod-instance-name* at the SAM UI prompt. Example:
`help tod`

10.3.7 PCI-E Bus Module

The PCI-E Bus module (`pcie_bus`) implements the abstraction of a PCIE bus. It routes the upstream/downstream accesses between the bridge and downstream devices. It maintains the I/O, Mem, and Config space maps of the downstream devices. It implements SAM's PCIE bus interface and routes the accesses by using SAM's PCIE device interface. This is a generic implementation of the PCIE bridge device.

Format The `sysconf` format for this module is

`sysconf pcie_bus` *pcie-bus-instance-name* `bridge=`*upstream-bridge-name* `[-d[0|1|2]]`

where

- `bridge` is the name of upstream bridge device. It could be `n2_piu`, `hammerhead`, `fire`, and so on.
- `-d` is the optional argument that sets the debug level to 0, 1, or 2.

`sysconf` Format Example `sysconf pcie_bus pciea bridge=piu`

UI Commands This module currently does not support a UI. The debug level can be set only through the `sysconf` line. The debug level 2 shows on-the-fly transactions going through this bus instance.

Mod Info `modinfo` *pcie-bus-instance-name* provides the PCIE I/O, Mem, and Config space mapping information for the downstream devices connected to this bus.

Output Example

```
stop: modinfo pciea
   bridge=<piu>
PCIE config::devices:
name <b2>, dev <0>, fun <2> base <2000> end <2fff>
name <b0>, dev <0>, fun <0> base <0> end <fff>
PCIE IO::devices:
name <b2>, dev <0>, fun <2> base <1000> end <1fff>
name <b0>, dev <0>, fun <0> base <0> end <fff>
PCIE MEM::devices:
name <b2>, dev <0>, fun <2> base <400000> end <7fffff>
name <b0>, dev <0>, fun <0> base <100000> end <3fffff>
```

10.3.8 PCIE-PCI Bridge Module

The PCIE-PCI Bridge module (`bridge`) is a functional model for the Intel 41210 PCIE-PCI bridge.

Format The `sysconf` format for this module is

`sysconf bridge` *bridge-instance-name* `bus=`*primary-pcie-bus* `dev=`*device* `fun=`*function* `secbus=`*secondary-pci-bus*

where

- *primary-pcie-bus* is the instance name of the upstream PCIE bus module.
- *device* is the device number of `bridge` on upstream PCIE bus.
- *function* is the function number of `bridge` on upstream PCIE bus.
- *secondary-pci-bus* is the instance name of the downstream PCI bus module.

`sysconf` Format Example The following three lines configure an upstream PCIE bus named `pciea`, a downstream PCI bus named `pcia`, and a PCIE-PCI bridge named `b0`, respectively.

```
sysconf pcie_bus pciea bridge=piu
sysconf bridge b0 bus=pciea dev=0 fun=0 secbus=pcia
sysconf pci_bus pcia bridge=b0
```

UI Commands The UI format for PCIE-PCI Bridge module is

bridge-instance-name command command-args . . .

The PCIE-PCI bridge, `b0`, supports the following UI commands:

- `debug [`*level*`]` — Set the debug level for `debug` to print *level*. If *level* is not provided, print current debug level.
  level = `[0|1|2]`.
- `dump [`*filename*`]` — Dump the PCIE CSR contents to *filename*`.pcie` in the current working directory; default is stderr. The `dump` format is
  csr-pciconf-offset csr-value csr-rw-mask csr-size csr-name
- `restore` *filename* — Restore the PCIE CSR contents from *filename*`.pci` in cwd. The `restore` file format is the same as the `dump` file format.

The debug level 2 shows CSR access and change in values in case of writes. The `dump` command can be used to get a snapshot of CSR contents at any time.

`dump` Example The example is based on the `sysconf` line above.

```
stop: b0 dump
       0x000000000x000080860x00000000 0x2 b0-pcieaVendor ID
       0x000000020x000003400x00000000 0x2 b0-pcieaDevice ID
       0x000000060x000000100x0000f900 0x2 b0-pcieaPrimary device
       status reg
       0x000000080x000000000x00000000  0x1 b0-pciea revision id
       0x000000090x000000000x00000000 0x1 b0-pcieaprog interface
<----snipped---->
```

Mod Info `modinfo` *bridge-instance-name* shows module specific information.

To retrieve the above information, type `help` *bridge-instance-name* at the SAM UI prompt. Example:
`help b0`

10.3.9 PCIE-PCIE Bridge Module

The PCIE-PCIE Bridge module (`pcie_bridge`) is a functional model for PCIE-PCIE bridge component of PLX 8532. The switch is composed by configuration of individual PCIE bridges connected by an internal PCIE bus in the `sysconf` file. FIGURE 10-2 illustrates a sample configuration.

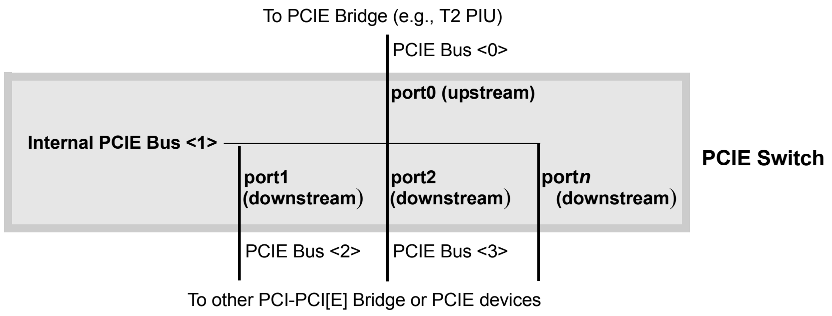


FIGURE 10-2 Sample Configuration of PCIE-PCIE Bridge Module

The module shown in FIGURE 10-2 can replace the PCIE-PCI bridge in the Huron system diagram.

Format The `sysconf` format for an individual PCIE bridge is

`sysconf pcie_bridge` *instance-name* `bus=`*upstream-bus* `secbus=`*downstream-bus* `dev=`*device-number* `[upstream]`

where

- `bus` is the upstream bus for this bridge instance.
- `secbus` is the secondary bus of this bridge instance.
- `dev` is the device number on the upstream bus.
- `upstream` is a boolean parameter that designates the bridge as a switch upstream port. If absent, the bridge is a downstream port.

`sysconf` Examples In the following examples of switch configuration, port 0 is the upstream port, and ports 1, 2, 8, 9 are the downstream ports. Note that the port is the same as `dev` here. Users can also add ports 3, 10, and 11.

Upstream port—`pcie_a` is the primary bus of upstream bridge:

```
sysconf pcie_bridge b0 bus=pcie_a dev=0 fun=0 secbus=pcie_int
upstream
```

Internal virtual PCIE bus:

```
sysconf pcie_bus pcie_int bridge=b0
```

Downstream ports connected at upstream to `pcie_int`:

```
sysconf pcie_bridge b1 bus=pcie_int dev=1 secbus=pcie_b
sysconf pcie_bridge b2 bus=pcie_int dev=2 secbus=pcie_c
sysconf pcie_bridge b3 bus=pcie_int dev=8 secbus=pcie_d
sysconf pcie_bridge b4 bus=pcie_int dev=9 secbus=pcie_e
```

Downstream PCIE buses:

```
sysconf pcie_bus pcie_b bridge=b1 sysconf pcie_bus pcie_c bridge=b2
sysconf pcie_bus pcie_d bridge=b3 sysconf pcie_bus pcie_e bridge=b4
```

UI Commands The UI format of the PCIE-PCIE Bridge module is

bridge-instance-name command command-args . . .

The module supports following UI commands:

- `debug` [*level*] — Set the debug level for `debug` to print *level*. If *level is* not provided, print current debug level.
  level = `[0|1|2]`
- `dump` [*filename*] — Dump the PCIE CSR contents to *filename*`.pcie` in cwd. Default is stderr. The `dump` format is
  csr-pciconf-offset csr-value csr-rw-mask csr-size csr-name
- `restore` *filename* — Restore the PCIE CSR contents from *filename*`.pci` in cwd. The `restore` file format is the same as the `dump` file format.

The debug level 2 shows CSR access and change in values in the case of writes. The `dump` command can be used to get a snapshot of CSR contents at any time.

`dump` Example The example, based on `sysconf` line above, prints the PCIE register value on stderr.

```
stop: b0 dump
```

Mod Info `modinfo` *bridge-instance-name* shows module-specific information.

To retrieve the above information, type `help` *bridge-instance-name* at the SAM UI prompt. Example:
`help b0`

10.3.10 Serially Attached SCSI Module

The Serial Attached SCSI module (`sas`) implements the functionality of the Serial Attached SCSI (SAS) disk controller. It models LSI SAS1064E, the design of which is based on the Fusion-MPT (Message Passing Technology) architecture. The module has a PCIE interface through which the controller can be connected to any PCIE bus. Each controller can support up to four SAS disks (target 0~3). This implementation does not include a timing model, so the I/O request is served and completed immediately after it is received by the controller, assuming no simulated disk delay. The SAS disk is implemented according to a generic disk model.

Format The `sysconf` format for this module is

`sysconf sas` *instance-name* `bus=`*bus-name* `dev=`*device-id* `fun=`*function-number* `targets=`*init-file* `[-d[0|1|2]]`

where

- *instance-name* is the name of this controller.
- *bus-name* is the name of PCIE bus to which this controller is to be connected.
- *device-number* is the PCIE device number.
- *function-number* is the PCIE function number.
- *init-file* is the configuration file for the attached SAS disks.
- `-d` is the optional argument that sets the debug level to 0, 1, or 2.

`sysconf` Format Example

A SAS controller called `sas0` is attached to the PCIE bus `pcie_a` as device 0, function 1. The configuration for the attached disks is in file `sasdisk.init`. The debug level is set to 2.

```
sysconf sas sas0 bus=pcie_a dev=0 fun=1 targets=sasdisk.init
-d2
```

UI Commands—This Module

The UI format for this model is
instance-name command command-args

UI commands for this module are as follows:

- `debug [`*level*`]` — Set the debug level for `debug` to print *level*. If *level* is not provided, print current debug level.

  level = `[0|1|2]`
- `disk` — Show all attached disks.

To retrieve the above information, type *instance-name* at the SAM UI prompt. Example:
`sas0`

UI Commands—Attached Disks

The UI format for attached disks is
`gdisk` *disk-name command command-args*

UI commands for attached disks are as follows:

- `label` — Display disk label.
- `geometry` — Display disk geometry.
- `stat` — Display any supported stats.
- `partitions` [*number*] — Display partition[*number*] information. Display all partitions by default.
- `vpd` — Display supported disk VPD data.
- `op` [*file*] — Redirect all o/p to *file*. By default, print current `op` file.
- `debug` [*level*] — Set `debug` verbosity to *level*.

To retrieve the above information, type **`gdisk help`** at the SAM UI prompt. Typing **`gdisk`** will list all attached disks in the system.

Configuration file for attached disks. Each controller requires a configuration file for the attached disks. In this file, each line specifies a disk partition that is attached to the controller. The format of this specification is described below.

```
tTdDsS filename [vtoc|ro|rw]     [vendor-id="vendor id"
                                  product-id="product id" |
                                  revision-id="revision id" |
                                  serial-no="serial no" |
                                  prodserial-no="product serial no" |
                                  brdserial-no="brd serial no" |
                                  port-wwn="port wwn" |
                                  node-wwn="node wwn" |
                                  bytes/sector=number of bytes per sector |
                                  sectors/track=number of sectors per track |
                                  tracks/cylinder=number of tracks per
                                                  cylinder |
                                  debug-file="debug file" |
                                  debug-level=debug level [0|1|2]]
```

where

- In *tTdDsS*, *T* is the target ID, *D* is the disk ID, and *S* is the partition ID. Only one disk is currently allowed for each target, so *D* should be always set to 0.
- *filename* is the name of disk image file.
- `vtoc` is the disk geometry included in disk image file.
- `ro` specifies that the disk image file is read-only.
- `rw` specifies that the disk image file is writable.

Additional information related to a disk, such as vendor ID, product ID, and revision ID, can be specified at the end of each line.

10.3.11 LLFS Module

The LL device allows host file system to be accessed from the simulated system running Solaris.

The LL device is configured into the system with the following `sysconf` line:

`sysconf ll` *ll-instance-name* `bus=`*upstream-pci-bus name* `dev=`*pci-device-number*

For example,

```
sysconf ll ll0 bus=pcia dev=2
```

10.4 Creation of a Root Disk Image File

A simulated platform can boot Solaris from the simulated disk. The easiest way to accomplish this is to create a root disk image from the reference system.

To create a root disk image:

1. Log in as root and run `prtvtoc` on the disk where the root partition resides to get the disk configuration.

   To locate the root partition, look at `/etc/vfstab` and find the device logical link that is mounted at /. For example, if the root partition is c0t1d0s0, execute `prtvtoc` on slice 2 of the disk, that is, `c0t1d0s2`. Here is an example.

```
#prtvtoc /dev/rdsk/c0t1d0s2
   * /dev/rdsk/c0t1d0s2 partition map
   *
   * Dimensions:
   *      512 bytes/sector
   *      248 sectors/track
   *      19  tracks/cylinder
   *      4712sectors/cylinder
   *      7508cylinders
   *      7506accessible cylinders
   *
   * Flags
   *      1: unmountable
   *      10: read-only
   *
   *                                First    Sector     Last    Mount
   * Partition  Tag  Flags     Sector     Count    Sector    Dir
         0        2   00          0      31273544 31273543    /
         1         3   01     31273544    4094728 35368271
         2         5   00          0      35368272 35368271
```

2. The preceding vtoc table shows that the root partition starts from 0 (first sector) and totals 31273544 sector counts. Slice 2 of a disk is a special slice that points to the entire disk, so when you execute `dd`, your input is slice 2 of the disk that holds the root partition.

   For example, use the following command to create a root disk image:

   ```
   dd if=/dev/dsk/c0t1d0s2 of=/your/work/dir/env09-root iseek=0
     count=31273544
   ```

   In general, the format is as follows:

   - `dd if` = partition 2 of the disk within which root resides; partition 2 points to the entire disk of disk image file name
   - `iseek` = first sector
   - `count` =sector count

3. Follow steps 1 and 2 for the swap partition to generate swap disk image file.

The root image can be modified to load additional drivers, for example, LL driver. This driver gives access to the local-host file system from the SAM simulated environment. SAM supports LLFS (local-host lookup file system), letting users read and write files on their host machine directly from the simulated host. Thus, the simulated machine can access */your-home-dir* or `/tmp/logman` on the host. More importantly, LLFS lets users run SAM as a user program and gain access to all NFS files, where test benchmark programs probably reside.

By mapping root directory on the host to `/ll/root` in the simulated host through LLFS, users can access their home directory from `/home/`*xyz* by going to `/ll/root/`*home/xyz* in SAM.

For convenience, symbolic links can be used to map the following directories in the simulated host to the same directory on the underlying host machine: `/home`, `/net`, `/vol`, `/ws`, `/import`, `/usr/ccs`, `/usr/share`, `/usr/shared`, `/usr/dist`. Thus, in the simulated machine, users can access their home directory via `/home/`*xyz* as well as via `/ll/root/home/`*xyz*.

10.5 Debugging With SAM

It is always possible to run a debugger (`kmdb` for kernel debugging or `dbx` for application debugging) on the simulated system. But doing that requires the system to get through reset and the initial phases of the boot-up process. In the early stages of development, a significant effort is required just to reach the point at which the simulated system could load a kernel debugger.

Another option is to run a remote debugger—`gdb` has an option to run as a remote debugger and to be connected to the target, the SAM simulator in this case.

SAM has a special module that can be loaded to establish communication with the remote `gdb` debugger. SAM configuration files need to load a special module, for example,

```
sysconf remote port=6450
```

loads the interface module and waits for the connection from `gdb`; in this example, it waits at the tcp connection on port 6450.

Now when `gdb` starts, to connect to SAM, the `gdb` command is executed:

`target remote` *host-name*`:`*port*

where

- *host-name* is the name of the host on which SAM is running.
- *port* is the port ID (6450 in this example), which needs to be the same as in the sysconf command line.

Although the remote debugger could load symbol information for source-level debugging, overall, the remote debugger approach is generally too restrictive to debug multiprocessor/multithreaded environments.

Debugging consumes most of a developer's energy and time, but it is one of the least discussed tasks for software development, system bringup, and integration projects. Usually, the basic steps for troubleshooting remain much the same:

- Try to make the bug repeatable.
- Isolate the problem.
- Make corrections.
- Test whether the corrections fixed the problem.

Certain classes of bugs are difficult to make repeatable, so some special strategies are required. Isolating the problem involves narrowing the range of possibilities until the bug can be correlated to a specific segment of code or data. There are a few general approaches to this problem.

One approach to localizing the bug is to single-step through the suspect code, trying to identify an abnormal behavior. The main problem with this approach is that it tends to be quite tedious. For large programs containing many loops, complex data structures, and complicated interaction between modules and threads, it is difficult to locate the code segment where the bug could be observed.

Another way to find a bug is to construct a hypothesis to explain how the software could reach such a state, then modify the experiment to test the hypothesis. This approach demands problem-solving skills very different from those required to design the code in the first place. A trace history of executed code if available could be most useful in localizing the bug.

The SAM user interface has convenient built-in features for debugging software during system bringup and does not require the use of another remote debugger. The SAM user interface can start and stop simulation at any time, advance any virtual processor by some number of instructions, and examine architecture state for each virtual processor, memory state, and state of simulated devices.

It is important to note that this discussion is about machine-level debugging, not program debugging that most software engineers are accustomed to. The debugger is not running on a simulated system; rather, the debugger is running on the host machine that runs simulated system. Normally, the debugger would get significant support from the compiler, loader, and operating system to map symbol information for the program being debugged. That luxury is denied us owing to the disconnect between a debugger running on the host machine and a simulated system that runs the program we are interested in. Overall, the process is similar to cross-debugging, that is, debugging code on a separate target system.

10.5.1 Simulated State Access

The `pselect` command selects a default virtual CPU ID to which other user interface commands will be referred. Many commands that have a VPCU ID as an argument operate on the default `vcpu` when the argument is omitted.

CPU registers can be read in groups or individually:

```
stop: pregs
cpu[0]:
pr  0              tpc = 0xf02398a4
pr  1             tnpc = 0xf02398a8
pr  2           tstate = 0xa520001600
pr  3               tt = 0x180
pr  4          pr_tick = 0x2a8052950
pr  5              tba = 0xf0200000
pr  6           pstate = 0x10
pr  7               tl = 0x1
pr  8              pil = 0xd
pr  9              cwp = 0x0
pr 10          cansave = 0x6
pr 11       canrestore = 0x0
pr 12         cleanwin = 0x7
pr 13         otherwin = 0x0
pr 14           wstate = 0x0
pr 16               gl = 0x1
asr 0                y = 0x0
asr 2              ccr = 0x99
asr 3              asi = 0x20
asr 4             tick = 0x2a8052950
asr 5               pc = 0x42e224
asr 6             fprs = 0x4
asr16              pcr = 0x0
asr17              pic = 0x0
asr19              gsr = 0x0
asr20      softint_set = 0x0
asr21      softint_clr = 0x0
asr22          softint = 0x0
asr23        tick_cmpr = 0x8000000000000000
asr24            stick = 0x2a8052950
asr25       stick_cmpr = 0x2a8d90dd8
hpr 0          hpstate = 0x4
hpr 1          htstate = 0x0
hpr 3            hintp = 0x0
hpr 5             htba = 0x400000
hpr 6             hver = 0x3e002420030607
hpr31      hstick_cmpr = 0x2aac737d7
stop:
      stop: %g1
0x28
```

It is possible to change register state:

```
stop: write-reg g1 0x1
stop: read-reg g1 cpu[0]: g1=0x1
```

Memory can be read in binary format or it can be disassembled, for example:

```
stop: mem -a 0x42e224 -s 0x10 -dis
disassemble saddr=0x42e220 eaddr=0x42e230
0 : 0x42e220: 0x82102028 : or   %g0, 0x28, %g1
0 : 0x42e224: 0xc2d849e0 : ldxa [%g1 + %g0]0x4f, %g1
0 : 0x42e228: 0xc2586000 : ldx  [%g1 + 0], %g1
0 : 0x42e22c: 0xca086980 : ldub [%g1 + 0x980], %g5
0 : 0x42e230: 0x80a16000 : subcc        %g5, 0, %g0
0 : 0x42e234: 0x267b8f1  : bpe,pn 1%xcc, 0x41c5f8
0 : 0x42e238: 0x1000000  : nop
0 : 0x42e23c: 0x80a16001 : subcc        %g5, 1, %g0
stop:
```

Memory state can changed with the `set` command:

```
stop: set 0x10000 0x1
stop: get 0x10000
0x0000000000010000: 0x00000001
```

There also commands to examine registers mapped to non-translating ASIs and TLB state for each virtual processor.

Each device module has a specific set of commands with which to access state for that module.

10.5.2 Symbol Information

SAM can process symbol information from the modules that run on the simulated machine. For reallocatable modules, starting text/data addresses are required; for executable modules, addresses for text and data could be extracted from the ELF file.

```
load_symbols -elffile -f unix                                  -ctx k
load_symbols -elffile -f krtld    -x 0x10a6878 -d 0x18929a0 -ctx k
load_symbols -elffile -f genunix   -x 0x10c1738 -d 0x1898940 -ctx k
```

The `load_symbols` SAM user-interface command loads symbol information for the ELF file, and it lets users specify the base address for text and data sections and the context in which the module will be accessed. Symbol information is kept on the host machine; it does not load these modules to memory—the simulated system will load them.

SAM can look up a symbol by name or address:

```
stop: sym -s main

genunix:main at address=0x11471d4, size=0x3fc
stop:
stop: sym -a 0x11471d4
genunix:main+0x0
at address = 0x11471d4
stop:
```

The kernel module unix in the preceding example cannot be re-allocated—the `load_symbols` command can extract the loading address directly from the ELF file. The same is true for the hypervisor module. Having symbol information for unix and hypervisor modules is quite helpful in the early stages of the bringup process. Module `genunix` is a relocatable ELF file; a starting address for text and data sections (`-x` and `-d` option, respectively) must be provided. Finding out loading addresses for the module requires extra effort. When the `kmdb` debugger runs on the simulated machine, it gets base address information from the kernel loader `krtld`; this information may not be accessible from the host machine.

Starting addresses for kernel modules usually can be found on the console log printout generated by the kernel loader when the `moddebug` option in the `/etc/system` file on the simulated system (root image) is enabled: `moddebug 0x80000000`. To enable logging load addresses for the first five modules (`unix`, `krtld`, `genunix`, `platmod`, and `SUNW`), specify `boot -vV`. Those five modules are always loaded by `usfboot` in the same order. The `moddebug` flag controls log information generated by `krtld` to load other modules after `krtld` itself was loaded. CODE EXAMPLE 10-3 presents an example.

CODE EXAMPLE 10-3 Starting Addresses for Kernel Modules

```
ok boot /pci@0/pci@0/pci@2/scsi@0/disk@0,0:f -vV -m verbose

Boot device: /pci@0/pci@0/pci@2/scsi@0/disk@0,0:f  File and args: -vV
  -m verbose

device path '/pci@0/pci@0/pci@2/scsi@0/disk@0,0:f'

The boot filesystem is logging. The ufs log is empty and will not be
  used.

standalone = 'kernel/sparcv9/unix', args = '-vm verbose'

Elf64 client

Size: 0xa6877+0x31c88+0x60d14 Bytes

modpath: /platform/SUNW,SPARC-Enterprise-T5120/kernel
  /platform/sun4v/kernel /kernel /usr/kernel

module /platform/SUNW,SPARC-Enterprise-T5120/kernel/sparcv9/unix:
  text at [0x1000000, 0x10a6877] data at 0x1800000
```

CODE EXAMPLE 10-3 Starting Addresses for Kernel Modules *(Continued)*

```
module misc/sparcv9/krtld: text at [0x10a6878, 0x10c1737] data at
  0x18929a0

module /platform/SUNW,SPARC-Enterprise-
  T5120/kernel/sparcv9/genunix: text at [0x10c1738, 0x129f7a7] data
  at 0x1898940

module /platform/SUNW,SPARC-Enterprise-
  T5120/kernel/misc/sparcv9/platmod: text at [0x129f7a8, 0x129f7bf]
  data at 0x18ecdf0

module /platform/SUNW,SPARC-Enterprise-
  T5120/kernel/cpu/sparcv9/SUNW,UltraSPARC-T2: text at [0x129f7c0,
  0x12a44ff] data at 0x18ed540
```

Finding base addresses for processes in the user context requires more work. A valid context ID must also be provided. `LD_AUDIT` and a special module that intercepts addresses at which user modules are loaded should be used to find out base address and context ID.

In some cases, symbol information is needed for the Java programs running on the Java virtual machine (JVM), which in turn runs on simulated system. In this case, the Java virtual machine should be augmented with a special agent that outputs addresses for routines generated by JVM.

10.5.3 Breakpoints

SAM lets users set a few types of breakpoints for a particular virtual processor: set a breakpoint on instruction virtual address, trap type, or RED mode.

Here is an example:

```
To set breakpoint 0 for cpu 0
stop: break 0x11471d4
stop: run
..
cpu[0] hit a breakpoint. stop.
stop:
stop: where
cpu[0]: pc=0x11471d4 genunix:main+0x0
called from address 0x10aa358 krtld:kobj_init+0x290
```

To remove the breakpoint:

```
stop: delete 0
```

10.5.4 Debug Tracing

A special built-in tracer, `vdebug`, is a convenient tool for finding the time window during which some particular problem occurs.

When symbol information is loaded, SAM will annotate a debug trace with routine names and for data accesses with structure names if a symbol match is found.

The trace format is

cpuid: *module*:*routine* *va*(*context*) : *pa* [opcode] *mode* *instruction*
cpuid: *reg*=*value*

Here is an example.

```
stop: vdebug -cpu 0 on
debug tracer is turned on for cpu 0
stop: stepi 4

0: unix:lgrp_root_init+0xc 1062b48(0) : 1003862b48 [f40763f4]
   p lduw [%i5 + 0x3f4], %i2
0:      i2=0x0
0:      load  0(ffffffff) from 18607f4(0):10034607f4
           unix:nlgrps+0x0
0: unix:lgrp_root_init+0x10 1062b4c(0) : 1003862b4c [b806a001]
   p add %i2, 1, %i4
0:      i4=0x1
0: unix:lgrp_root_init+0x14 1062b50(0) : 1003862b50 [f82763f4] p stw
   %i4, [%i5 + 0x3f4]
0:     store 1(ffffffff) to 18607f4(0):10034607f4 unix:nlgrps+0x0
0: unix:lgrp_root_init+0x18 1062b54(0) : 1003862b54 [3b006154]
   p sethi %hi(0x1855000), %i5
0:      i5=0x1855000
```

The debug tracer outputs value-change information for every executed instruction. Tracing produces a huge amount of data, and debugging is tedious process—no one solution can be used to approach all problems.

For further narrowing of the window to focus only on essential information during the debug session, it is convenient to use probes.

10.5.5 Probes

A probe monitors some condition during the simulation. It separates the notion of condition and action. Probe implementation internally is based on breakpoints to temporarily pause the simulation and execute probe commands.

The following example observes cross-calls for virtual processors 0 through 15:

```
probe -cpu 0..15 -trap 0x7c -exec "where"
```

The `where` command prints the call stack for each virtual processor every time the CPU mondo trap occurs. Probe commands can enable/disable debug tracing, a feature that helps users navigate through the massive amount of information accumulated during a debug session. Using this feature is similar to using a logic analyzer that is connected to some probes on the system board and could collect trace fragments to the buffer.

10.6 Cycle-Accurate Simulation

While an instruction-level model is useful for system software development, verification, system bringup, tracing, and the like, for performance analysis a cycle-accurate model is needed. Behavioral cycle-accurate models are used not only for CPU microarchitecture but also for overall system-performance exploration. A few approaches can be used to accomplish cycle-accurate simulation; each approach has its pros and cons.

10.6.1 Trace-Driven Approach

CPU architects are accustomed to running trace-driven timing models to make microarchitecture trade-offs on a particular set of benchmarks. A timing model is essentially a behavioral model that implements the notion of cycles, pipelines, caches, store/write/merge buffers, speculative/scout execution, and so on. The model does not capture all the microarchitecture details, but captures only the most significant ones. For example, the timing model is commonly used when only cache tags are modeled to get cache hit/miss behavior but cache line data is not included.

The trace-driven approach assumes that instruction traces should be collected somehow, usually by means of instruction-level models. Traces take massive amount of storage space, so some strategy is needed to figure out significant trace fragments that represent the benchmark behavior. At the same time, traces are collected and verified as to whether they could be reused to analyze microarchitecture trade-offs. Instruction trace contains only instructions that were committed, but the timing model also has to deal with speculatively executed instructions to get the correct timing behavior. That is where the postprocessing nature of the trace-based approach becomes a problem—the accuracy that a trace-driven timing model can achieve is limited.

10.6.2 Execution-Driven Approach

For improvement to the timing accuracy of the model, an execution-driven approach is more often used. The timing model in an execution-driven approach is advanced cycle by cycle and must keep architecture-committed and speculative states. There is a reverse dependency between accuracy (how many features are reflected in the model) and model performance. This approach could potentially be as accurate as the RTL model, but in that case it would be as slow as the RTL model. Microarchitecture developers try to capture only some essential behavior in the timing model that contributes the most to the accuracy to keep model performance at the level acceptable for running real benchmarks.

An execution-driven, cycle-accurate model can be implemented in a few ways. One approach would be to build it from the ground-up, keeping a notion of cycles, pipeline, and architectural state (committed and speculative) inside one model. With modern microarchitectures (decoupled fetch and execution pipelines, out-of-order instruction issue and completion, and the like), it is hard for a generic timing model to be configurable by some kind of script. For modeling control logic in one way or another, the modules need to be written in a high-level programming language; C or C++ is the most common choice. The main benefit with this approach is that it provides realistic cycle simulation for the system interface. For system performance modeling, this approach provides the best results.

The problem with the execution-driven approach is that making changes in the microarchitecture becomes a difficult task; changes can affect not only the timing but also the correctness of basic instruction execution. Verification for every update in the model became tedious; in addition to timing verification, functional verification should be done. The CPU cycle behavioral model is desirable for system performance modeling, but it should be done after the CPU microarchitecture is stable.

10.6.3 Submodule Approach

Sometimes it is easier to split a timing model into a few modules: one submodule to model a committed state, another submodule to model only speculative state, and a main module that controls two other submodules and deals with timing aspect of a cycle-driven model. Two instances of the instruction-level model could be used to keep track of architecture and speculative state correspondingly.

So, the timing module would issue fetch, execute, retire, abort calls only to the speculative execution submodule. For mispredicted branches, some instructions need to be aborted, but that does not affect the state of the submodule that keeps the committed state. That state is changed only when it is time to commit the instruction—instruction-level simulation is advanced.

This approach works fairly well for CPU microarchitecture exploration. It is relatively easy to make changes to explore new features in the microarchitecture. The changes affect only the timing aspect of the model; functional correctness remains untouched. At the same time, it is hard to provide realistic behavior simulation for the system interface, so system-cycle simulation could be problematic

10.6.4 Conclusion

Overall, there is no "one size fits all" approach when it comes to choosing a cycle-accurate modeling approach. The requirements should be analyzed very carefully to find the right solution.

10.7 Verification by Cosimulation

One important function of the SAM simulator is to provide a CPU golden reference model with which to verify modules under development. In this section, RTL serves as a useful paradigm of cosimulation.

10.7.1 RTL Cosimulation

During RTL development, RTL code is verified by being run in cosimulation mode with SAM's CPU model. The basic idea is that after an instruction is retired on the RTL side, the golden reference model will execute the same instruction and the two sides will then compare architecture register changes caused by the instruction. If there is a mismatch, the cosimulation run is terminated and an error is reported in verification log.

FIGURE 10-3 shows the interaction between the RTL module and the SAM CPU golden reference model.

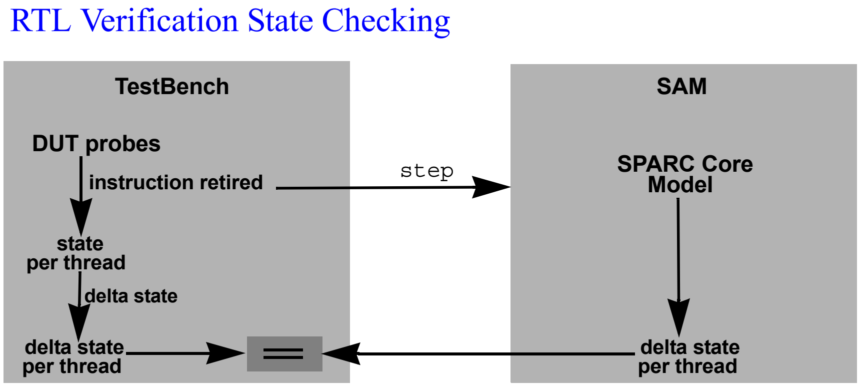


FIGURE 10-3 Interaction Between the RTL Module and SAM CPU Reference Model

The components are connected by a bidirectional socket, which is shown in FIGURE 10-3 as two unidirectional arrows between the components to illustrate their interaction across the socket. The main driver, Verification TestBench, is linked with the RTL module (DUT = device under test), as shown in FIGURE 10-4. TestBench probes DUT activity.

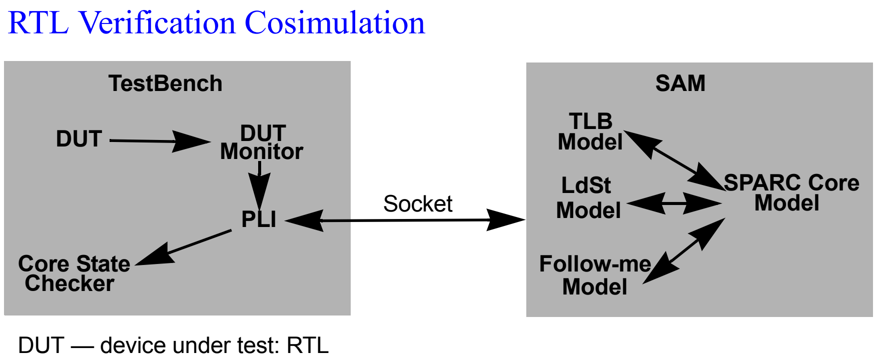


FIGURE 10-4 Models Needed for Multistrand Operation

Whenever TestBench detects that an instruction is retired, it sends a `step` command to SAM through the socket connection. TestBench also collects the DUT's architecture register changes and keeps them in a queue (call it *delta queue*) for later comparison with SAM's architecture register changes. The `step` command syntax indicates which strand should execute one instruction.

On the SAM side, upon receiving the `step` command, the target strand fetches one instruction, based on the strand's program counter (PC) value, and executes the instruction. After the instruction is retired, the related architecture register changes are collected and sent over the socket to TestBench.

When TestBench receives a set of architecture register changes, it finds the corresponding DUT set in its delta queue and compares the two sets of data. If there is a mismatch, TestBench terminates the execution and reports the error.

It is important to note that after TestBench sends a `step` command to SAM, DUT and TestBench do not stop and wait for SAM to complete its corresponding execution and send back architecture register changes (call it *delta set*). Instead, DUT continues its execution, and TestBench monitors DUT's activity and continues to send cosimulation command(s) through the socket to SAM. TestBench periodically checks socket input. When a new delta set is available from SAM, the value is read from the socket and is compared with the corresponding set in TestBench's delta queue. Since DUT and TestBench continue their operation after an instruction is retired and do not wait for SAM to complete its operation, we call the cosimulation a loosely lock-stepped cosimulation.

The preceding description works well for a single-strand execution, but SAM is a functional simulator, it does not have concept of time (that is, cycle count), and it may not have the same strand execution order as DUT's execution order. Because of that, when more than one strand is used in a RTL verification run, more information exchange is needed between TestBench and SAM. FIGURE 10-4 shows the new models that are needed for multistrand operation.

The TLB model (TLB-sync, as it is called) maintains TLB access order among multiple strands. The load-store model (LdSt-sync) maintains cache and memory access order among multiple strands. The Follow-me model synchronizes DUT architecture state that SAM cannot maintain independently. These synchronization models are described in greater detail in the following subsections.

10.7.1.1 TLB-Sync Model

Several factors can affect TLB access order:

- All strands in a core share the same TLB; the sequence in which the strands access the TLB affects the TLB content received by each strand.

- Each strand can fetch several instructions at one time and store them in the ifetch buffer for later use, so there can be many cycles between the time when an instruction is fetched (where TLB is referenced) and the time when the instruction is dispatched to a pipeline for execution. Since SAM does not maintain cycle count and does not model the pipeline in detail, it fetches and executes an instruction in the same cycle.
- SAM does not model the TLB replacement algorithm exactly as DUT does, so a TTE entry's location in the TLB may be different between DUT and SAM. For cosimulation to match every strand and every instruction's execution, the TLB activity must be synchronized between DUT and SAM.

One set of TLB-sync commands maintains consistent TLB state between DUT and SAM. The commands exchange information regarding TLB read, TLB write, and TLB hardware tablewalk. Each TLB-sync command contains a time stamp that represents the cycle count, where DUT conducts the corresponding operation. SAM uses the commands (with the corresponding time stamp) to build up a TLB access history. When a SAM operation requires access to TLB, the history provides access to the proper TLB content. A safeguard built into SAM's TLB-sync model ensures that if DUT and SAM have different TLB access patterns, an error will be reported. For example, if DUT does an ITLB hardware tablewalk and SAM does not, an error is flagged.

FIGURE 10-5 presents an example of TLB-sync usages.

Without TLB-sync

RTL		SAM
T0 ifetch itlb#1 instr1 ts1		
T1 replace itlb#1 with itlb#1-new ts2	→	T1 replace itlb#1 with itlb#1-new
T1 ifetch itlb#1-new instr2 ts3		
T1 retire instr2	→	T1 exec instr2, with itlb#1-new
T0 retire instr1	→	T0 exec **instr?**, with itlb#1-new

With TLB-sync

RTL		SAM
T0 ifetch itlb#1 instr1 ts1	→	*T0 itlb-read ts1*
T1 replace itlb#1 with itlb#1-new ts2	→	*T1 itlb-write itlb#1-new ts2*
	→	T1 replace itlb#1 with itlb#1-new
T1 ifetch itlb#1-new instr2 ts3	→	*T1 itlb-read ts3*
T1 retire instr2	→	T1 exec instr2, with itlb#1-new
T0 retire instr1	→	T0 exec instr1, with itlb#1

FIGURE 10-5 TLB-sync Usages

The top part of the figure shows the TLB discrepancy between DUT and SAM when TLB-sync is not used. The left-hand side shows DUT activity broken down into five operations:

1. At cycle count ts1 (time-stamp 1), strand-0 fetches instruction instr1, using ITLB entry itlb#1.
2. At cycle count ts2, strand-1 replaces ITLB entry itlb#1 with a new entry itlb#1-new.
3. At cycle count ts3, strand-1 fetches instruction instr2 using ITLB entry itlb#1-new.
4. Strand-1 executes and retires instruction instr2.
5. Strand-0 executes and retires instruction instr1.

Because SAM does not model the instruction fetch buffer, it does not separate ITLB access and instruction execution. The two operations occur together when a `step` command is received. That is, when a `step` command arrives from DUT, SAM accesses the ITLB, fetches the corresponding instruction, and then executes the instruction, all in one operation. Because of that, operations (1) and (3) are not visible to SAM. The first operation that happens within SAM is operation (2), by which a ITLB entry is replaced. It is followed by operation (4), by which SAM has strand-1 access ITLB, where itlb#1-new is used; in such case, instruction instr2 is fetched and executed. In operation (5), SAM has strand-0 access ITLB, where itlb#1-new is used, and an instruction other than instr1 is fetched and executed, causing a mismatch between DUT and SAM.

The bottom part of FIGURE 10-5 shows how TLB-sync resolves the problem. Here, TLB-sync commands let DUT inform SAM of all the ITLB activities that take place so that SAM can properly build up an ITLB access history. When strand-0 and strand-1 come to reference ITLB in order to fetch the proper instructions, they give the same content as the strands in DUT have, so the same instructions (as DUTs) are fetched and executed.

10.7.1.2 LdSt-Sync Model

SAM does not model cache; all data accesses go directly to SAM's memory model. This behavior does not conform well to real RTLs, for which two levels of cache and cache coherence keep track of the data access order among all strands (of all cores). For synchronization of the data access order between DUT and SAM, a set of cosimulation commands inform SAM when the following DUT activities occur:

1. A load operation is committed; also reported is whether that load operation gets its data from the L1 cache or others.

2. A load operation's L1 cache miss brings the data into L1 cache.

3. A store operation is committed.

4. A store operation causes cache invalidation and when the cache content changes because of the invalidation.

SAM uses the information provided by DUT to build a data-access history queue, complete with L1 and L2 cache behavior. FIGURE 10-6 shows a sample LdSt-sync operation.

Without LdSt-sync

RTL		SAM
T0 store addr1 data1	→	T0 exec instr store addr1 data1
T8 load addr1 l1$-hit (data0)	→	T8 exec instr load addr1 (**data1**)
C1 store-invalid addr1	→	
T8 load addr1 l1$-miss (data1)	→	T8 exec instr load addr1 (data 1)

With LdSt-sync

RTL		SAM
T0 store addr1 data1	→	T0 exec instr store addr1 data1
	→	*T0 store-commit addr1*
T8 load addr1 l1$-hit (data0)	→	*T8 load-data addr1 l1$-hit*
		T8 exec instr load addr (data0)
C1 store-invalid addr1	→	*C1 store-invalid addr1*
T8 load addr1 l1$-miss (data1)	→	*T8 load-data addr1 l1$-miss*
	→	*T8 load-fill addr1*
		T8 exec instr load addr1 (data1)

FIGURE 10-6 LdSt-Sync Operations

The top part of the figure shows the data access operations in DUT and in SAM. In DUT, several operations are observed:

1. Strand-0 stores data *data1* to address *addr1*.

2. Strand-8, which is in core 1, loads data from address *addr1*. At this point, strand-8's L1 cache is not invalidated, so the load has an L1 cache hit and data *data0* is returned.

3. The cache invalidation request is carried out in core 1, so strand-8's L1 cache entry for address *addr1* becomes invalid at this point.

4. Strand-8 does another load from address *addr1*. This time it encounters an L1 cache miss, and the new data *data1* is loaded into strand-8's L1 cache and returned for the load request.

Because SAM does not model cache, operation (3) is not visible to SAM since all the load and store data go directly to SAM's memory model. And because SAM does not model cache, the data stored at operation (1) is returned, so the load operation gets data *data1*, which is different from DUT's data *data0*. Thus, a mismatch will occur when the delta data is sent over the socket to TestBench. At operation (4), the second load operation by strand-8 will get the same data as its first load operation at operation (2).

The bottom part of FIGURE 10-6 demonstrates the information exchange between TestBench and SAM when the LdSt-sync model added to SAM and TestBench enables the related commands. What happens in this case is described below:

1. When strand-0 stores data *data1* to address *addr1*, the data does not go into SAM's memory model directly; instead, it is kept in SAM's LdSt-sync model, along with other load and store operations. These load and store operations are committed to SAM' s memory only when their predefined condition is satisfied.

2. When strand-8 loads data from address *addr1*, with its resource as "L1 cache hit," the data of the latest store by strand-0 is not used; instead, a data entry that occurs before the store and that fits "L1 cache hit" criteria is used.

3. When core-1 invokes cache invalidation, the corresponding entries in SAM's LdSt-sync model are marked accordingly.

4. When strand-8 does the next load from address *addr1* with "L1 cache miss," TestBench issues a Load-Fill command to instruct SAM to load the corresponding data into SAM LdSt-sync's data-access history queue, such that strand-8v will get the new data stored by strand-0 in operation (1).

Again, SAM's LdSt-sync model does not blindly take information regarding DUT's data access from TestBench. SAM's LdSt-sync model has a built-in safeguard so that if TestBench gives the wrong commands (or order), SAM detects and reports the error.

10.7.1.3 Follow-Me Model

Because SAM does not have the concept of time (or cycle count), it cannot always raise nonprecise traps in synchronization with DUT, which in turn triggers mismatches during cosimulation. To overcome this problem, TestBench sends an Interrupt Sync command to SAM whenever a nonprecise trap is raised at DUT so that SAM can raise the same trap at the same cycle count. A safeguard built into SAM checks a nonprecise trap's priority and condition before the trap is raised; thus, SAM and DUT can maintain some degree of independence even when the Interrupt Sync command is used.

Another use of the "follow me" scheme is to maintain consistency for architecture registers that can be changed by a hardware condition. For example, in the case of a RAS error, the change to a register is not the result of executing an instruction, so SAM cannot change the register at the right moment, that is, at the same cycle count at which DUT makes the change. A "follow me" used on the targeted register ensures that both DUT and SAM have the same register state.

10.7.2 RTL-SAM Cosimulation Summary

In most cases, the DUT and SAM should execute instructions independently and compare their architecture register changes after every instruction execution. But because SAM is a functional model, it does not have the concept of time and does not maintain cycle count. Moreover, since SAM does not simulate cache operation, synchronization operations must be used to synchronize TLB and data access events. For nonprecise traps and hardware-originated register change, the follow-me model may have to be used to keep DUT and SAM's architecture state consistent. In those synchronization and follow-me operations, SAM has built-in safeguards to double-check the synchronization and follow-me action reported by DUT, to ensure that DUT does the same thing in the first place.

Without a golden reference model, RTL verification would become a lot more complicated. For example, more monitors and checkers would have to be built into DUT and TestBench to check DUT state. Without a golden reference model, test diagnostics used in the verification process would need some degree of self-checking so that each test diagnostic could report on its own whether or not the test passes. When cosimulation is conducted with a golden reference model (in this case, SAM), most of the checking burden is shifted to the golden reference model, so DUT and TestBench do not need as many monitors and checkers, and test diagnostics do not need the complexity of built-in self-checking.

In short, the golden reference model (SAM) plays an important role in ensuring the correctness and efficiency of RTL verification.

CHAPTER **11**

OpenSPARC Extension and Modification—Case Study
"Putting OpenSPARC Into an Open Core"

This chapter exemplifies how users can get started with the OpenSPARC RTL and tools. For this chapter, we invited our Italian friends at Simply RISC to describe how they extended and modified OpenSPARC for the Wishbone interface. We are pleased to thank these authors for their enthusiastic acceptance of our invitation.

Their narrative, titled "Put OpenSPARC Into an Open Core," contains the following sections (lightly edited):

About the Authors of Chapter 11

Vincenzo Catania received a degree in Electrical Engineering from the University of Catania in 1982, when he became responsible for microprocessor testing at STMicroelectronics. In 1985 he joined the Department of Computer and Telecommunications Engineering (DIIT) at the University of Catania, where his research interests included parallel/distributed architectures, dealing with architectural issues, management, reliability, fault tolerance, and performance evaluation; hardware/software co-design methods for VLSI systems; embedded and low-power designs. He is now Full Professor of Computer Science and Director of the DIIT department, he has published about 100 papers in international journals and conference proceedings, and he holds two U.S. patents.

Fabrizio Fazzino received a degree in Computer Engineering from the University of Catania in 1997. Until 2001 he was responsible for the functional verification of 32-bit lines of microprocessors at STMicroelectronics. Since 2004 he collaborates with the Department of Computer and Telecommunications Engineering. In 2006 he founded Simply RISC LLP, where he is Managing Director and Chief System Architect.

Putting OpenSPARC Into an Open Core

Starting Up the Project

As soon as OpenSPARC T1 was released, a group of people working at Simply RISC and at the University of Catania started thinking about the development of a cut-down version of OpenSPARC T1 that could be suitable for embedded systems. But we also wanted to gain regard in the open-source community, and to do so we tried importing the design in a GNU/Linux environment, using only free tools. After a while we had built a SPARC core and could simulate it on a fresh installation of Ubuntu. Moreover, developers could download just 1 megabyte, rather the complete OpenSPARC tarball that requires a registration and weighs in at 70 megabytes.

Project Name

We start every project by choosing a proper name. We chose to name this project the “S1 Core,” where the initial “S” stood interchangeably for “single core,” “Simply RISC,” the code name “Sirocco,” or just the letter preceding the “T” of T1 in the Latin alphabet (thus being S1, a decreased version of T1). Hereafter, we refer to this project as the S1 Core.

All the code reported in this chapter is available in the download section of the Simply RISC website (`http://www.srisc.com`), and everything, including the parts originating from Sun Microsystems, is released under the GPL 2 license.

Project Guidelines

To start designing our own derivative project, we established some guidelines to follow.

The first difference between the S1 Core and the T1 processor had to be the number of cores: Usually the communities developing open-source processors do not target the server market (of course!) but instead target the sector of embedded systems, network appliances, and prototyping boards, where area minimization is a key factor.

We chose to use only one SPARC core from the eight available in T1, since we could then execute four threads at the same time while cache misses and other stalls waited in the pipeline.

We also tried to merge the power of the OpenSPARC community with the efforts already existing in the field of open-source hardware design; for instance, with the large community of designers built around the OpenCores.org website. (Please note that the purists of the Free Software Foundation discourage the use of the term "open source" since it focuses on the accessibility of the code and not on the freedom of use).

Once designed, the S1 Core would have to be connected to other Internet Protocol (IP) cores. (Please use the term "IP core" with caution again—Richard Stallman, founder of the Free Software Foundation, does not approve of this name, either!)

Then the original proprietary PCX/CPX bus interface had to be replaced with a standard Wishbone interface, a royalty-free protocol. The protocol was initially developed by a company named Silicore and then donated to the OpenCores community whose site hosts the specification along with several other cores (both masters and slaves) using the same interface. Since not all the details of the PCX/CPX protocol were clear from the specification, we decided to reverse-engineer Sun's protocol by using the waveforms generated by the `sims` script provided with the official OpenSPARC environment.

But to encourage the open-source community, we also decided to make it possible to run all the activities of the newborn S1 Core environment using only free tools (some of them free as in beer and some others free as in speech). We also reduced the size of the tarball by a factor of 50 and removed any kind of registration required to download, thus encouraging downloading (we personally don't like registrations since every site has its own rules for choosing login names and passwords).

So here is our final list of guidelines:

- We will use only one SPARC core of the eight in the full OpenSPARC T1.
- The final design will feature a standard Wishbone interface.
- We will use only free tools to make the environment work.

Using the PCX/CPX Protocol

To interface the SPARC core of the OpenSPARC T1 project with the "external world," we reverse-engineered the protocol as we had decided when setting up the guidelines.

To generate the Value Change Dump (VCD) waveforms, we just added a line into the testbench and ran a single `sims` simulation script.

We also wrote the `tracan` (trace analyzer) tool to convert the waveforms into a more suitable format (similar to that of a log file).

Run on a SPARC Machine

To run the simulations inside the original OpenSPARC environment, you need to have access to a SPARC machine (and this is quite simple) and to some EDA tools (for example, from Synopsys). If your company cannot afford these, you can exploit the open-source approach and ask some academic groups working on OpenSPARC to share with you their results of the script execution (usually even expensive licenses are provided at very cheap prices for university programs). At the end, all they will give you back is a bunch of VCD trace files.

Modify the Testbench

If you want to easily analyze the produced waveforms on your machine by using a free tool, the best way to do so is to modify the testbench to dump the simulation values in a standard format such as VCD.

The testbench inside the OpenSPARC environment does not trace in VCD format "as is," but all you need to do is add a few simple lines of Verilog code into the top-level testbench.

The testbench is in the `$DV_ROOT/verif/env/cmp/cmp_top.v` file. Just add the following four lines into its Verilog module `cmp_top`:

```
initial begin
   $dumpfile("trace.vcd");
   $dumpvars(1, cmp_top.iop.sparc0);
end
```

Run the Simulations

When running the smallest-possible subset of tests, always remember to add the switch that prevents the simulation scripts from deleting the results at the end; otherwise, for each test the VCD will be generated and then deleted before the execution of the next test.

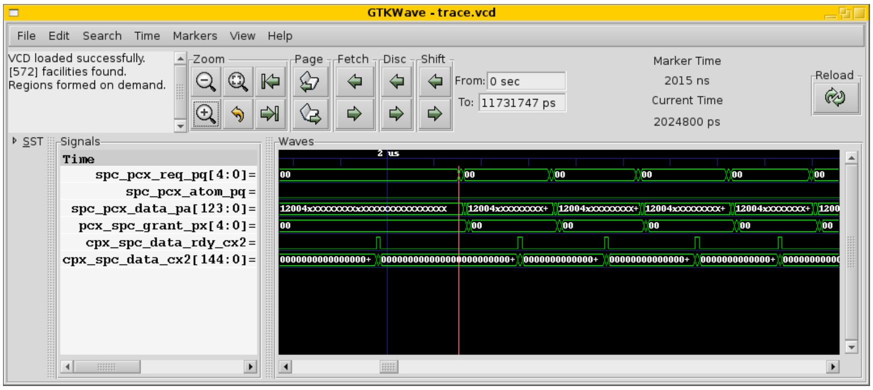
Use this command:

```
sims -sim_type=vcs -group=core1_mini -copyall
```

As the Design and Verification manual pages tell us, the `copyall` command "copies back all files to launch directory after passing regression run. Normally, only failing runs cause a copy back of files."

Now you can look at the waveforms. If you don't have access to commercial tools, just ask other people to lend you the trace file and use GTKWave to show the waveforms. GTKWave is free software, and if you use a common GNU/Linux distribution, you'll find a ready-to-use package for it.

FIGURE 11-1 shows an example of what you would see with GTKWave.



```
VCD loaded successfully.
[572] facilities found.
Regions formed on demand.
From: 0 sec
To: 11731747 ps
Marker Time
2015 ns
Current Time
2024800 ps
```

FIGURE 11-1 Waveforms of the Official OpenSPARC T1 Simulations
The time marker is on the rising edge of the first boot of the SPARC core (the only previous activity is the so-called wake-up packet).

Employ the PCX/CPX Protocol

As you can see in FIGURE 11-1, the protocol itself is simple and the signal names are self-explanatory: it is a request-grant-ready protocol and most information is contained in the packets.

The SPARC core has two packet-based interfaces, one for incoming packets (PCX) and another for outgoing packets (CPX).

Note PCX stands for "Processor-to-Cache Xbar," where "Xbar" stands in turn for "Crossbar interconnect"; CPX stands for "Cache-to-Processor Xbar."

The only ports that connect the Wishbone bridge to the SPARC core are the following:

- **PCX Ports**
  - output `spc_pcx_req_pq`{4:0}
  - output `spc_pcx_atom_pq`

  The 124 bits that compose the PCX packet are interpreted as a valid bit (1 bit), request type (5 bits), noncacheable (1), CPU ID (3), thread ID (2), invalidate (1), prefetch (1), block init store (1), replace L1 way (2), size (3), address (40), and data (64).
- **CPX Ports**
  - input `cpx_spc_data_rdy_cx2`
  - input `cpx_spc_data_cx2`{144:0}

  The 145 bits that compose the CPX packet are interpreted as a valid bit (1 bit), return type (4 bits), L2 miss (1), error (2), noncacheable (1), thread ID (2), way valid (1), cache way (2), and data (128).

Interpret the Signals

The OpenSPARC documentation explains how signals at the boundary of the SPARC cores must be interpreted; watching the waveforms is usually much easier than just reading a table.

The Verilog code for encoding and decoding the PCX/CPX information is shown in EXAMPLE 11-1.

EXAMPLE 11-1 Encoding and Decoding PCX/CPX Information (Verilog)

```
/*
 * Encode/decode incoming info
 *
 * Legend: available constants for some of the PCX/CPX fields.
 *
 * spc2wbm_size (3 bits) is one of:
 * - PCX_SZ_1B
 * - PCX_SZ_2B
 * - PCX_SZ_4B
 * - PCX_SZ_8B
 * - PCX_SZ_16B (Read accesses only)
 *
 * spc2wbm_type (5 bits) is one of:
 * { LOAD_RQ, IMISS_RQ, STORE_RQ, CAS1_RQ, CAS2_RQ, SWAP_RQ,
STRLOAD_RQ,
 *  STRST_RQ, STQ_RQ, INT_RQ, FWD_RQ, FWD_RPY, RSVD_RQ }
 *
 * wbm2spc_type (4 bits) is one of:
 * { LOAD_RET, INV_RET, ST_ACK, AT_ACK, INT_RET, TEST_RET, FP_RET,
 *  IFILL_RET, EVICT_REQ, ERR_RET, STRLOAD_RET, STRST_ACK, FWD_RQ_RET,
 *  FWD_RPY_RET, RSVD_RET }
 *
 */
```

EXAMPLE 11-1 Encoding and Decoding PCX/CPX Information (Verilog) *(Continued)*

```
 *
// Decode info arriving from the SPC side
assign spc2wbm_req = ( spc_req_i[4] | spc_req_i[3] | spc_req_i[2] |
    spc_req_i[1] | spc_req_i[0] );
assign spc2wbm_valid = spc2wbm_packet[`PCX_VLD];
assign spc2wbm_type = spc2wbm_packet[`PCX_RQ_HI:`PCX_RQ_LO];
assign spc2wbm_nc = spc2wbm_packet[`PCX_NC];
assign spc2wbm_cpu_id = spc2wbm_packet[`PCX_CP_HI:`PCX_CP_LO];
assign spc2wbm_thread = spc2wbm_packet[`PCX_TH_HI:`PCX_TH_LO];
assign spc2wbm_invalidate = spc2wbm_packet[`PCX_INVALL];
assign spc2wbm_way = spc2wbm_packet[`PCX_WY_HI:`PCX_WY_LO];
assign spc2wbm_size = spc2wbm_packet[`PCX_SZ_HI:`PCX_SZ_LO];
assign spc2wbm_addr = spc2wbm_packet[`PCX_AD_HI:`PCX_AD_LO];
assign spc2wbm_data = spc2wbm_packet[`PCX_DA_HI:`PCX_DA_LO];

// Encode info going to the SPC side assembling return packets
assign wbm2spc_packet = { wbm2spc_valid, wbm2spc_type, wbm2spc_miss,
    wbm2spc_error, wbm2spc_nc, wbm2spc_thread, wbm2spc_way_valid,
    wbm2spc_way, wbm2spc_boot_fetch, wbm2spc_atomic, wbm2spc_pfl,
    wbm2spc_data };
```

Convert the Waveforms

We converted the VCD trace files into a more suitable format, to simplify counterchecking against our own log file. We wrote a simple C program to convert the files and named it `tracan`, for “trace analyzer.” Using this tool to convert a VCD trace file of the OpenSPARC T1 environment into a plain-text log file, we obtained the result shown in EXAMPLE 11-3.

EXAMPLE 11-2 Conversion of VCD Trace File to Plain-Text Log File

```
INFO: WBM2SPC: *** RETURN PACKET TO SPARC CORE ***
INFO: WBM2SPC: Valid bit is 1
INFO: WBM2SPC: Return Packet of Type Unknown
INFO: WBM2SPC: L2 Miss is 0
INFO: WBM2SPC: Error is 0
INFO: WBM2SPC: Non-Cacheable bit is 0
INFO: WBM2SPC: Thread is 0
INFO: WBM2SPC: Way Valid is 0
INFO: WBM2SPC: Replaced L2 Way is 0
INFO: WBM2SPC: Fetch for Boot is 0
INFO: WBM2SPC: Atomic LD/ST or 2nd IFill Packet is 0
INFO: WBM2SPC: PFL is 0
INFO: WBM2SPC: Data is 00000000000000000000000000010001
INFO: SPC2WBM: *** NEW REQUEST FROM SPARC CORE ***
INFO: SPC2WBM: Valid bit is 1
INFO: SPC2WBM: Request of Type IMISS_RQ
INFO: SPC2WBM: Non-Cacheable bit is 1
INFO: SPC2WBM: CPU-ID is 0
```

EXAMPLE 11-2 Conversion of VCD Trace File to Plain-Text Log File *(Continued)*

```
INFO: SPC2WBM: Thread is 0
INFO: SPC2WBM: Invalidate All is 0
INFO: SPC2WBM: Replaced L1 Way is 3
INFO: SPC2WBM: Request size is 1 Byte
INFO: SPC2WBM: Address is fff0000020
INFO: SPC2WBM: Data is 0000000000000000
INFO: WBM2SPC: *** RETURN PACKET TO SPARC CORE ***
INFO: WBM2SPC: Valid bit is 1
INFO: WBM2SPC: Return Packet of Type IFILL_RET
INFO: WBM2SPC: L2 Miss is 0
INFO: WBM2SPC: Error is 0
INFO: WBM2SPC: Non-Cacheable bit is 1
INFO: WBM2SPC: Thread is 0
INFO: WBM2SPC: Way Valid is 0
INFO: WBM2SPC: Replaced L2 Way is 0
INFO: WBM2SPC: Fetch for Boot is 1
INFO: WBM2SPC: Atomic LD/ST or 2nd IFill Packet is 0
INFO: WBM2SPC: PFL is 0
INFO: WBM2SPC: Data is 03000000050001008210600084l0a0c0
```

Writing the Wishbone Bridge

Then we had to write our own bridge from the PCX/CPX protocols used by the SPARC core to a master Wishbone interface. A schematic view of the bridge function is shown in FIGURE 11-2. The corresponding Verilog file is named `spc2wbm.v` (it stands for “SPARC core to Wishbone Master”).

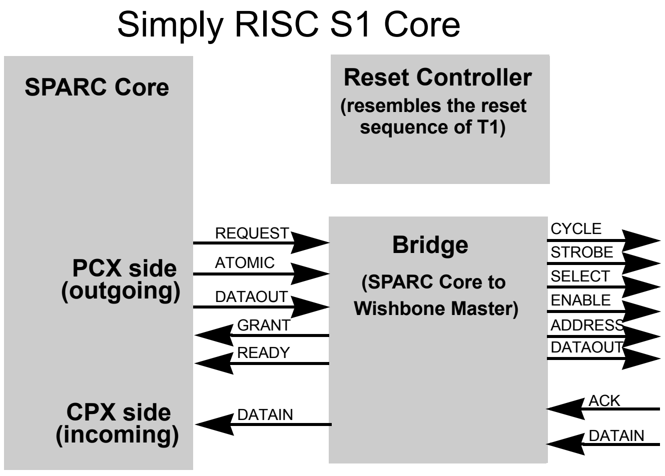


FIGURE 11-2 Our Bridge Just Transforms PCX/CPX Packets Into Wishbone Requests

The Wishbone Protocol

The Wishbone protocol was initiated by a company named Silicore and then donated to the open-source community. There is no longer a website for Silicore, but all the documentation can be found at OpenCores.org.

The bridge from the PCX and CPX protocols of the SPARC core and the Wishbone protocol has been designed as follows:

- The bridge has a Wishbone master interface that follows the Wishbone specification revision B.3.
- In the Verilog code, signal names related to the Wishbone interface are identified by leading wbm_ chars.
- There is no support for errors (ERR/RTY).
- The address bus is 64 bits wide (with some bits unused, since the address bus of the SPARC core refers to 40-bit physical addresses).
- The data bus is 64 bits wide and supports 8-, 16-, 32-, and 64-bit accesses.
- Data transfer ordering is big endian.
- Single read/write cycles are supported.

The Finite State Machine

The Wishbone bridge makes use of a finite state machine (FSM) to serve the requests coming from the SPARC core and convert them to the Wishbone world.

At any moment the FSM can be in one of the following 12 states:

- `S0` — Wake-up (the bridge sends the special interrupt packet to wake up the core)
- `S1` — Idle (stands here until a request arrives)
- `S2` — Request latched
- `S3` — Packet latched
- `S4` — Request granted
- `S5` — Begin second memory access
- `S6` — Second memory access completed (if there was a request with size ≤ 128 bits, can then jump to `S11`)
- `S7` — Begin third memory access if required
- `S8` — Third memory access completed
- `S9` — Begin fourth memory access if required (four 64-bit Wishbone reads can return up to 256 bits of data)

- `S10` — Fourth memory access completed
- `S11` — Packet ready (and then go back to `S1`)

Transitions are allowed from each state to the following one; other transitions are possible where specified.

Choosing the Testbench

All that was missing now was just a top-level testbench with a reset controller. For the first run we chose to run the same memory images of the official OpenSPARC environment.

Simulating With Verilog Icarus

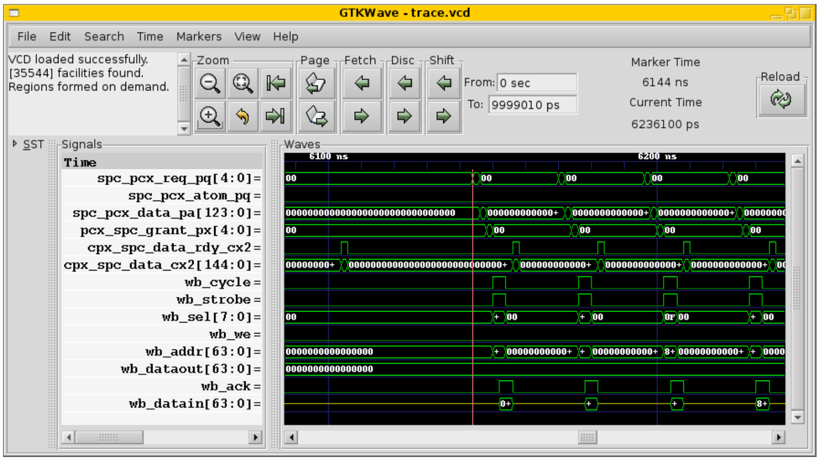
At the end, the only requirement was the free (as in speech!) simulator Icarus Verilog.

Compare Waveforms

When you look at the waveforms, you can see both6 the PCX/CPX and the Wishbone protocols. Obviously, the sequence is as follows:

- The SPARC core sends a PCX packet to the bridge.
- The bridge performs one or more Wishbone accesses to read or write the external memory.
- The bridge sends back a CPX packet to the SPARC core.

FIGURE 11-3 shows what you would see this time with GTKWave.



VCD loaded successfully.
[35544] facilities found.
Regions formed on demand.

From: 0 sec
To: 9999010 ps

Marker Time
6144 ns
Current Time
6236100 ps

FIGURE 11-3 Waveforms Obtained With GTKWave Simulating S1 Core With Icarus Verilog

Compare Log Files

You can use the `tracan` tool to compare the log file with the fake one obtained from the official OpenSPARC environment. As shown in EXAMPLE 11-3, excluding the wake-up packet, the two files are exactly the same!!!

EXAMPLE 11-3 Comparison of Log Files

```
INFO: SPC2WBM: *** NEW REQUEST FROM SPARC CORE ***
INFO: SPC2WBM: Request targeted to I/O Block
INFO: SPC2WBM: Request is not atomic
INFO: SPC2WBM: Valid bit is 1
INFO: SPC2WBM: Request of Type IMISS_RQ
INFO: SPC2WBM: Non-Cacheable bit is 1
INFO: SPC2WBM: CPU-ID is 0
INFO: SPC2WBM: Thread is 0
INFO: SPC2WBM: Invalidate All is 0
INFO: SPC2WBM: Replaced L1 Way is 3
INFO: SPC2WBM: Request size is 1 Byte
INFO: SPC2WBM: Address is fff0000020
INFO: SPC2WBM: Data is 0000000000000000
INFO: WBM2SPC: *** RETURN PACKET TO SPARC CORE ***
INFO: WBM2SPC: Valid bit is 1
INFO: WBM2SPC: Return Packet of Type IFILL_RET
INFO: WBM2SPC: L2 Miss is 0
INFO: WBM2SPC: Error is 0
INFO: WBM2SPC: Non-Cacheable bit is 1
INFO: WBM2SPC: Thread is 0
INFO: WBM2SPC: Way Valid is 0
INFO: WBM2SPC: Replaced L2 Way is 0
INFO: WBM2SPC: Fetch for Boot is 1
INFO: WBM2SPC: Atomic LD/ST or 2nd IFill Packet is 0
INFO: WBM2SPC: PFL is 0
INFO: WBM2SPC: Data is 03000000050001000000000000000000
```

Conclusions

We have used the SPARC core of T1 as a black box. We did not study its internals extensively—anyway, it works! And you can use free tools on a “whatever” Linux box to make it run.

APPENDIX **A**

Overview: OpenSPARC T1/ T2 Source Code and Environment Setup

This appendix gives an overview of the source code in the OpenSPARC T1 and OpenSPARC T2 releases; it also summarizes the requirements for setting up an OpenSPARC environment. The appendix includes these sections:

- *OpenSPARC T1 Hardware Package* on page 239
- *OpenSPARC T2 Hardware Package* on page 243
- *Setup for an OpenSPARC Environment* on page 246

A.1 OpenSPARC T1 Hardware Package

The OpenSPARC T1 hardware package contains all the source code needed to implement an OpenSPARC T1 design. It contains the following components:

- A complete documentation set that includes a microarchitecture specification and a design and verification user's guide
- Complete RTL for the entire design
- A Xilinx EDK project setup to implement a single T1 core on a field-programmable gate array (FPGA) and an environment in which to exercise it
- Libraries of Verilog design block descriptions
- A fully functional verification environment featuring a complete suite of diagnostic tests that verify the design after changes have been made
- Synthesis scripts that synthesize the design into an ASIC library or an FPGA

A.1.1 T1 Hardware Package Structure

The T1 hardware package is available at `www.opensparc.net`. It is distributed as a compressed `tar` file of the directory structure. When uncompressed into a directory, the top-level data structure will appear as follows:

```
BINARY_SLA.txt
GPLv2_License_OpenSPARCT1.txt
OpenSPARCT1.bash
OpenSPARCT1.cshrc
README
THIRDPARTYLICENSEREADME.txt
design/
doc/
lib/
tools/
verif/
```

A.1.2 Documentation

After downloading the hardware package, the user may first want to read some of the documentation to get an overview of the design. In the `doc` directory are five important documents covering various aspects of the design:

- The *OpenSPARC T1 Processor Design and Verification User's Guide* is a complete overview of how to run simulations, synthesize the design, and run the design on an FPGA.
- The *OpenSPARC T1 Processor Data Sheet* describes the OpenSPARC T1 processor at a high level and describes external interfaces of this processor, including the J-Bus interface, the DDR2 memory interface, clocking, reset, and the reliability, availability, and serviceability (RAS) features.
- The *OpenSPARC T1 Processor External Interface Specification* describes in more detail all the external interfaces of the design.
- The *OpenSPARC T1 Processor Megacell Specification* describes the design and implementation of megacells within the design, including RAM or array designs, register files, Translation Lookaside Buffers (TLBs), content-addressable memories (CAMs), and cache structures.
- The *OpenSPARC T1 Processor Microarchitecture Specification* examines the internal design structure of the processor. It describes how the various blocks are designed and documents some of the internal interfaces within the processor.

A.1.3 Design Source Code

The design source code can be found in the following directory:

```
design/sys/iop
```

Each major block of the design has a directory with a standard directory structure under it. This directory structure contains the following elements. Not all blocks will contain every subdirectory listed here.

- The `rtl` directory contains the Verilog® HDL source code for the block.
- The `synopsys` directory contains synthesis scripts for the Design Compiler® synthesis tool from Synopsys®.
- The `synplicity` directory contains synthesis scripts needed for the Synplify® FPGA synthesis tool from Synplicity®.
- The `xst` directory contains synthesis scripts needed by the XST FPGA synthesis tool from Xilinx®.

  The `magellan` directory contains System Verilog Assertions (SVA) properties and scripts for the Magellan™ formal verification tool from Synopsys.

  This tool formally verifies the block. Only the `ccx2mb` block was verified in this manner.
- The block directory will also contain subdirectories for any of the subblocks that it contains.

  For example, the `sparc` directory will contain subdirectories for all of the components of the SPARC core: `ifu`, `lsu`, `exu`, etc.

Following is a description of some of the important blocks of the design:

- The `iop` block (I/O and pads) is the top-level block of the OpenSPARC T1 processor. It contains all the I/O, pads, all eight cores, and level-2 cache.
- The `iop_fpga` block is the top-level block of an FPGA design. It contains a single core. The code for this block is in the `iop/rtl` directory.
- The `sparc` block is the top-level block of the T1 SPARC core.
- The `ccx2mb` block was developed for the Xilinx FPGA design. It adapts the cache-crossbar interface of the SPARC core to the Xilinx Fast Simplex Links (FSL) interface.

A.1.4 Xilinx Embedded Development Kit Project

The Xilinx Embedded Development Kit (EDK) project included in the T1 hardware package enables the operation of a single-core design on Xilinx FPGAs. Users can download a single-core design to an FPGA on an evaluation board, run diagnostic tests on it, bring up the hypervisor layer and run stand-alone programs on top of it, and even boot the Solaris operating system.

The EDK project is located in the following directory:

```
design/sys/edk
```

A.1.5 Design Libraries

Design libraries for the OpenSPARC T1 design are found in the `lib` directory. These library files contain Verilog descriptions of the various standard cell and datapath building blocks of the design.

A.1.6 Verification Environment

The complete verification environment for the OpenSPARC T1 design is located in the `verif` directory. This comprehensive environment allows a user to completely verify the processor after making changes, thus encouraging greater exploration of design enhancements.

The verification system consists of the following components.

- The `env` directory contains the simulation environment files. These files drive stimulus to the processor inputs and check its output. Three environments are available:
  - The `core1` environment, which simulates a single SPARC core
  - The `cmp8` environment, which simulates up to eight SPARC cores
  - The `chip8` environment, which simulates the full processor, including the SPARC cores, and the complete I/O subsystem
- The `model` directory contains models of standard components that are used in the simulation environments.

- The diag directory contains over a thousand diagnostic tests. Most of the tests are assembly language programs that are assembled into an executable file and then simulated on the processor. The diag directory also lists tests for several standard regression suites.

A.1.7 Tools

The OpenSPARC T1 package comes with a set of tools and scripts to assist with the implementation and verification of the design. These are located in the tools directory. Here are some of the tools in the tools/bin directory.

- rsyn script — Runs synthesis on a block, using Design Compiler
- rsynp script — Synthesizes a block to an FPGA netlist, using Synplify
- rxil script — Synthesizes a block to an FPGA netlist, using XST from Xilinx
- sims script — Sets up and runs simulation on the OpenSPARC T1 design
- midas program — Is an assembler that sets up simulations; called by the sims script
- bas program — Is the Simics architectural simulator; runs in parallel with the RTL simulation to verify the correct operation of the RTL
- runreg script — Calls sims to run a regression
- regreport script — Generates a report on the results of a simulation regression run

A.2 OpenSPARC T2 Hardware Package

The structure of the OpenSPARC T2 hardware package is similar to that of the T1 package. It consists of documentation, design source code, a verification environment, and a set of tools. However, OpenSPARC T2 does not yet support FPGA.

A.2.1 Documentation

Like OpenSPARC T1, OpenSPARC T2 contains a complete set of documentation. In the `doc` directory are four important documents covering various aspects of the design:

- The *OpenSPARC T2 Processor Design and Verification User's Guide* gives a complete overview of how to run simulations and synthesize the design.
- The *OpenSPARC T2 Processor Megacell Specification* describes the design and implementation of megacells within the design, including RAM or array designs, register files, Translation Lookaside Buffers (TLBs), content-addressable memories (CAMs), and cache structures.
- The *OpenSPARC T2 Processor Core Microarchitecture Specification* examines the internal design structure of the SPARC core. It describes how the various blocks are designed and documents some of the internal interfaces within the core.
- The *OpenSPARC T2 Processor System-on-Chip (SOC) Microarchitecture Specification* examines the internal design structure of the OpenSPARC T2 processor. This document covers the I/O subsystem, level-2 cache, and everything else outside of the SPARC core. It describes how the various blocks are designed and documents some of the internal interfaces between these blocks.

A.2.2 Design Source Code

The design source code can be found in the following directory:

```
design/sys/iop
```

Each major block of the design has a directory with a standard directory structure under it. This directory structure contains the following elements. Not all blocks will contain every subdirectory listed here.

- The `rtl` directory contains the Verilog HDL source code for the block.
- The `synopsys` directory contains synthesis scripts for the Design Compiler synthesis tool from Synopsys.
- The `jasper` directory contains System Verilog Assertions (SVA) properties, Verilog code, and scripts for the JasperGold® formal verification tool from Jasper Design Automation®. JasperGold formally verifies certain blocks.

Note that this code was not in the first release of OpenSPARC T2 but will be included in later releases.

- The `magellan` directory contains SVA properties and scripts for the Magellan formal verification tool from Synopsys. Magellan is another formal verification tool that verifies certain blocks.

  Like the code in the `jasper` subdirectory, this code was not in the first release of OpenSPARC T2 but will be included in later releases.
- The block directory will also contain subdirectories for any of the subblocks that it contains.

  For example, the `sparc` directory will contain subdirectories for all of the components of the SPARC core: `ifu`, `lsu`, `exu`, etc.

Following is a description of some of the important blocks of the T2 design:

- The `cpu` block is the top-level block of the OpenSPARC T2 processor. It contains all the I/O, pads, all eight cores, and level-2 cache.
- The `spc` (SPARC core) block is the top-level block of the T2 SPARC core.

A.2.3 Design Libraries

Design libraries for the OpenSPARC T2 design are found in the `lib` directory. These library files contain Verilog descriptions of the various standard cell and datapath building blocks of the design.

A.2.4 Verification Environment

The complete verification environment for the OpenSPARC T2 design is located in the `verif` directory. This comprehensive environment allows a user to completely verify the processor after making changes, thus encouraging greater exploration of design enhancements.

The verification system consists of the following components.

- The `env` directory contains the simulation environment files. These files drive stimulus to the processor inputs and check its output. Two environments are available:
  - The `cmp1` environment, which simulates a single SPARC core, crossbar, and level-2 cache
  - The `fc` environment, which simulates up to eight SPARC cores, crossbar, level-2 cache, and full I/O subsystem
- The `model` directory contains models of standard components that are used in the simulation environments.

- The diag directory contains over a thousand diagnostic tests. Most of the tests are assembly language programs that are assembled into an executable file and then simulated on the processor. The diag directory also lists tests for several standard regression suites.

A.2.5 Tools

The OpenSPARC T2 hardware package comes with a set of tools and scripts to assist with the implementation and verification of the design. These are located in the tools directory. Note that although the OpenSPARC T2 package has many of the same tools as the OpenSPARC T1 package, there may be different versions of these tools and the required tool options may be different. Here are some of the tools in the tools/bin directory.

- rsyn script — Runs synthesis on a block, using Design Compiler
- sims script — Sets up and runs simulation on the OpenSPARC T2 design
- midas program — Is an assembler that sets up simulations; called by the sims script
- nas program — Is the Riesling architectural simulator for OpenSPARC T2; run by the sims script in parallel with the RTL simulation to verify the correct operation of the RTL
- runreg script — Calls sims to run a regression

A.3 Setup for an OpenSPARC Environment

The environment setup for OpenSPARC environments is the same for OpenSPARC T1 and OpenSPARC T2. In the top-level directory of each release is a C shell script to set the proper environment for the project. The script can be copied and then customized to the user's environment.

The OpenSPARC projects require the definition of the environment variables listed below.

- The PROJECT variable is set by the script to either OpenSPARC T1 or OpenSPARC T2.
- The DV_ROOT variable should be modified to point to the location where the OpenSPARC project has been placed.

- The `MODEL_DIR` variable should point to the location where the user wants to run any simulations.
- Other environment variables may have to be set to enable other third-party tools that are needed.

For more information on setting up the environment, see the "Quick Start" chapter of either the *OpenSPARC T1 Processor Design and Verification User's Guide* or the *OpenSPARC T2 Processor Design and Verification User's Guide*.

APPENDIX **B**

Overview of OpenSPARC T1 Design

This appendix adds to the material presented in Chapter 4 and was excerpted from *OpenSPARC T1 Microarchitecture Specification*, Part number 819-6650-10 August 2006, Revision A. See that manual for specific details about the material covered in this appendix. The appendix contains these sections:

See FIGURE 4-2 on page 28 for a block diagram of the OpenSPARC T1 chip.

B.1 SPARC Core

Each SPARC core has the following units:

- **Instruction fetch unit (IFU)** — Maintains the program counters (PCs) of different threads and fetches the corresponding instructions. It also includes the following pipeline stages—fetch, thread selection, and decode—and an instruction cache complex. For every SPARC core clock cycle, two instructions are fetched for every instruction issued.

- **Execution unit (EXU) —** Generates the necessary select signals that control the multiplexors, keeps track of the thread and reads of each instruction, generates the write-enables for the integer register file (IRF), and implements the bypass logic. It includes the execute stage of the pipeline and contains four subunits:
  - Arithmetic and logic unit (ALU)
  - Shifter (SHFT)
  - Integer multiplier (IMUL)
  - Integer divider (IDI)
- **Load-store unit (LSU) —** Processes memory referencing operation codes (opcodes) such as various types of loads, various types of stores, CAS, SWAP, LDSTUB, FLUSH, PREFETCH, and MEMBAR instructions. The LSU interfaces with all the SPARC core functional units and acts as the gateway between the SPARC core units and the CPU-cache crossbar (CCX). Through the CCX, data transfer paths can be established with the memory subsystem and the I/O subsystem (the data transfers are done with packets). The LSU includes memory and write-back stages and a data cache complex. The LSU pipeline has four stages:
  - E stage – Cache and TLB setup
  - M stage – Cache/Tag TLB read
  - W stage – Store buffer look-up, trap detection, and execution of data bypass
  - W2 stage – Generates PCX requests and write-backs to the cache
- **Trap logic unit (TLU) —** Supports six trap levels. A trap can be in one of the following four modes:
  - Reset-error-debug (RED) mode
  - Hypervisor (HV) mode
  - Supervisor (SV) mode
  - User mode

  Traps cause the SPARC core pipeline to be flushed and a thread-switch to occur until the trap vector (redirect PC) has been resolved.

  Software interrupts are delivered to each of the virtual cores by the *interrupt_level_n* exception through the SOFTINT_REG register. I/O and CPU cross-call interrupts are delivered to each virtual core by the *interrupt_vector* exception. Up to 64 outstanding interrupts can be queued per thread—one for each interrupt vector.
- **Stream processing unit (SPU)** — used for modular arithmetic functions for crypto.

- **Memory management unit (MMU)** — Maintains the contents of the instruction Translation Lookaside Buffer (ITLB) and the data Translation Lookaside Buffer (DTLB). The ITLB resides in the IFU, and the DTLB resides in LSU. FIGURE 11-4 illustrates the role of the MMU in virtualization.

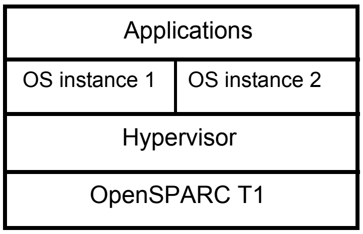


FIGURE 11-4 Visualization Diagram

The Hypervisor (HV) layer virtualizes the underlying central processing units. The multiple instances of the OS images form multiple partitions of the underlying virtual machine. The Hypervisor ensures that failure in one domain would not affect the operation in the other domains. The OpenSPARC T1 processor supports up to eight partitions, and the hardware provides three bits of partition ID to distinguish one partition from another.

- **Floating-point front-end unit (FFU)** — Dispatches floating-point operations (FP ops) to the floating-point unit (FPU) through the LSU, executes simple FP ops (MOV, ABS, NEG) and VIS instructions, and maintains the Floating-Point State register (FSR) and the Graphics State register (GSR).

B.2 L2 Cache

The OpenSPARC T1 processor L2 cache is 3 Mbytes in size and is composed of four symmetrical banks that are interleaved on a 64-byte boundary. Each bank operates independently of the others. The banks are 12-way set associative and 768 Kbytes in size. The block (line) size is 64 bytes, and each L2 cache bank has 1024 sets.

The L2 cache accepts requests from the SPARC CPU cores on the processor-to-cache crossbar (PCX), responds on the cache-to-processor crossbar (CPX), and maintains on-chip coherency across all L1 caches on the chip by keeping a copy of all L1 tags in a directory structure. Since the OpenSPARC T1 processor implements System-On-a-Chip, with single memory interface and no L3 cache, no off-chip coherency is required for the OpenSPARC T1 L2 cache other than that the L2 cache must be coherent with main memory.

Each L2 cache bank consists of three main subblocks:

- *sctag* (secondary cache tag) — Contains the tag array, VUAD array, L2 cache directory, and the cache controller
- *scbuf* — Contains the write-back buffer (WBB), fill buffer (FB), and DMA buffer
- *scdata* — Contains the *scdata* array

Coherency and ordering in the L2 cache are described as follows:

- Loads update directory and fill the L1 cache on return.
- Stores are nonallocating in the L1 cache.
  - There are two flavors of stores: total store order (TSO) and read memory order (RMO).

    Only one outstanding TSO store to the L2 cache per thread is permitted in order to preserve the store ordering. There is no such limitation on RMO stores.
  - No tag check is done at a store buffer insert.
  - Stores check the directory and determine an L1 cache hit.
  - The directory sends store acknowledgments or invalidations to the SPARC core.
  - Store updates happen to D$ on a store acknowledge.
- The crossbar orders the responses across cache banks.

B.2.1 L2 Cache Single Bank

The L2 cache is organized into four identical banks. Each bank has its own interface with the J-Bus, the DRAM controller, and the CCX. The L2 cache consists of the following blocks:

- **Arbiter** — Manages the access to the L2 cache pipeline from the various sources that request access. Gets input from the following:
  - Instructions from the CCX and from the bypass path for input queue (IQ)
  - DMA instructions from the snoop input queue
  - Instructions for recycle from the fill buffer and the miss buffer
  - Stall signals from the pipeline
- **L2 tag** — Contains the *sctag* array and the associated control logic. Each 22-bit tag is protected by 6 bits of SEC ECC. The state of each line is maintained by valid (v), used (u), allocated (a), and dirty (d) bits. These bits are stored in the L2 VUAD array.

- **L2 VUAD array** — Organizes the four state bits for *sctag*s in a dual-ported array structure:
  - Valid (v) – Set when a new line is installed in a valid way; reset when that line is invalidated.
  - Used (u) – Is a reference bit used in the replacement algorithm. Set when there are any store/load hits (1 per way); cleared when there are no unused or unallocated entries for that set.
  - Allocated (a) – Set when a line is picked for replacement. For a load or an ifetch, cleared when a fill happens; for a store, when the store completes.
  - Dirty (d) – (per way) Set when a store modifies the line; cleared when the line is invalidated.
- **L2 data** (*scdata*) — Is a single-ported SRAM structure. Each L2 cache bank is 768 Kbytes in size, with each logical line 64 bytes in size. The bank allows read access of 16 bytes and 64 bytes, and each cache line has 16 byte-enables to allow writing into each 4-byte part. A fill updates all 64 bytes at a time.

  Each 32-bit word is protected by seven bits of SEC/DED ECC. (Each line is $32 \times [32 + 7 \text{ ECC}] = 1248$ bits). All subword accesses require a read-modify-write operation to be performed, referred to as partial stores.
- **Input queue** (IQ, a 16-entry FIFO) — Queues the packets arriving on the PCX when they cannot be immediately accepted into the L2 cache pipe. Each entry in the IQ is 130 bits wide. The FIFO is implemented with a dual-ported array. The write port is used for writing into the IQ from the PCX interface. The read port is for reading contents for issue into the L2 cache pipeline.
- **Output queue** (OQ, a 16-entry FIFO) — Queues operations waiting for access to the CPX. Each entry in the OQ is 146 bits wide. The FIFO is implemented with a dual-ported array. The write port is used for writing into the OQ from the L2 cache pipe. The read port is used for reading contents for issue to the CPX.
- **Snoop input queue** (SNPIQ, a 2-entry FIFO) — Stores DMA instructions coming from the JBI. The nondata portion (the address) is stored in the SNPIQ. For a partial line write (WR8), both the control and the store data are stored in the SNPIQ.
- **Miss buffer** (MB) — Stores instructions that cannot be processed as a simple cache hit; includes the following:
  - True L2 cache misses (no tag match)
  - Instructions that have the same cache line address as a previous miss or an entry in the write-back buffer

  - Instructions requiring multiple passes through the L2 cache pipeline (atomics and partial stores)
  - Unallocated L2 cache misses
  - Accesses causing tag ECC errors

  A read request is issued to the DRAM and the requesting instruction is replayed when the critical quadword of data arrives from the DRAM. All entries in the miss buffer that share the same cache line address are linked in the order of insertion to preserve coherency. Instructions to the same address are processed in age order, whereas instructions to different addresses are not ordered and exist as a free list.

- **Fill buffer** (FB) — Temporarily stores data arriving from the DRAM on an L2 cache miss request. The fill buffer is divided into a RAM portion, which stores the data returned from the DRAM waiting for a fill to the cache, and a CAM portion, which contains the address. The fill buffer has a read interface with the DRAM controller.
- **Write-back buffer** (WBB) — Stores the 64-byte evicted dirty data line from the L2 cache. The WBB is divided into a RAM portion, which stores evicted data until it can be written to the DRAM, and a CAM portion, which contains the address.The WBB has a 64-byte read interface with the *scdata* array and a 64-bit write interface with the DRAM controller. The WBB reads from the *scdata* array faster than it can flush data out to the DRAM controller.
- **Remote DMA write buffer** (RDMA–4-entry buffer) — Accommodates the cache line for a 64-byte DMA write. The output interface is with the DRAM controller, which it shares with the WBB. The WBB has a direct input interface with the JBI.
- **L2 cache directory** — Participates in coherency management and maintains the inclusive property of the L2 cache. Also ensures that the same line is not resident in both the I-cache and the D-cache (across all CPUs).

  Each L2 cache directory has 2048 entries, with one entry per L1 tag that maps to a particular L2 cache bank. Half the entries correspond to the L1 instruction cache (I-cache), and the other half of the entries correspond to the L1 data cache (D-cache).

  The L2 cache directory is split into two directories, which are similar in size and functionality: an I-cache directory (`icdir`) and a D-cache directory (`dcdir`).

B.2.2 L2 Cache Instructions

The following instructions follow a skewed, rather than simple, pipeline.

- A load instruction to the L2 cache is caused by any one of the following conditions:
  - A miss in the L1 cache (the primary cache) by a load, prefetch, block load, or a quad load instruction
  - A streaming load issued by the stream processing unit (SPU)
  - A forward request read issued by the IOB

  The output of the *scdata* array, returned by the load, is 16 bytes in size. This size is same as the size of the L1 data cache line. An entry is created in the D-cache directory. An I-cache directory entry is invalidated if it exists. An I-cache directory entry is invalidated for the L1 cache of every CPU in which it exists.

  From an L2 cache perspective, a block load is the same as eight load requests. A quad load is same as four load requests.
- An ifetch instruction is issued to the L2 cache in response to an instruction missing the L1 I-cache. The size of I-cache is 256-bits. The L2 cache returns the 256 bits of data in two packets over two cycles to the requesting CPU over the CPX. The two packets are returned as an atomic. The L2 cache then creates an entry in the I-cache directory and invalidates any existing entry in the D-cache directory.
- A store instruction to the L2 cache is caused by any of the following conditions:
  - A miss in the L1 cache by a store, block store, or a block init store instruction
  - A streaming store issued by the stream processing unit (SPU)
  - A forward request write issued by the IOB

  The store instruction writes (in a granularity of) 32-bits of data into the *scdata* array. An acknowledgment packet is sent to the CPU that issued the request, and an invalidate packet is sent to all other CPUs. The I-cache directory entry for every CPU is cammed and invalidated. The D-cache directory entry of every CPU, except the requesting CPU, is cammed and invalidated.

  Partial stores (PSTs) perform sub-32-bit writes into the *scdata* array. A partial store is executed as a read-modify-write operation. In the first step, the cache line is read and merged with the write data. It is then saved in the miss buffer. The cache line is written into the *scdata* array in the second pass of the instruction through the pipe.
- Three types of atomic instructions are processed by the L2 cache—load-store unsigned byte (LDSTUB), SWAP, and compare and swap (CAS). These instructions require two passes down the L2 cache pipeline.

- The following I/O instructions from the J-Bus interface (JBI) are processed by the L2 cache:
  - RD64 (block read) — Goes through the L2 cache pipe like a regular load from the CPU. On a hit, 64 bytes of data are returned to the JBI. On a miss, the L2 cache does not allocate but sends a nonallocating read to the DRAM. It gets 64 bytes of data from the DRAM and sends the data directly to the JBI (read-once data only) without installing it in the L2 cache. The CTAG (the instruction identifier) and the 64-byte data are returned to the JBI on a 32-bit interface.
  - WRI (write invalidate) — Accepts a 64-byte write request and looks up tags as it goes through the pipe. Upon a tag hit, invalidates the entry and all primary cache entries that match. Upon a tag miss, does nothing (it just continues down the pipe) to maintain the order. The CTAG is returned to the JBI when the processor sends an acknowledgment to the cache line invalidation request sent over the CPX. After the instruction is retired from the pipe, 64 bytes of data are written to the DRAM.
  - WR8 (partial line write) — Supports the writing of any subset of eight contiguous bytes to the *scdata* array by the JBI. Does a two-pass partial store if an odd number of byte enables are active or if there is a misaligned access; otherwise, does a regular store. On a miss, data is written into the *scdata* cache and a directory entry is not created. The CTAG is returned to the JBI when the processor sends an acknowledgment to the cache line invalidation request sent over the CPX.
  - Eviction — Sends a request to the DRAM controller to bring the cache line from the main memory after a load or a store instruction miss in the L2 cache.
  - Fill — Issued following an eviction after an L2 cache store or load miss. The 64-byte data arrives from the DRAM controller and is stored in the fill buffer. Data is read from the fill buffer and written into the L2 cache *scdata* array.
  - L1 cache invalidation — Invalidates the four primary cache entries as well as the four L2 cache directory entries corresponding to each primary cache tag entry. Is issued whenever the CPU detects a parity error in the tags of I-cache or D-cache.
  - Interrupts — When a thread wants to send an interrupt to another thread, that interrupt is sent through the L2 cache. The L2 cache treats the thread like a bypass. After a decode, the L2 cache sends the instruction back to destination CPU if it is a interrupt.

- Flush — The OpenSPARC T1 processor requires the FLUSH instruction. Whenever a self-modifying code is performed, the first instruction at the end of the self-modifying sequence should come from a new stream. An interrupt with br = 1 is broadcast to all CPUs. (Such an interrupt is issued by a CPU in response to a FLUSH instruction.) A flush stays in the output queue until all eight receiving queues are available. This is a total store order (TSO) requirement.

B.2.3 L2 Cache Pipeline

L2 cache pipeline — The L2 cache processes three main types of instructions:

- Requests from a CPU by way of the PCX. Instructions include load, streaming load, ifetch, prefetch, store, streaming store, block store, block init store, atomics, interrupt, and flush.
- Requests from the I/O by way of the JBI. Instructions include block read (RD64), write invalidate (WRI), and partial line write (WR8).
- Requests from the IOB by way of the PCX. Instructions include forward request load and forward request store (these instructions are used for diagnostics).

The L2 cache access pipeline has eight stages (C1 to C8). Cache miss instructions are reissued from the miss buffer after the data returns from the DRAM controller; reissued instructions follow the L2 cache pipeline.

B.2.4 L2 Cache Memory Coherency and Instruction Ordering

Cache coherency is maintained by a mixture of structures in the miss buffer, fill buffer, and the write-back buffer. The miss buffer maintains a dependency list for the access to the 64 bytes of cache lines with the same address. Responses are sent to the CPUs in the age order of the requests for the same address.

The L2 cache directory maintains the cache coherency in all primary caches. The L2 cache directory preserves the inclusion property—all valid entries in the primary cache should reside in the L2 cache as well. It also keeps the I-cache and D-cache exclusive for each CPU.

B.3 Memory Controller

See Section 4.3.4, *DRAM Controller*, on page 31.

B.4 I/O Bridge

The I/O bridge (IOB) performs an address decode on I/O-addressable transactions and directs them to the appropriate internal block or to the appropriate external interface (J-Bus or the serial system interface). Additionally, the IOB maintains the register status for external interrupts.

B.4.1 IOB Main Functions

The main IOB functions include the following:

- **I/O address decoding** — The IOB
  - maps or decodes I/O addresses to the proper internal or external destination;
  - generates control/status register (CSR) accesses to the IOB, JBI, DRAM, and CTU clusters;
  - generates programmed I/O (PIO) accesses to the external J-Bus.
- **Interrupt handling** — The IOB
  - collects the interrupts from clusters (errors and EXT_INT_L) and mondo interrupts from the J-Bus;
  - forwards interrupts to the proper core and thread;
  - wakes up a single thread at reset.
- **And more** — The IOB
  - interfaces between the read/write/ifill to the serial system interface (SSI);
  - enables test access ports (TAPs) to access CSRs, memory, the L2 cache, and CPU ASIs.
  - provides debug port functionality (both to an external debug port and to the JBI).

The IOB operates in both the CMP and J-Bus clock domains.

B.4.2 IOB Miscellaneous Functions

Other IOB functions include the following:

- **Launching of one thread after reset**
  - Sends resume interrupt to thread 0 in the lowest available core (the EFC sends the available cores information)
  - The RSET_STAT CSR shows the RO and RW status for POR, FREQ, and WRM
- **Making eFuse data visible to software —** Serial data is shifted in after a power-on reset (POR)
  - CORE_AVAIL
  - PROC_SER_NUM
  - IOB_EFUSE – contains parity check results from the EFC. If non-zero, the chip is suspect with a potentially bad CORE_AVAIL or a memory array redundancy).
- **Managing power —** Thermal sensor sends an idle/resume interrupt to threads specified in the tm_stat_ctl mask.

B.4.3 IOB Interfaces

The following are the main interfaces to and from the IOB.

- **Crossbar (CCX) –** Interface to the PCX to the CPX (both are parallel interfaces).
- **Universal connection bus (UCB) –** a common packetized interface to all clusters for CSR accesses.
  - Common width-parameterized blocks in the IOB and clusters
  - Separate request and acknowledge/interrupt paths with parameterized widths, various blocks, and widths
  - IOB is master; cluster/block, slave, with the exception of the test port access (TAP), which is both master and slave
  - All interfaces visible through the debug ports
- **J-Bus mondo interrupt interface:**
  - Sixteen-bit request interface and a valid bit
  - Header with 5-bit source and target (thread) IDs
  - Eight cycles of data – 128 bits (J-Bus Mondo Data 0 and 1)
  - Two-bit acknowledge interface – ACK and NACK
- **EFuse controller (EFC) –** Serial interface:

    - Shifted-in at power-on reset (POR) to make the software visible (read-only)
    - CORE_AVAIL
    - PROC_SER_NUM
- **Debug ports:**
    - Internal visibility port on each UCB interface
    - L2 cache visibility port input from the L2 cache (2 × 40 bits @ CMP clock)
    - Debug port A output to the debug pads (40 bits @ J-Bus clock)
    - Debug port B output to the JBI (2 × 48 bits @ J-Bus clock)

B.5 Floating-Point Unit (FPU)

The OpenSPARC T1 floating-point unit (FPU) has the following features and supports the following functions.

- The FPU implements the SPARC V9 floating-point instruction set with the following exceptions:
    - Does not implement FSQRT<s|d> and all quad precision instructions
    - Move-type instructions executed by the SPARC core floating-point front-end unit (FFU): FMOV<s|d>, FMOV<s|d>cc, FMOV<s|d>r, FABS<s|d>, FNEG<s|d>
- Loads and stores (the SPARC core FFU executes these operations).
- The FPU does not support the visual instruction set (VIS). (The SPARC core FFU provides limited VIS support.)
- The FPU is a single shared resource on the OpenSPARC T1 processor. Each of the eight SPARC cores may have a maximum of one outstanding FPU instruction. A thread with an outstanding FPU instruction stalls (switches out) while waiting for the FPU result.
- The floating-point register file (FRF) and floating-point state register (FSR) are not physically located within the FPU. The SPARC core FFU owns the register file and FSR. The SPARC core FFU also performs odd/even single-precision address handling.
- The FPU complies with the IEEE 754 standard.
- The FPU includes three independent execution pipelines:
    - Floating-point adder (FPA) – Adds, subtracts, or compares conversions
    - Floating-point multiplier (FPM) – Multiplies
    - Floating-point divider (FPD) – Divides

- One instruction per cycle may be issued from the FPU input FIFO queue to one of the three execution pipelines.
- One instruction per cycle may complete and exit the FPU.
- All IEEE 754 floating-point data types (normalized, denormalized, NaN, zero, infinity) are supported. A denormalized operand or result will never generate an unfinished_FPop trap to the software. The hardware provides full support for denormalized operands and results.
- IEEE nonstandard mode (FSR.ns) is ignored by the FPU.
- The following instruction types are fully pipelined and have a fixed latency, independent of operand values: add, subtract, compare, convert between floating-point formats, convert floating point to integer, convert integer to floating point.
- The following instruction types are not fully pipelined: multiply (fixed latency, independent of operand values), divide (variable latency, dependent on operand values).
- Divide instructions execute in a dedicated datapath and are nonblocking.
- Underflow tininess is detected before rounding. Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded (inexact condition).
- A precise exception model is maintained. The OpenSPARC T1 implementation does not require early exception detection/prediction. A given thread stalls (switches out) while waiting for an FPU result.
- The FPU includes three parallel pipelines and these pipelines can simultaneously have instructions at various stages of completion.

TABLE B-1 summarizes FPU features.

TABLE B-1 OpenSPARC T1 FPU Feature Summary

Feature	**OpenSPARC T1 Processor FPU Implementation**
ISA	SPARC V9
VIS	Not available
Issue	1
Register file	In FFU
FDIV blocking	No
Full hardware denorm support	Yes
Hardware quad support	No

B.5.1 Floating-Point Instructions

TABLE B-2 describes the floating-point instructions, including the execution latency and the throughput for each instruction.

TABLE B-2 SPARC V9 Single- and Double-Precision FPop Instruction Set

Mnemonic	**Description**	**Pipe**	**Execution Latency**	**Through-put**
FADD<s|d>	Floating-point add	FPA	4	1/1
FSUB<s|d>	Floating-point subtract	FPA	4	1/1
FCMP<s|d>	Floating-point compare	FPA	4	1/1
FCMPE<s|d>	Floating-point compare (exception if unordered)	FPA	4	1/1
F<s|d>TO<d|s>	Convert between floating-point formats	FPA	4	1/1
F<s|d>TOi	Convert floating point to integer	FPA	4	1/1
F<s|d>TOx	Convert floating point to 64-bit integer	FPA	4	1/1
FiTOd	Convert integer to floating point	FPA	4	1/1
FiTOs	Convert integer to floating point	FPA	5	1/1
FxTO<s|d>	Convert 64-bit integer to floating point	FPA	5	1/1
FMUL<s|d>	Floating-point multiply	FPM	7	1/2
FsMULd	Floating-point multiply single to double	FPM	7	1/2
FDIV<s|d>	Floating-point divide	FPD	32 SP, 61 DP (less for zero or denormalized results)	29 SP, 58 DP (less for zero or denormalized results)
FSQRT<s|d>	Floating-point square root	*Unimplemented*		
FMOV<s|d>	Floating-point move	*Executed in the SPARC core FFU*		
FMOV<s|d>cc	Move floating-point register if condition is satisfied			
FMOV<s|d>r	Move floating-point register if integer register contents satisfy condition			
FABS<s|d>	Floating-point absolute value			
FNEG<s|d>	Floating-point negate			

B.5.2 Floating-Point Unit Power Management

FPU power management is accomplished by way of block-controllable clock gating. Clocks are dynamically disabled or enabled as needed.

The FPU has independent clock control for each of the three execution pipelines (FPA, FPM, and FPD). Clocks are gated for a given pipeline when it is not in use, so a pipeline has its clocks enabled only under one of the following conditions:

- The pipeline is executing a valid instruction.
- A valid instruction is issuing to the pipeline.
- The reset is active.
- The test mode is active.

The input FIFO queue and output arbitration blocks receive free running clocks.

The FPU power-management feature automatically powers up and powers down each of the three FPU execution pipelines, based on the contents of the instruction stream. Also, the pipelines are clocked only when required.

B.5.3 Floating-Point Register Exceptions and Traps

The SPARC core FFU physically contains the architected floating-point state register (FSR). The characteristics of the FSR, as well as exceptions and traps, are as follows:

- The FFU provides FSR.rd (IEEE rounding direction) to the FPU. IEEE nonstandard mode (FSR.ns) is ignored by the FPU and thus is not provided by the FFU.
- The FFU executes all floating-point move (FMOV) instructions. The FPU does not require any conditional move information. A 2-bit FSR condition code (fcc) field identifier (fcc0, fcc1, fcc2, fcc3) is provided to the FPU so that the floating-point compare (FCMP) target fcc field is known when the FPU result is returned to the FFU.
- The FPU provides IEEE exception status flags to the FFU for each instruction completed. The FFU determines if a software trap (*fp_exception_ieee_754*) is required based on the IEEE exception status flags supplied by the FPU and the IEEE trap enable bits located in the architected FSR.

- A denormalized operand or result will never generate an unfinished_FPop trap to the software. The hardware provides full support for denormalized operands and results.
- Each of the five IEEE exception status flags—invalid operation (`nv`), division by zero (`dz`), overflow (`of`), underflow (`uf`), and inexact (`nx`)—and associated trap enables are supported.
- IEEE traps enabled mode – If an instruction generates an IEEE exception when the corresponding trap enable is set, then an *fp_exception_ieee_754* trap is generated and results are inhibited by the FFU.
  - The destination register remains unchanged.
  - FSR condition codes (fcc) remain unchanged.
  - FSR.aexc field remains unchanged.
  - FSR.cexc field has one bit set corresponding to the IEEE exception.
- All four IEEE round modes are supported in hardware.

B.6 J-Bus Interface

The OpenSPARC T1 J-Bus interface (JBI) generates J-Bus transactions and responds to external J-Bus transactions. In a T1 processor, the JBI interfaces with the L2 cache, the I/O Bridge, and J-Bus I/O pads.

Two subblocks—J-Bus parser and J-Bus transaction issue—in the JBI are specific to the J-Bus. All the other blocks are J-Bus independent. J-Bus-independent blocks can be used for any other external bus interface implementation.

Most of the JBI subblocks use the J-Bus clock, and the remainder run at the CPU core clock. The data transfer between the two clock domains is by way of queues within the two clock domains: the Request header queues and the Return data queues. The interface to the L2 cache is through the direct memory access (DMA) reads and DMA writes.

The IOB debug port data is stored in the debug FIFOs and sent out to the external J-Bus.

IOB PIO requests are stored in the PIO queue, and the return data is stored in the PIO return queue. Similarly, an interrupt queue and an interrupt ACK/NACK queue in the JBI interface to the IOB.

B.6.1 J-Bus Requests to the L2 Cache

The J-Bus sends two types of requests to the L2 cache: read and write.

A DMA read request from the J-Bus is parsed by the J-Bus parser. The information is passed to the write decomposition queue (WDQ), which sends the request header to the *sctag* of the L2 cache. Data returned from the L2 cache *scbuf* is then passed from the return queues to the J-Bus transaction issue, and then to the J-Bus. Reads to the L2 cache may observe strict ordering with respect to writes to the L2 cache (software programmable)

A DMA write request from the J-Bus is parsed by the J-Bus parser. The information is passed to the WDQ, which sends the request header and data to *sctag* of the L2 cache. Writes to the L2 cache may observe strict ordering with respect to the other writes to the L2 cache (software programmable).

B.6.2 I/O Buffer Requests to the J-Bus

Write requests (NCWR) can be 1-, 2-, 4-, or 8-byte writes and those writes are aligned to size. Write requests come from the I/O buffer (IOB), are stored in the PIO request queue, and then go out on the J-Bus.

Read requests come from the IOB, are stored in the PIO request queue, and then go out on the J-Bus. The data read from J-Bus is parsed by the J-Bus parser and then stored in the PIO return queue, which is sent to the IOB.

The Read transactions (NCRD) can be 1-, 2-, 4-, 8-, or 16-byte reads and are aligned to size. There is maximum support for 1 to 4 pending reads to the J-Bus (software programmable). Read returns to the IOB may observe strict ordering with respect to the writes to the L2 cache (software programmable).

B.6.3 J-Bus Interrupt Requests to the IOB

Interrupts to the IOB may observe strict ordering with respect to the writes to the L2 cache (software programmable).

A J-Bus interrupt in the mondo vector format is received by the J-Bus parser and stored in the interrupt queue before being sent to the IOB. The mondo interrupt queue is maximally sized to 16 entries, and there is no flow control on the queue. A modified mondo interrupt transaction is one in which only the first data cycle is forwarded to the CPU.

An interrupt ACK/NACK received from the IOB is first stored in the interrupt ACK/NACK queue and then sent out on the J-Bus.

APPENDIX **C**

Overview of OpenSPARC T2 Design

This appendix adds more detail to the overview presented in Chapter 4. It is excerpted from *OpenSPARC T2 Core Microarchitecture Specification*, Part number 820-2545-11 December 2007, Revision A, and also from *OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification*, Part number 820-2620-05 July 2007, Revision 05. See those manuals for specific details about the material covered in this appendix. This appendix gives the flavor of those manuals in these sections:

Additional overview information from *OpenSPARC T2 System-On Chip (SoC) Microarchitecture Specification* is contained in the following sections:

C.1 OpenSPARC T2 Design and Features

Key design aspects of OpenSPARC T2 are the following:

- Full implementation of the SPARC V9 instruction set except for quad instructions including load/store; full implementation of the VIS 2.0 specification
- Support for eight threads
- Ability to sustain one FGU operation per thread every clock
- Two integer execution units (EXUs), one shared load-store unit (LSU), one shared floating-point and graphics unit (FGU)
- Eight-way, 16-Kbyte instruction cache, four-way, 8-Kbyte data cache
- These pipelines:
  - Eight-stage integer pipeline
  - Extended pipeline for long latency operations
  - Twelve-stage floating-point and graphics pipeline
- Instruction fetching of up to four instructions per cycle
- Data TLB of 128 entries, fully associative; instruction TLB of 64 entries, fully associative
- Hardware tablewalk support

OpenSPARC T2 consists of the following components and functionality

- **Instruction fetch unit (IFU)—** Provides instructions to the rest of the core. The IFU generates the Program Counter (PC) and maintains the instruction cache (I-cache).
- **Execution unit (EXU)** — Executes all integer arithmetic and logical operations except for integer multiplies and divides, calculates memory and branch addresses, and handles all integer source operand bypassing.
- **Load-store unit (LSU)** — Handles memory references between the SPARC core, the L1 data cache, and the L2 cache. All communication with the L2 cache is through the crossbars (processor-to-cache and cache-to-processor, a.k.a. PCX and CPX) via the gasket. All SPARC V9 and VIS 2.0 memory instructions are supported with the exception of quad-precision floating-point loads and stores.
- **Cache crossbar (CCX)** — Connects the eight SPARC cores to the eight banks of the L2 cache. An additional port connects the SPARC cores to the I/O bridge. A maximum of eight load/store requests from the cores and eight data returns/acks/invalidations from the L2 can be processed simultaneously.

- **Floating-point and graphics unit (FGU) —** Implements the SPARC V9 floating-point instruction set; the SPARC V9 integer multiply, divide, and population count (POPC) instructions; and the VIS 2.0 instruction set.
- **Trap logic unit (TLU) —** Manages exceptions, trap requests, and traps for the SPARC core. The TLU maintains processor state related to traps as well as the Program Counter (PC) and Next Program Counter (NPC).
- **Memory management unit (MMU) —** Reads translation storage buffers (TSBs) for the Translation Lookaside Buffers (TLBs) for the instruction and data caches. The MMU receives reload requests for the TLBs and uses its hardware tablewalk state machine to find valid translation table entries (TTEs) for the requested access. The TLBs use the TTEs to translate virtual addresses (VAs) and real addresses (RAs) into physical addresses (PAs). The TLBs also use the TTEs to validate that a request has the permission to access the requested address.
- **Reliability and serviceability (RAS) services —** The expected failure-in-time (FIT) rates of OpenSPARC T2 microarchitectural structures drive the RAS features.
- **ASI/ASR/HPR/PR access —** OpenSPARC T2 conceptually has ASI "rings"—fast, local, and global—to access registers defined in ASI space. These registers are accessed using Load and Store alternate instructions. Access to Ancillary State registers (ASRs), privileged registers (PRs), and hyperprivileged registers (HPRs) via RDasr/WRasr, RDPR/WRPR, and RDHPR/WRHPR instructions also occur over the ASI rings.
- **Reset operations —** Like previous SPARC processors, OpenSPARC T2 provides several flavors of resets. Resets can be activated by any of the following:
  - Side effect of an internal processor or system error, related either to instruction execution or to an external event such as failure of a system component
  - Result of explicit instruction execution (e.g., SIR);
  - Result of a processor write to an ASI register, which generates a reset
  - Command over an external bus, such as the system bus or the JTAG interface, to the test control unit (TCU)
  - Result of activating a pin on the OpenSPARC T2 chip

  Some resets are local to a given physical core or affect only one thread (CMP core). Other resets affect all threads.

- **Power management** — Hardware power management uses clock gating within functional units to reduce power consumed by flops, latches, and static arrays. Since the OpenSPARC T2 core is static, there is no dynamic logic to be power-managed. Hardware power management can be enabled by software.
- **Performance monitors —** Performance monitoring is aimed at the following:
  - Enabling data collection to develop accurate modeling for OpenSPARC T2 and future highly threaded processors
  - Enabling debug of performance issues
  - Minimizing hardware cost consistent with the above objectives
- **Debugging features —** OpenSPARC T2 hardware features for post-silicon debuggability involve debugging any issues that interfere with early bringup as well as debugging the difficult, complex bugs that eluded pre-silicon verification, are unexpected, or are unusual corner cases. These features make silicon debugging more efficient, shortening the time to discover the root cause of complex bugs and thereby reducing time to remove and replace such bugs.

C.2 SPARC Core

Each SPARC physical core is supported by system-on-chip (SoC) hardware components. For a broader look at the functioning units of the OpenSPARC T2 SOC, see the sections beginning below. For details about them, see *OpenSPARC T2 System-On Chip (SoC) Microarchitecture Specification*.

OpenSPARC T2 cores use 8-bit bytemask fields for L2 cache stores instead of the 2-bit size field that OpenSPARC T1 cores use. The main reason for this is to support VIS partial stores with random byte enables.

C.2.1 Instruction Fetch Unit (IFU)

The instruction fetch unit (IFU) contains three subunits: the fetch unit, pick unit, and decode unit.

The OpenSPARC T2 IFU differs from the OpenSPARC T1 IFU in the following ways:

- OpenSPARC T2 has added the B pipeline stage between the M and W stages, which in turn adds one early port to the rs1, rs2, rs3, and rcc flops above the E stage. The new port requires an additional set of source-destination comparators.
- The multiplier resides in the floating-point and graphics unit. The FGU, not the execution unit, executes integer multiplies. The EXU reads the integer register file (IRF) for integer multiplies as it does for all other integer instructions and forwards the operands to the FGU.
- OpenSPARC T2 does not have a dedicated integer divider. The FGU executes integer divides. The EXU reads the IRF for integer divides as it does for all other integer instructions and forwards the operands to the FGU.
- As part of VIS 2.0 support, the EXU executes edge instructions. The twelve forms of the EDGE instruction include EDGE8[L]N, EDGE16[L]N, EDGE32[L]N. These support both left and right edge, as well as big and little-endian. A 2-bit (EDGE32), 4-bit (EDGE16), and 8-bit (EDGE8) pixel mask is stored in the least significant bits of rd.
- As part of VIS 2.0 support, the EXU executes array instructions. These instructions convert three-dimensional (3D), fixed-point addresses contained in rs1 to a blocked-byte address and store the result in rd. These instructions specify an element size of 8 (ARRAY8), 16 (ARRAY16), and 32 bits (ARRAY32). The rs2 operand specifies the power-of-two size of the X and Y dimensions of a 3D image array.
- As part of VIS 2.0 support, the EXU executes the BMASK instruction. BMASK adds two integer registers, rs1 and rs2, and stores the result in rd. The least significant 32 bits of the result are stored in the GSR.mask field.
- An OpenSPARC T2 core contains two instances of the EXU. One instance supports Thread Group 0 (threads 0 through 3); the other supports Thread Group 1 (threads 4 through 7).
- In addition to the two EXUs, OpenSPARC T2 has a single load-store unit (LSU) and a single floating-point and graphics unit (FGU). The FGU executes the following integer instructions:
  - Integer multiply
  - Integer divide
  - Multiply step (MULSCC)
  - Population count (POPC)

C.2.2 Execution Unit

The EXU is composed of the following subunits:

- Arithmetic logic unit (ALU)
- Shifter (SHFT)
- Operand bypass (BYP): rs1, rs2, rs3, and rcc bypassing
- Integer register file (IRF)
- Register management logic (RML)

Differences between the T1 and T2 execution units are the same as for the instruction fetch unit.

C.2.3 Load-Store Unit (LSU)

The load-store unit (LSU) ensures compliance with the TSO memory model with the exception of instructions that are not required to strictly meet those requirements (block stores, for example). Like OpenSPARC T1, OpenSPARC T2 does not support an explicit RMO mode.

The LSU is responsible for handling all ASI operations including the decode of the ASI and initiating transactions on the ASI ring. The LSU is also responsible for detecting the majority of data access related exceptions.

C.2.3.1 Changes From OpenSPARC T1

- OpenSPARC T2 supports store pipelining. OpenSPARC T1 requires any store to receive an ACK before the next store could issue to the PCX.
- The store buffer supports eight threads.
- The load miss queue supports eight threads.
- The data Translation Lookaside Buffer (DTLB) is 128 entries.
- Only load operations access the D-cache from the pipeline.This reduces conflicts with the CPQ.
- Partial Store instructions from the VIS 2.0 ISA are supported.
- Pipeline is E/M/B/W vs. OpenSPARC T1's E/M/W/W2. Pipeline timings are different from those of OpenSPARC T2.
- OpenSPARC T2 has additional RAS features and enhanced error detection and protection.

C.2.3.2 Functional Units of the LSU

The functional units of the LSU are the following:

- **Data cache** — An 8-Kbyte, 4-way set-associative cache with 16-byte lines. The DCA array stores data, the DTAG array stores tags, the DVA array stores valid bits, and the LRU array stores used bits.

- **Data Translation Lookaside Buffer (DTLB) —** Specified in the MMU specification but located in the LSU because of its physical proximity and close linkage with the D-cache and store buffers.
- **Load miss queue (LMQ) —** Contains loads that have missed the D-cache and are waiting on load return data from the L2 or NCU. All internal ASI loads are also placed into the LMQ. The LMQ also holds loads which RAW in the store buffer while they wait for resolution.
- **Store buffer (STB) —** Holds all store instructions and instructions that have store semantics (atomics, WRSR, WRPR, WRHPR). These are inserted into the STB after address translation through the DTLB, assuming the stores do not generate an exception. The STB is threaded and contains eight entries per thread.
- **PCX interface (PCXIF) —** In conjunction with the LMQ and STB, arbitrates between load and store requests and manages outgoing packets. The arbitration between loads and stores is done as follows. The goal is to minimize load-miss latencies while avoiding store-buffer-full occurrences. To achieve that, a weighted favor system is used.
  - Loads are favored over stores by default.
  - If a store has been waiting for four cycles, it is favored.
  - If any thread's store buffer is full, stores have favor every other cycle.
- **CPX interface (CPXIF) —** Monitors all packets from the CPX. Packets destined for units other than the LSU are ignored by this interface. The CPXIF receives all CPX packets that affect the D-cache. This includes load returns, store acknowledgments, and invalidation requests.

C.2.3.3 Special Memory Operation Handling

Special memory operations are those that do not conform to the standard pipeline or that require additional functionality beyond standard loads and stores. The following are such instructions:

- CASA and CASXA — Compare and Swap instructions have load and store semantics. The value in `rs2` is compared with the value in memory at `rs1`. If the values are the same, the value in memory is swapped with the value in `rd`. If the values are not the same, the value at `rs1` is loaded into `rd`, but memory is not updated.
- LDSTUB, LDSTUBA, and SWAP — Load and Store Unsigned Byte and Swap instructions have load and store semantics. A byte from memory is loaded into `rd` and the memory value replaced with either all 1's (for LDSTUB) or the value from `rd` (for SWAP).

- Atomic quad loads — Atomic quad load instructions load 128 bits of data into an even/odd register pair. Since there is no path to bypass 128 bits of data to the IRF, atomic quads force a miss in the L1 cache. One 128-bit load request is made to the L2 cache. The return data is written to the IRF over two cycles, once for the lower 64 bits and once for the upper 64 bits. Load completion is signaled on the second write. The load does not allocate in the L1 cache.
- Block loads and stores — Block loads and stores are loads and stores of 64 bytes of data. Memory ordering of block operations is not enforced by the hardware—that is, they are RMO. A block load requires a MEMBAR `#Sync` before it to order against previous stores. A block store requires a MEMBAR `#Sync` after it to order against following loads. Block loads and stores force a miss in the L1 cache and they do not allocate.
- FLUSH — The IFU postsyncs FLUSH instructions, so no LSU synchronization is necessary. Once all stores prior to the FLUSH instruction have been committed, which implies all previous stores have been acknowledged and necessary invalidations performed, the LSU signals the TLU to redirect the thread to the instruction following the FLUSH via a trap sync.

  Because hardware enforces I-cache/D-cache exclusivity, any stores to an address in the I-cache are automatically invalidated. Therefore, the FLUSH instruction doesn't actually do anything to the caches. It acts solely as a synchronization point, much like MEMBAR.
- MEMBAR — MEMBAR (all forms) and STBAR are all executed identically. MEMBAR instructions behave identically to FLUSH. The IFU postsyncs the instruction, so no LSU synchronization is required. Once all stores for that thread have been committed, the LSU signals the TLU through a trap sync to redirect the thread to the instruction following the MEMBAR.
- PREFETCH — Prefetch instructions load data into the L2 cache but do not update the L1 caches. When the LSU receives a prefetch instruction, it signals LSU synchronization to the IFU and inserts the entry into the LMQ. A load request packet is sent to the L2 with the prefetch indicator asserted. Once the packet is sent to the PCX, lsu_complete can be signaled and the entry in the LMQ retired. The L2 does not return any data.

  Except when used with illegal function codes, PREFETCH instructions do not cause exceptions, including MMU miss exceptions. If the PREFETCH encounters an exception condition, it is dropped.

C.3 L2 Cache

The L2 cache accepts requests from the SPARC cores on the processor to the cache crossbar (PCX) and responds on the cache to the processor crossbar (CPX). The L2 cache also maintains on-chip coherency across all L1 caches on the chip by keeping a copy of all L1 tags in a directory structure. Since OpenSPARC T2 implements system-on-a-chip with single memory interface and no L3 cache, there is no off-chip coherency requirement for the L2 cache other than being coherent with main memory. Other L2 features:

- The L2 cache is a write-back cache and has lines in one of three states: invalid, clean, or dirty.
- Each L2 bank has a 128-bit fill interface and a 64-bit write interface with the DRAM controller.
- Requests arriving on the PCI-EX/Enet interface are sent to the L2 from the system interface unit (SIU).
- The L2 cache unit works at the same frequency as the core (1.4 GHz).

C.3.1 L2 Functional Units

The L2 cache is organized into eight identical banks, each with its own interface with the SIU, memory controller unit (MCU), and crossbar. Each L2 cache bank interfaces with the eight cores through a processor cache crossbar. The crossbar routes the L2 request (loads, ifetches, stores, atomics, ASI accesses) from all eight cores to the appropriate L2 bank. The crossbar also accepts read return data, invalidation packets, and store acknowledgment packets from each L2 bank and forwards them to the appropriate core(s).

Every two L2 cache banks interface with one MCU to issue reads and evictions to DRAM on misses in the L2. Write-backs are issued 64 bits at a time to MCU. Fills happen 128 bits at a time from the MCU to L2.

For 64-byte I/O writes from the SIU, L2 does not allocate, but issues the writes to DRAM through a 64-bit interface with the MCU. Round-robin arbitration is used between the write-back buffer and the I/O write buffer for access to the MCU.

Each L2 cache bank also accepts RDD (read to discard), WRI (block write invalidate) and WR8 (partial write with random byte enables) packets from SIU over a 32-bit interface and queues the packet in the SIU queue. RDD and WRI do not allocate in the L2. On a hit, WRI invalidates in the L2 and issues a 64-byte block write to DRAM. On a hit, RDD gets back 64 bytes of data from L2. On a miss, RDD fetches data from DRAM but does not install in the L2, while WRI (on a miss) issues a 64-byte block write to DRAM. WR8 packets cause partial stores to happen in the L2 like regular CPU stores with random byte enables.

Each L2 cache bank is composed of the following subblocks:

- **Input queue** (IQ – 16-entry FIFO**)** — Queues packets arriving on the PCX when they cannot be immediately accepted into the L2 pipe. Each entry in the IQ is 130 bits wide.
- **Output queue** (OQ – 16-entry FIFO) — Queues operations waiting for access to the CPX. Each entry in the OQ is 146 bits wide. The FIFO is implemented with a dual-ported array.
- **SIU queue (SIUQ)** — Accepts RDD, WRI, and WR8 packets from the SIU and issues them to the pipe after arbitrating against other requests.

The following are other functional units of the L2 cache:

- **Arbiter** — Manages access to the L2 pipeline from the various sources that request access. The IQ, MB, SIUQ, FB and stalled instructions in the pipe all need access to the L2 pipe. Access to the pipe is granted according to the following priority:
  1. Access currently stalled in the pipe
  2. Second packet of a CAS operation
  3. SIU instruction from SIU queue
  4. Miss buffer instruction
  5. Fill buffer instruction
  6. Instruction from the IQ
  7. Background scrub request
- **L2 tag** — Holds the L2 tag array and associated control logic. Tag is protected by SEC ECC.
- **L2 VUAD** — Contains the Valid, Dirty, Used, and Allocated bits for the tags in L2 organized in an array structure. There is one array for Valid and Dirty bits and a separate array for Used and Allocate bits. Each array is protected by SEC DED ECC.
- **L2 data** — Contains 512 Kbytes of L2 data storage and associated control logic. Data is protected by SEC DED ECC on a 32/7 boundary.

  Each L2 data array bank is a single-ported SRAM structure capable of performing the following operations:
  - 16-byte read
  - 64-byte read
  - 8-byte write with any combination of word enables
  - 64-byte write (with any combination of word enables). However, fills would update all 64 bytes at a time.

- **L2 directory** — Maintains a copy of the L1 tags for coherency management and also ensures that the same line is not resident in both the I-cache and dcache (across all cores). The directory is split into an I-cache directory (`icdir`) and a dcache directory (`dcdir`), which are similar in size and functionality.
- **Miss buffer** (MB – 32-entry) — Stores instructions that cannot be processed as a simple cache hit. Includes the following:
  - True L2 cache misses (no tag match)
  - Instructions that have the same cache line address as a previous miss or an entry in the write-back buffer
  - Instructions requiring multiple passes through the L2 pipeline (atomics and partial stores)
  - Unallocated L2 misses
  - Accesses causing tag ECC errors
- **Fill buffer** (8-entry) — Temporarily stores data arriving from DRAM on an L2 miss request. Data arrives from DRAM in four 16-byte quad-words starting with the critical quad-word.
- **Write-back buffer** (WBB – 8-entry) — Stores dirty evicted data from the L2 on a miss. Evicted lines are opportunistically. streamed out to DRAM. The WBB is divided into a RAM portion that stores the evicted data until it can be written to DRAM and a CAM portion that contains the address.

  I/O write buffer (IOWB – 4-entry) — Stores incoming data from the PCI-EX interface in the case of a 64-byte write operation. Since the PCI-EX interface bus width is only 32 bits wide, the data must be collected over 16 cycles before being written to DRAM. The IOWB is divided into a RAM portion that stores the data from the I/O interface until it can be written to DRAM and a CAM portion that contains the address.

  The I/O interface must use a handshaking protocol to track the state of the IOWB. The I/O interface must never issue an operation requiring the buffer when the buffer is full.

C.3.2 L2 Cache Interfaces

The L2 cache interfaces with the following:

- **Crossbar** — The L2 cache receives requests from the core through the crossbar. These requests are received, decoded and forwarded to the arbiter logic by the Input queue (IQ), depending on the status of the arbiter block.

- **SIU** — Requests from I/O's (PCI Express and Ethernet) are received by the L2 cache through the SIU queue block. Three kinds of requests can be received from the SIU: RDD (read 64 bytes), WRI (write 64 bytes) and WR8 (write 8 bytes).
- **MCU** — L2 cache issues read and write requests to the MCU. All instructions that do not hit in the L2 cache are recorded in the miss buffer (MB). The MB evaluates and sets a bit (dram_pick) if it needs to be issued to the MCU. Reads that must be dispatched to the MCU should satisfy the following criteria:
  - Win arbitration among all pending reads (with the dram_pick bit set for reads).
  - Have no pending (read or write) transactions to MCU waiting for an ack.
  - Have enough place in the fill buffer for the read data to return.

  An L2 cache bank can send one read or write request at a time to the MCU. Once a bank sends a request, it must wait for the appropriate acknowledge before sending the next request. Three cycles must elapse from the completion of a transaction (acknowledge for a read, last data word for a write) until the next request can be made. A total of eight outstanding read requests and eight outstanding write requests can be outstanding from each L2 cache bank at any time.

C.3.3 L2 Cache Instructions

Some representative L2 operations are briefly described below.

- **Load instructions** — Loads always return 16 bytes of data, and lower address bits are ignored. The 8-bit bytemask field is ignored for loads.

  The instruction types that fall in the category of loads are load, prefetch, stream load, MMU load. Out of these, prefetch, stream load, and MMU load are noncacheable (will have the nc bit set in the PCX packet). These loads do not cam the `I$` directory and do not update the `D$` directory.
- **Store instructions** — Eight bytes of store data are always sent to the L2. The LSU ensures that data is properly aligned to the 8B boundary. The 8-bit bytemask indicates which bytes are to be stored. Again, the lower address bits are ignored. (This is different from OpenSPARC T1. OpenSPARC T1 L2 used the lower address bits along with the size to determine what to store.)

  The instruction types that fall in the category of stores are stores and stream stores.

To improve the performance of stores from L1, the L2 cache in OpenSPARC T2 sends acks to core if stores from an L1 hit to outstanding store miss to the same line in the miss buffer. If any of the addresses it hits is a load miss, the ack is not generated.

- **Partial store instructions** — Partial stores are stores (stores, stream stores) that have any combination of byte masks other than 0000 1111, 1111 0000, and 1111 1111.

  Even for partial stores, eight bytes of store data are always sent to the L2. The LSU will ensure that the data is properly aligned to the 8-byte boundary. The 8-bit bytemask indicates which bytes are to be stored. Again, the lower address bits (0, 1, 2) are ignored.

  Partial stores are handled as a read-modify-write operation in two passes through the pipe.

- **Instruction miss** — An instruction that does not hit the L2 cache, fill buffer, or the write-back buffer is queued in the miss buffer as a "true miss." Eviction is performed during the second pass of the miss operation. This removes the hit/miss determination from the critical C1 stall signal. To improve the performance of stores from L1, the L2 cache in OpenSPARC T2 sends acks to core in case stores from L1 hit to outstanding store miss to the same line in the miss buffer.

- **Atomic instructions** — LDSTUB and SWAP are handled the same as loads. CAS[X]A instructions are handled as two packets. The first packet (CAS(1)) reads the data from memory, sends the data back to the requesting processor, and performs the comparison in C8 of the pipeline. The second packet (CAS(2)) is inserted into the miss buffer as a store. If the comparison result is true, the second packet proceeds like a normal store. If the result was false, the second pass proceeds to only generate the store acknowledgment. The data arrays are not written.

  CASA/CASXA are similar, but with one difference. CASA is 32 bits, aligned on a 4-byte boundary, and CASXA is 64 bits. The compare and conditional store are assumed to be on an 8-byte boundary (except the load return, which is always 16 bytes). The 8-bit bytemask indicates which bytes to compare and conditionally store.

  Prefetch ICE — L2 supports the Prefetch ICE instruction that software uses to flush lines in L2, based on an index and a way specified as part of the physical address in the instruction itself.

C.4 Cache Crossbar

The cache crossbar is divided into two separate pieces: the processor-to-cache crossbar (PCX) and the cache-to-processor crossbar (CPX). Sources issue requests to the crossbar. These requests are queued to prevent head of the line blocking. The crossbar queues requests and data to the different targets.

Since multiple sources can request access to the same target, arbitration within the crossbar is required. Priority is given to the oldest requestor(s) to maintain fairness and ordering. Requests appearing to the same target from multiple sources in the same cycle are processed in a manner that does not consistently favor one source.

The arbitration requirements of the PCX and CPX are identical except for the numbers of sources and targets that must be handled. The CPX must also be able to handle multicast transactions. To facilitate reuse, the arbitration logic is designed as a superset that can handle PCX or CPX functionality. The arbiter performs the following functions:

- Queues transactions from each source to a depth of 2.
- Issues grants in age order, with oldest having highest priority.
- Resolves requests of the same age without persistent bias to any one source.
- Can stall grants based on input from the target.
- Stalls the source if the queue is full.
- Handles two packet transactions atomically.

Each target has its own arbiter. All targets can arbitrate independently and simultaneously.

C.5 Memory Controller Unit

Section 5.2, *Memory Controller Unit (MCU)*, on page 47 neatly summarizes the main features and functions of the OpenSPARC T2 MCU. This section adds information about the following:

- *Changes to the OpenSPARC T2 MCU* on page 281
- *DDR Branch Configuration* on page 282
- *FBD Channel Configuration* on page 282
- *SDRAM Initialization* on page 282

C.5.1 Changes to the OpenSPARC T2 MCU

Some changes to the OpenSPARC T2 MCU are listed below.

- The OpenSPARC T2 MCU differs from the OpenSPARC T1 MCU in these respects:
  - Uses higher DDR2 SDRAM frequency: 266 MHz, 333 MHz, and 400 MHz instead of 166 MHz to 200 MHz.
  - Uses FBD channels to access memories instead of direct DDR2 interface.
  - Interfaces to two L2 cache banks per MCU instead of one or two L2 Cache banks interface per MCU.
  - Has minimum configuration with one DIMM per MCU branch.
- Several changes to the OpenSPARC T2 MCU were made in support of FBDs:
  - A new FBD controller with channel initialization, error detection, and frame encode and decode logic is added.
  - Address decoding is updated to support up to 16 ranks of DIMMs. Can support either one or two channels per MCU.
  - Write data rate is reduced to half the DDR rate. Data is buffered in the advanced memory buffer (AMB) to allow more flexibility in issuing write commands.
  - Read and write operations to different DIMMs can occur in parallel. Reads and writes to a single FBD must be scheduled so that no data collisions occur on the DIMM's local DDR2 bus. However, since the northbound and southbound channels are independent, read data from one DIMM can be returning to the host at the same time that write data is being sent to a different DIMM.
  - Separate read and write schedulers communicate with each other to ensure that no FBD bus data collisions occur.
  - No dead cycle occurs when read or write commands are switched between DIMMs; however, a dead cycle is still needed when access is switched to the other sides of same DIMM.
  - Sync frame generation to AMBs in the state machine—at least once every 42 frames—is included.
  - Support for read DQS strobe placement is removed. OCD and ODT support is programmed through the AMBs.
  - Transactions are spread over different DIMMs instead of staying in one DIMM as long as possible. That way, thermal dissipation is better spread across DIMMs.
  - L0s power-saving mode is supported.

C.5.2 DDR Branch Configuration

A DDR branch is a minimum aggregation of DDR channels (data channels with 72 bits of data and an address/control channel) that operate in lock-step to support error correction. A rank spans a branch. In OpenSPARC T2, a branch consists of one or two DDR channels.

C.5.3 FBD Channel Configuration

A channel is a port that connects the processor to a DIMM. Northbound is the direction of signals running from the farthest DIMM toward the host. Southbound is the direction of signals running from the host controller toward the DIMMs.

The FBD specification supports two southbound channel configurations and five northbound channel configurations. OpenSPARC T2 supports both southbound configurations—the 10-bit mode and the 10-bit failover mode—and two of the northbound configurations, the 14-bit mode and the 14-bit failover mode. These modes support data packets of 64 bits of data and 8 bits of ECC. The 10-bit southbound mode provides 22 bits of CRC; the 10-bit failover mode has 10 bits of CRC. The 14-bit northbound mode provides 24 bits of CRC on read data (12-bits per 72-bit data packet), and the 14-bit failover mode provides 12 bits of CRC (6 bits per 72-bit data packet).

During channel initialization, software determines whether a channel can be fully utilized (10-bit southbound or 14-bit northbound mode) or whether a failover mode, in which one of the bit lanes is multiplexed out, must be used.

Data is transmitted across the southbound and northbound channels in frames. For the southbound channel, 10 bits of data are sent per cycle over 12 cycles. For the northbound channel, 14 bits of data are sent per cycle over 12 cycles.

C.5.4 SDRAM Initialization

The initialization sequence within the MCU for the SDRAMs follows the same flow as for UltraSPARC T1, which is basically the sequence from the DDR2 SDRAM specification. However, the interface is different. The MCU initializes the SDRAMs indirectly through registers in the AMBs. The MCU issues a command to the AMBs and then polls status registers to determine when the AMBs have completed issuing the command to the SDRAMs.

After SDRAM initialization is complete, the MCU begins scheduling commands directly to the SDRAMs.

The DDR2 SDRAMs must be powered up and initialized in a predefined manner. Operational procedures other than those specified may result in undefined operation. Below is the mandatory sequence:

1. Apply power to the VDD.
2. Apply power to VDDQ.
3. Apply power to VREF and to the system VTT.
4. Start clock and maintain stable condition for 200 s.
5. Apply No Operation or Deselect command and take CKE high.
6. Wait minimum of 400 ns, then issue a Precharge-all command.
7. Issue Extended Mode Register 2 Set (EMRS(2)) command.
8. Issue Extended Mode Register 3 Set (EMRS(3)) command.
9. Issue Extended Mode Register 1 Set (EMRS(1)) command to enable DLL.
10. Issue Mode Register Set (MRS) command to reset DLL.
11. Issue Precharge-all command.
12. Issue two or more Auto-Refresh commands.
13. Issue MRS command with low on A8 to initialize device operation (i.e., to program operating parameters without resetting the DLL).
14. At least 200 clocks after step 8, execute OCD Calibration (Off Chip Driver Impedance adjustment). If OCD calibration is not used, EMRS OCD Default command (A9=A8=A7=1) followed by EMRS OCD Calibration Mode Exit command (A9=A8=A7=0) must be issued with other parameters of EMRS.
15. The DDR2 SDRAM is now ready for normal operation.

C.6 Noncacheable Unit (NCU)

One of the main functions of the noncacheable unit (NCU) is to communicate between the CMP cores (64 threads total) and the various blocks in the I/O subsystem. FIGURE 11-5 shows the connectivity of NCU with various IO subsystem blocks as well as the XBAR, which connects to the CMP core on the other side. Traffic on the XBAR side runs at CPU clock frequency, whereas traffic on the I/O subsystem side is at I/O clock frequency. In general, traffic going to the NCU does not require high performance and can tolerate high latency.

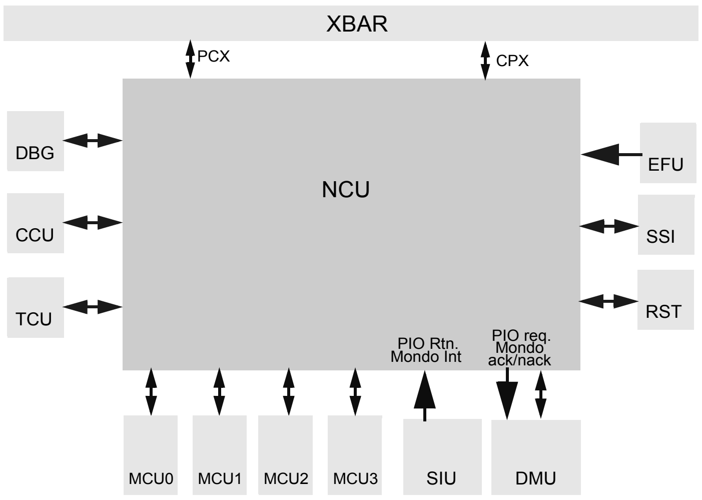


FIGURE 11-5 Connectivity of the Noncacheable Unit

C.6.1 Changes from OpenSPARC T1 I/O Bridge

- Changed from two MCUs to four MCUs
- CTU changes made to CCU + TCU
- J-Bus changes made to SIU + DMU with different interface format
- DMU CSR support added
- DMU PIO token ID engine added to limit numbers of outstanding PIO to DMU
- DMU PIO memory added because of OpenSPARC T2 I/OMMU changes
- Support added for Mondo Interrupt ID return for DMU
- ASI register added to comply with Sun's CMP specification
- L2 partial bank mode support added
- Internal memories upsized to accommodate 64 threads and memory pipelines adjustment
- XBAR packet format changes made
- Reset handling modified to comply with OpenSPARC T2's reset scheme
- SSI (boot ROM i/f logics) integrated
- RAS logics added
- OpenSPARC T2 naming rule complied with

Note The NCU retains most of the internal block names from OpenSPARC T1 IOB since it does not violate the OpenSPARC T2 naming rules.

C.6.2 NCU Interfaces

- **NCU-MCU** — The four MCUs on OpenSPARC T2 are all connected to the NCU in the same manner. The downstream and upstream paths are both 4-bit-wide data buses with two control signals. The interface protocol is a 128-bit packet spread out over 32 cycles of transactions. The NCU sends only types READ_REQ and WRITE_REQ with 8-byte request size to the MCU for CSR access. The MCU sends the following packet types upstream to NCU:
  - READ_ACK, with 8-byte payload in response to a successful READ_REQ (128-bit UCB packet)
  - READ_NACK, without payload in response to an unsuccessful READ_REQ (64-bit UCB packet without payload)
  - INT, for on-chip interrupt, resulting from some error conditions in the MCU (64-bit UCB Int. packet with dev_id = 1)
- **NCU-SSI (Boot ROM Interface)** — The NCU has integrated the SSI interface logics that originated from OpenSPARC T1. Here are some of the OpenSPARC T2 differences from OpenSPARC T1:
  - The ncu_mio_ssi_sck frequency is now programmable.
  - Four I/O pins connect directly connect to the external pins.
  - The original SSI UCB interface has become NCU internal signals and is no longer visible from outside the NCU cluster.
  - ncu_mio_ssi_sck could be programmed as iol2clk/8 or iol2clk/4, depending on the CSR register NCU_SCKSEL. This register is warm_reset protected. The new value programmed into NCU_SCKSEL register cannot affect the current ncu_mio_ssi_sck until the next warm reset.

C.7 Floating-Point and Graphics Unit (FGU)

The OpenSPARC T2 floating-point and graphics unit (FGU) implements the SPARC V9 floating-point instruction set, the SPARC V9 integer multiply, divide, and population count (POPC) instructions, and the VIS 2.0 instruction set, with the following exception: All quad-precision floating-point instructions are unimplemented (including LDQF[A] and STQF[A]). FIGURE 11-6 illustrates the functional block of the FGU.

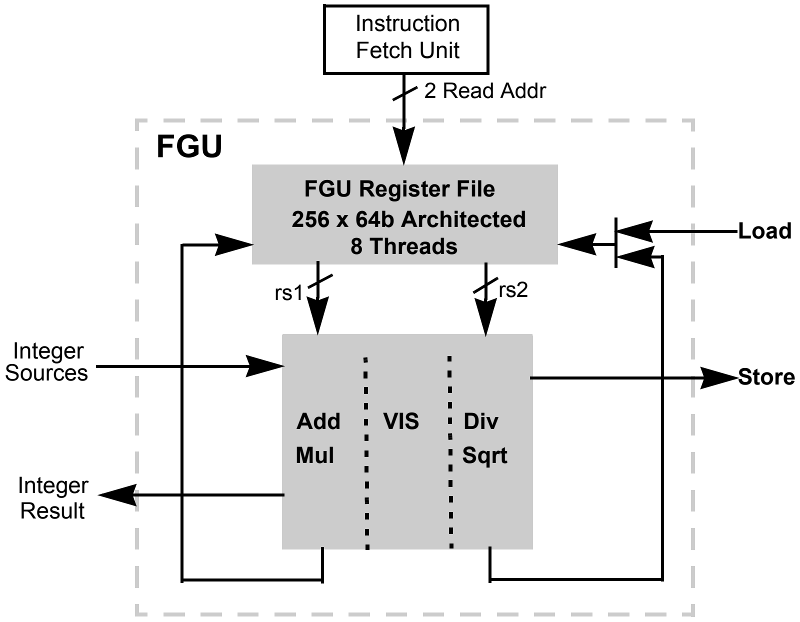


FIGURE 11-6 Floating-Point and Graphics Unit

Following are OpenSPARC T2 features of note for the FGU:

- Contains one dedicated FGU per core.
- Complies with the IEEE 754 standard.
- Supports IEEE 754 single-precision (SP) and double-precision (DP) data formats. All quad precision floating-point operations are unimplemented.
- Supports all IEEE 754 floating-point data types (normalized, denormalized, NaN, zero, infinity). Certain denormalized operands or expected results may generate an unfinished_FPop trap to software, indicating that the FGU was unable to generate the correct results. The conditions that generate an unfinished_FPop trap are consistent with UltraSPARC I/II.
- Includes three execution pipelines:
  - Floating-point execution pipeline (FPX)
  - Graphics execution pipeline (FGX)
  - Floating-point divide and square root pipeline (FPD)
- Up to one instruction per cycle can be issued to the FGU. Instructions for a given thread are executed in order. Floating-point register file (FRF) bypassing is supported for FGU results having a floating-point register file destination (excluding FDIV/FSQRT results).

- Maintains a precise exception model. The FGU uses an exception prediction technique to support full floating-point, single-thread pipelining, independent of IEEE trap enables. FGU operations are also pipelined across threads. A maximum of two FGU instructions (from different threads) may write back into the FRF in a given cycle (one FPX/FGX result and one FPD result).
- Has a 256-entry × 64-bit floating-point register file (FRF) with two read and two write ports.
  - FRF supports eight-way multithreading (eight threads) by dedicating 32 entries for each thread. Each register file entry also includes 14 bits of ECC for a total of 78 bits per entry. Correctable ECC errors (CEs), and uncorrectable ECC errors (UEs) result in a trap if the corresponding enables are set. CEs are never corrected by hardware but may be corrected by software following a trap.
  - One FRF write port (W2) is dedicated to floating-point loads and FPD floating-point results. FPD results always have highest priority for W2 and are not required to arbitrate. The other FRF write port (W1) is dedicated to FPX and FGX results. Arbitration is not necessary for the FPX/FGX write port because of single instruction issue and fixed execution latency constraints. FPX, FGX, and FPD pipelines never stall.
  - To avoid stalling FPX or FGX, integer multiply, MULSCC, pixel compare and POPC results are guaranteed access to the integer register file (IRF) write port by the IFU.
  - Floating-point store instructions share an FRF read port with the execution pipelines.
- Focuses FGU pipelines on area and power reduction:
  - Merges floating-point and VIS datapaths where possible (partitioned add/subtract, partitioned compare, 8x16 multiply)
  - Merges floating-point add, multiply, and divide datapaths where possible (format, exponent)
  - For integer multiply and divide implementations, utilizes the respective floating-point datapaths
  - For POPC implementation, leverages the PDIST datapath
  - Simplifies floating-point adder pipeline (no independent LED/SED organization, no dedicated i2f prenormalization)
  - Eliminates OpenSPARC T1 denormalized operand and result handling
  - No floating-point quad precision support
  - Clock gating strategy for dynamic power management

All FGU-executed instructions have the following characteristics:

- Fully pipelined, single pass.
- Single-cycle throughput.
- Fixed six-cycle execution latency, independent of operand values, with the exception of

  – floating-point and integer divides and
  – floating-point square root.

  Divide and square root are not pipelined but execute in a dedicated datapath, and are nonblocking with respect to FPX and FGX. Floating-point divide and square root have a fixed latency. Integer divide has a variable latency, dependent on operand values.
- Pixel distance (PDIST)

  PDIST is a three-source instruction and requires two cycles to read the sources from the FRF, which has only two read ports. No FGU executed instruction may be issued the cycle after PDIST is issued. PDIST has a fixed six-cycle execution latency and a throughput of one instruction every two cycles.

- Complex instruction helpers are not used in the OpenSPARC T2 design. Some UltraSPARC implementations use helpers to support instructions such as pixel distance (PDIST) and floating-point block loads and stores.
- FPX uses a parallel normalize/round organization, eliminating the serial delay of a post-normalizer followed by a post-normalization increment by performing the normalization and round function in parallel.
- Execution pipelines are multi-precision in that SP scalar, DP scalar, VIS/integer scalar and VIS/integer SIMD values are stored in the FRF and interpreted by the execution pipeline as unique formats.
- Floating-point State register (FSR) for IEEE control and status.
- Graphics Status register (GSR) for VIS control.
- Underflow tininess is detected before rounding. Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded (inexact condition).
- IEEE exception and nonstandard mode support (FSR.ns = 1) are consistent with UltraSPARC I/II.

C.7.1 FGU Feature Comparison of OpenSPARC T2 and OpenSPARC T1

TABLE C-1 compares the FGU features in OpenSPARC T2 with similar features in OpenSPARC T1.

TABLE C-1 OpenSPARC T2 FGU Feature Summary

Feature	OpenSPARC T2	OpenSPARC T1
ISA, VIS	SPARC V9, VIS 2.0	SPARC V9, subset of VIS 2.0
Core multithreading	8 threads	4 threads
Core issue	1	1
Out-of-order execution (per thread)	No	No
FGU instantiations	1 per core	1 per chip
FGU issue	1	1
FGU architected register file	256 × 64 b for 8 threads + 14 b ECC per entry 2R/2W ports	128 × 64 b for 4 threads + 14 b ECC per entry 1 port
FSQRT implemented	Yes	No
IMUL/IDIV execute in FGU	Yes	No
POPC executes in FGU	Yes	No
Number of instructions executed in FGU	129	23
Execution latency		
FADD, FSUB	6	4
FCMP	6	4
FP/integer convert types	6	4 or 5
FMOV, FABS, FNEG	6	1
FMULs	6 (1/1 throughput)	7 (1/2 throughput)
FGU IMUL, IMULScc	5 (1/1 throughput)	Not applicable
FDIV	19 SP, 33 DP	32 SP, 61 DP
FGU IDIV	12–41	Not applicable
FSQRT	19 SP, 33 DP	Unimplemented
FMUL	6 (1/1 throughput)	Not applicable
FGU FPADD, FPSUB	6	Not applicable
PDIST	6 (1/2 throughput)	Not applicable
FGU VIS other	6	Not applicable
FGU POPC	6	Not applicable

TABLE C-1 OpenSPARC T2 FGU Feature Summary *(Continued)*

Feature	**OpenSPARC T2**	**OpenSPARC T1**
Single thread throughput		
FADD, FSUB	1/1	1/27
FMUL	1/1	1/30
FDIV/FSQRT/IDIV blocking	No	No
Hardware quad implemented	No	No
Full hardware denorm implemented	No	Yes

C.7.2 Performance

While the OpenSPARC T1 to OpenSPARC T2 microarchitecture evolution offers many performance enhancements, in some cases performance may decrease.

- Certain denormalized operands or expected results may generate an unfinished_FPop trap to software on OpenSPARC T2 (for details, see Section 7.10.4.5 in the *OpenSPARC T2 Core Microarchitecture Specification*). Unlike on other UltraSPARC implementations, a denormalized operand or result never generates an unfinished_FPop trap to software on T1.
- A small set of floating-point and VIS instructions are executed by the OpenSPARC T1 SPARC core FFU (not the off-core FPU). These include FMOV, FABS, FNEG, partitioned add/subtract, FALIGNDATA, and logical instructions. The OpenSPARC T2 instruction latency is equivalent to or less than OpenSPARC T1 for these instructions.

C.7.3 FGU Interfaces

The FGU interfaces with the units shown in FIGURE 11-7 as described below:

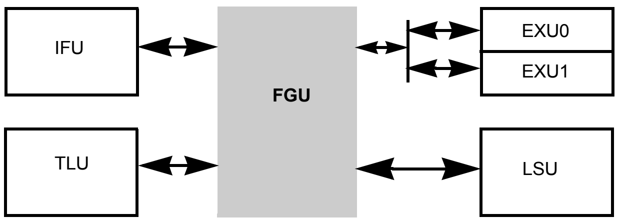


FIGURE 11-7 FGU Interfaces

- **Instruction fetch unit (IFU)** — The IFU provides instruction control information as well as rs1, rs2, and rd register address information. It can issue up to one instruction per cycle to the FGU.

  The IFU does the following:

  - Sends the following flush signals to the FGU:
    – Flush execution pipeline stage FX2 (transmitted during FX1/M stage)
    – Flush execution pipeline stage FX3 (transmitted during FX2/B stage)
  - Maintains copies of fcc for each thread.
  - Provides a single FMOV valid bit to the FGU indicating whether the appropriate icc, xcc, fcc, or ireg condition is true or false.

  Correspondingly, the FGU does the following:

  - Flushes the FPD based on the IFU- and trap logic unit (TLU)-initiated flush signals. Once an FPD instruction has executed beyond FX3, it cannot be flushed by an IFU- or TLU-initiated flush.
  - Provides appropriate FSR.fcc information to the IFU during FX2 and FX3 (including load FSR). The information includes a valid bit, the fcc data, and thread ID (TID) and is non-speculative.
  - Provides the FPRS.fef bit to the IFU for each TID (used by the IFU to determine *fp_disable*).

- **Trap logic unit (TLU)** — The FGU provides the following trap information to the TLU:

  - unfinished_FPop
  - *fp_exception_ieee_754*
  - *fp_cecc* (FRF correctable ECC error)
  - *fp_uecc* (FRF uncorrectable ECC error)
  - *division_by_zero* (integer).
  - Exception trap prediction

  The FGU receives the following flush signal from the TLU:

  - Flush execution pipeline stage FX3 (transmitted during FX2/B stage)

- **Load-store unit (LSU)** — Floating-point load instructions share an FRF write port with FPD floating-point results, which always have priority for the shared write port. FPD notifies the IFU and LSU when a divide or square root is near completion to guarantee that load data does not collide with the FPD result. Loads update the FRF or FSR directly, without proceeding down the execution pipeline. Load FSR is a serializing operation for a given thread (all previous FPops have completed, then load FSR completes prior to issuing subsequent FPops).

The LSU always delivers 32-bit load data replicated on both the upper (even) and lower (odd) 32-bit halves of the 64-bit load data bus. ECC information is generated by the FGU prior to updating the FRF.

Floating-point store instructions share an FRF read port with the execution pipelines. Store FSR is a serializing operation for a given thread (all previous FPops have completed, then store FSR completes prior to issuing subsequent FPops).

The FGU always delivers 32-bit store data on the upper (even) 32 bits of the 64-bit store data bus. The lower (odd) 32 bits are undefined. FGU delivers FRF ECC UE/CE information to the LSU one cycle after the data.

The FGU does not perform any byte swapping based on endianness (handled by LSU) or load data alignment for 32-, 16-, and 8-bit loads (also handled by LSU).

- **Execution Units** — Each EXU can generate the two 64-bit source operands needed by the integer multiply, divide, POPC, SAVE, and RESTORE instructions. The EXUs provide the appropriate sign-extended immediate data for rs2; provide rs1 and rs2 sign extension; and provide zero fill formatting as required. The IFU provides a destination address (rd), which the FGU provides to the EXUs upon instruction completion.

  The architected Y register for each thread is maintained within the EXUs. MULScc and 32-bit IMUL instructions write the Y register. MULScc and 32-bit IDIV instructions read the Y register.

  Each EXU provides GSR.mask and GSR.align fields, individual valid bits for those fields, and the TID.

  The FGU provides a single 64-bit result bus, along with appropriate icc and xcc information. The same result bus provides appropriate 64-bit formatted "gcc" information to the EXUs upon completion of the VIS FCMP (pixel compare) instructions. The result information includes a valid bit, TID, and destination address (rd). FGU clears the valid bit under the following conditions:

  - *division_by_zero* trap (IDIV only)
  - Enabled FRF ECC UE/CE (VIS FCMP only)
  - EXU-, IFU-, or TLU-initiated flush

C.8 Trap Logic Unit (TLU)

Exceptions and trap requests are conditions that may cause a thread to take a trap. A trap is a vectored transfer of control to supervisor software through a trap table (from the SPARC Version 9 Architecture). In the event of an exception or trap request, the trap logic unit (TLU) prevents the update of architectural state for the instruction or instructions after an exception. In many cases, the TLU relies on the execution units and the IFU to assist with the preservation of architectural state.

The TLU preserves the PC and NPC for the instruction with the exception. In some cases, the TLU must create a precise interrupt point for exceptions and interrupt requests not directly related to the instruction stream. In all cases, the TLU maintains the trap stack.

The TLU supports several logical functions:

- Flush logic — Generates flushes in response to exceptions to create precise interrupt points (when possible)
- Trap stack array (TSA) — Maintains trap state for the eight threads for up to six trap levels per thread
- Trap state machine — Holds and prioritizes trap requests for the eight threads in two thread groups

The TLU supports the following trap categories:

- **Precise trap —** Caused by a specific instruction. When a precise trap occurs, processor state reflects that all previous instructions have executed and completed and that the excepting instruction and subsequent instructions have not executed.

  The TLU ensures that the thread has completed all instructions prior to and no instruction subsequent to a precise trap exception so that the trap handler accesses the correct architectural state.

- **Disrupting trap —** Caused by a condition, not an instruction. The TLU services SPU exceptions, hardware exceptions, and XIR requests with disrupting traps.

  Once a disrupting trap has been serviced, the program may pick up where it left off. The condition that causes a disrupting trap may or may not be associated with a specific instruction. In some cases, the condition may be or may lead to a corruption of state, and therefore a disrupting trap may degenerate into a reset trap.

- **Reset trap** — Occurs when hardware or software determines that the hardware must be reset to a known state Once a reset trap has been serviced, the program does not resume.

  On OpenSPARC T2, a POR reset can only occur after a power-on. All other reset traps can be taken only if the thread can make forward progress. A reset trap will not resolve a deadlock.
- **Deferred trap** — On OpenSPARC T2, the *store_error* trap is the only deferred trap; it is implemented as though it were a deferred trap.

The TLU receives exception reports and trap requests from trapping instructions, SPU, hardware monitors, steering registers, and the crossbar. When the TLU receives an exception or trap request, it must first flush the relevant thread from the machine, to ensure that the trap handler can proceed without corruption from the thread itself.

Only instructions from the trapping thread are flushed. Instructions for other threads continue executing or remain in instruction buffers.

In OpenSPARC T2, exceptions can be caused by the following:

- Execution unit instructions
- Load-store unit instructions
- Floating-point and graphics unit instructions
- Illegal instructions
- Invalid instructions
- Translation exceptions (MMU miss, access exception)
- Out-of-range virtual or real addresses
- Instructions with ECC errors (integer instruction, floating-point and graphics instructions, load misses, stores, instruction cache misses)
- DONE and RETRY instructions
- SIR instructions

Each thread can generate only one trap at a time. However, since the threads within a thread group share a trap interface to the IFU, only one thread per thread group can trap per cycle. The TLU prioritizes trap requests for the threads within a thread group as follows:

1. Reset trap requests
2. Disrupting trap requests
3. Exceptions on divides
4. Exceptions on load misses and long latency instructions
5. Exceptions on normal pipe FGU instructions
6. Exceptions on normal pipe EXU and LSU instructions
7. Microarchitectural redirects and ITLB reloads

Within a trap request priority level, the TLU uses a static priority from thread 0 to thread 3 to select which request to service.

C.9 Reliability and Serviceability

Soft errors fall into one of four classes: reported corrected (RC), reported uncorrected (RU), silent corrected (SC), and silent uncorrected (SU). The OpenSPARC T1 design minimizes silent errors, whether corrected or uncorrected. Most SRAMs and register files have ECC or parity protection. OpenSPARC T2 protects more arrays or increases protection of a given array by adding ECC or parity or by duplicating arrays to further reduce the silent error rate and the reported uncorrected (for example, fatal) error rate. OpenSPARC T2 cores do not support lockstep, checkpoint, or retry operations.

The RAS design considers four major types of structures for protection:

- Six-device, single-ported SRAM cells optimized for density, such as cache data arrays. These SRAM cells have high failure-in-time (FIT) rates (300–400 FITs per Mb in Epic8c). SRAMs that store critical data have ECC protection. Other SRAMs have parity protection.
- Register files, which are typically multiported. A register file cell has FIT rates on the order of half or less of a high-density SRAM cell. OpenSPARC T2 protects register files with parity or with ECC where a single-bit error cannot be tolerated.
- Flip-flops or latches used in datapath or control blocks. A flop has a FIT rate of one-third or less of a single-ported SRAM cell. In general, OpenSPARC T2 does not protect flops or latches. Flops and latches have parity or ECC protection where they are part of a larger datapath which is so protected.
- A CAM cell, whose FIT rate may be half of a standard SRAM cell. CAM cells are difficult to protect. Adding parity to a CAM cell eliminates false CAM hits due to single-bit errors but cannot detect false misses.

  OpenSPARC T1 “scrubs” large CAMs. CAM scrubbing is different from traditional DRAM scrubbing. As in DRAM scrubbing, CAM scrubbing reads a CAM location and checks its parity. Unlike DRAM scrubbing, CAM scrubbing cannot correct single-bit failures in all cases: If parity is bad and hardware cannot innocuously reload the entry, an error results. CAM protection on OpenSPARC T2 is pending.

The FIT rates for OpenSPARC T2 structures are similar to their OpenSPARC T1 counterparts. To improve FIT rates for the core and L2, OpenSPARC T2 protects structures that are unprotected on OpenSPARC T1 and improves protection on structures already protected on OpenSPARC T1. Alternatively, OpenSPARC T2 may redesign structures with a higher Qcrit to lower the FIT rates.

OpenSPARC T2 contains error status registers (ESRs) that describe hardware errors to software. The ESRs isolate the first error. In the case of multiple errors, they also indicate that multiple errors have occurred. Software can read and write the registers through ASI instructions. Software can use a software-controlled bit in the register to emulate a parity or ECC error (to allow debug of diagnostic software). In addition, the structures protected by parity or ECC provide mechanisms to inject errors (for test).

C.9.1 Core Error Attribution

In the following discussion, the term "core" refers to a virtual core or a specific thread on a physical core (for example, core 20 refers to thread 4 on physical core 2). Since OpenSPARC T2 has eight physical cores with eight threads each, cores are numbered from 0 to 63, inclusively.

Hardware-detected errors can either be attributed directly to a specific core or not. An example of the former is an instruction cache tag parity error during an instruction fetch. An example of the latter is an uncorrectable error on the write-back of a modified L2 cache line.

An error that can be attributed to a given core can either be precise or imprecise (disrupting). For example, an ITLB parity error is precise. An uncorrectable error on a read of a core's store queue data entry is imprecise. Even though the store instruction is known, the core has updated architectural state past the store by the time the store data is read from the store queue.

In parallel with error handling within the affected core, OpenSPARC T2 can request traps for arbitrary cores. The `ASI_CMP_ERROR_STEERING` register controls disrupting trap requests for arbitrary cores in response to corrected and uncorrected errors.

C.9.2 Core Error Logging

OpenSPARC T2 logs errors that are attributable to a given core in an ESR associated with that core. If enabled, these errors result in either precise, disrupting, or deferred traps. OpenSPARC T2 logs errors that are not attributable to a given core in a "global" ESR and, if enabled, directs a disrupting trap to the core identified in the `ASI_CMP_ERROR_STEERING` register.

Each major structure on the OpenSPARC T2 core with a significant FIT rate has an error detection scheme. The scheme for each structure depends on the way the structure is used and the effect of the scheme on the timing and physical layout. These schemes seek to reduce the number of silent errors and reported uncorrected errors.

The design defines specific hardware behavior for each recorded error. Handling of each error follows a general template. Hardware corrects any correctable errors and retries the operation (either implicitly or explicitly). If a structure does not support ECC or if the structure detects an uncorrectable error, the structure invalidates the corrupted data. After invalidation, the core retries the operation (either implicitly or explicitly). If the data cannot be invalidated or another error occurs as a result of a retry, hardware signals an unrecoverable error and requests a trap for the affected core.

C.10 Reset

A reset is usually raised in response to a catastrophic event. Depending on the event, it may not be possible for a core or for the entire chip to continue execution without a full reset (POR). All resets place the processor in `RED_state`.

OpenSPARC T2 provides the following resets (listed in priority order):

- **POR (power-on reset) —** Also known as a "hard" reset. Initiated through an external pin. Activated when the chip is first powered-up and power and clocks have stabilized. A hard reset completely erases the current state of the machine and initializes all on-chip flops, latches, register files, and memory arrays such as TLBs and caches to a known good state. If the chip is working properly, a hard reset is guaranteed to put each processor in a consistent state where it can begin to fetch and execute instructions. Although called POR, the clearing of all machine state does not require power cycling.

  The default POR state is for all available cores to be enabled and the lowest-numbered available core to be running. These values may be changed by the system controller, if present, during reset. These values take effect upon the deactivation or completion of POR. Caches are disabled following POR.

- **WMR (warm reset) —** Also known as a "soft" reset. Only partially clears OpenSPARC T2 state before branching to the new trap address and executing instructions under the new machine state. A soft reset has been used in previous SPARC processors to synchronize updating of registers that control clock ratios for the bus and memory interfaces. It is also defined as the synchronization point for disabling or enabling CMP cores. Updates to clock ratio and core enable registers do not take effect until after the next chip reset.

As with POR, the default for WMR is that all available cores are enabled and the lowest-numbered available core is running (unparked). These values may be changed by the SC, if present, during chip reset. The new values take effect upon the deactivation or completion of WMR. Depending upon the error state of the chip, it may not be possible for the chip to continue executing instructions.

- **XIR (externally initiated reset) —** A SPARC V9-defined trap. An XIR may be generated externally to OpenSPARC T2 through a chip pin. XIR does not reset any machine state other than internal pipeline state required to cause an OpenSPARC T2 core to take a trap and other than the V9-required side effects of state updates for an XIR trap. An XIR may be routed to all cores (threads) or a subset of them based upon the contents of the CMP `ASI_XIR_STEERING` register. Following recognition of an XIR, instruction fetching occurs at *RSTVADDR* || 60_{16}.
- **WDR (watchdog reset) —** A SPARC V9-defined trap. WDR can be initiated by an event (such as taking a trap when TL = *MAXTL*) that causes an entry into the V9 error state—the processor immediately generates a watchdog reset trap to take the core to `RED_state`. On OpenSPARC T2, a WDR also can result from a fatal error condition detected by on-chip error logic. A WDR affects only the strand that created it. When a WDR is recognized, instruction fetching begins at *RSTVADDR* || 40_{16}.
- **SIR (software-initiated reset) —** Occurs when privileged software on a thread executes the SIR instruction. SIR affects only the core that executed the SIR instruction. When an SIR is recognized, instruction fetching begins at *RSTVADDR* || 80_{16}.

`RED_state` is entered when any of the above resets occur. `RED_state` is indicated when PSTATE.red = 1. However, setting PSTATE.red = 1 with software does not result in a reset. In `RED_state`, the I-TLB, the D-TLB, and the MMU MSA and MCM are disabled. Address translation is also disabled; addresses are interpreted as physical addresses. Bits 63:47 of the address are ignored. `RED_state` does not affect the enabling or disabling of the caches.

C.11 Performance Monitor Unit (PMU)

The performance monitor unit (PMU) consists of the PCR and PIC registers for each thread. The PMU takes in events from the rest of the core and, based on the configuration of the PCR registers for each thread, optionally increments a PIC, sets an overflow bit if the PIC is within range, and indicates a trap request to the TLU.

To save area, all threads share two 32-bit adders, one for the PICH and one for the PICL. This means that a given thread has access only to the adders every eighth cycle. In turn, each PIC has an associated 4-bit accumulator, which increments each cycle an event occurred for that PIC. When the thread is selected, each of its two PICs and their corresponding accumulators are summed together in their corresponding 32-bit adder.

To save power, the PMU is clock-gated. It wakes up whenever an ASI read or write is active on the ASI ring, or when at least one counter from any thread is enabled for counting.

The PMU is divided into two parts: pmu_pct_ctl and pmu_pdp_dp. The former contains the control logic and PCR registers; the latter contains the PIC registers and associated adder and muxing logic, and the PMU ASI ring interface.

C.12 Debugging Features

The following are notable features in OpenSPARC T2 that aid in debugging.

- clk/pll observability on pll_char_out[1:0] pins connected to pll_charc block in PLL
- A 166-pin-wide debug port that serves as an observability vehicle to promote repeatability, tester characterization, chip hang debug, and general CPU and SOC debug.
- A high level of repeatability within OpenSPARC T2's synchronous clock domains. These include the CPU clock domain, the DRAM domain, and the I/O clock domain. Thus, a group of tests can be run many times, with slightly different starting parameters.

- Retraining of FBDs after reset deassertion.
- I/O quiescence during checkpoint, to get the chip to a quiescent state on every checkpoint before dumping software-visible state and asserting debug reset to get the synchronous portion of the chip to a known state.
- Implementation of debug events in SPARC cores and SoC.
- Access to debug capabilities through JTAG access.
- OpenSPARC T2 core debugging:
  - Instruction breakpoints
  - Instruction and data watchpoints
  - *control_transfer_instruction* exception taken each time the T2 core executes a taken control-transfer instruction
  - Single-instruction stepping
  - Overlap disabling
  - Soft-stop request from TCU to core
  - Shadow scan
  - For each physical core, one hyperprivileged, read/write, Core Debug Event Control register (DECR), shared by all strands.
- Interfaces with the test control unit (TCU):
  - Clock interface
  - Debug event interface
  - Scan interface
  - Single-step mode signals
  - Disable-Overlap mode signals

See *OpenSPARC T2 System-On Chip (SoC) Microarchitecture Specification for details.*

C.13 Test Control Unit (TCU)

The OpenSPARC T2 test control unit (TCU) provides access to the chip test logic. It also participates in reset, eFuse programming, clock stop/start sequencing, and chip debugging. The TCU, including JTAG, is completely stuck-fault testable via ATPG manufacturing scan.

Features available for debugging or test, implemented in OpenSPARC T2 and supported by the TCU, are as follows:

- ATPG or manufacturing scan — For stuck-fault testing
- TAP and boundary scan (JTAG) — Support for IEEE 1149.1 and 1149.6
- JTAG scan — For scan chain loading and unloading
- JTAG shadow scan — For inspection of specific registers while a part is running in system
- Support for Macrotest
- JTAG UCB — For CREG access through instructions sent to the NCU

- These instruction then intermix the transaction with normal requests. The NCU passes the results back to the TCU, which then sends out TDO.
- EFuse — For control and programming
- Transition fault testing — Done on the tester while PLLs are locked; slower domains may be directly driven through pins
- MBIST (memory built-in self-test) — Tests array bit cells and write/read mechanisms. BISI (built-in self-initialization) allows arrays to be initialized.
- LBIST - Logic BIST, implemented in cores
- Reset — For handshaking with RST unit to control scan flop reset and clock stop/start sequencing.
- L2 access — With JTAG through the SIU
- Debug support
- Support for SerDes — ATPG, STCI, boundary scan

C.14 System Interface Unit (SIU)

OpenSPARC T2 has on-chip multiple system I/O subsystems. OpenSPARC T2 integrates Fire's high-speed I/O core and connects directly to an x8 PCI-Express channel (2 GB/s/direction). OpenSPARC T2's integrated network I/O unit includes two 10-Gb Ethernet MACs (2.5GB/s/direction). The SIU provides 12 GB/s of raw bandwidth per direction and has flexible interfaces for the network interface unit (NIU) and data management unit (DMU) to access memory via eight secondary-level cache (L2) banks. SIU supports Fire's PCI-Express. For the NIU, the SIU was architected so as to allow write traffic to bypass other posted write traffic. The SIU does not support coherency.

The SIU also provides a data return path for reads to the peripheral I/O subsystems. The data for these PIO reads and interrupt messages generated by the PCI Express subsystem are ordered in the SIU before delivery to the noncacheable unit (NCU).

The SIU is partitioned physically and logically into two parts based on flow direction: SIU Inbound (SII) for inbound traffic and SIU Outbound (SIO) for outbound traffic.

All inbound traffic continues inbound through the SIU until it reaches the NCU or an L2 bank. All outbound traffic from NCU or L2 must leave SIU in the outbound direction. The NCU and L2 banks cannot send traffic to each other through the SIU, nor can the DMU and NIU send traffic toward each other through the SIU. Because the L2 banks have their own paths through the memory controllers to memory, the SIU sees each L2 bank as a slave device. The SIU assumes that L2 never initiates requests to the SIU. Likewise, network blocks are always seen as master devices pulling from and pushing data to L2 only.

All traffic uses a packet transfer interface. Each packet is 1 or 2 consecutive address/header cycles immediately followed by 0 or more consecutive data/ payload cycles. The SIU follows L2's addressing convention: big endian where the databytes for the lowest address are transferred first. Where applicable, byte enables are positional where byte_enable{0} always refer to databits{7:0} for all interfaces.

The interfaces between the SIU and L2 are in the core clock domain—1.5 GHz. The interfaces between SIU and DMU, NIU, NCU are in the I/O clock domain—350 MHz or 1/4 core clock frequency.

APPENDIX **D**

OpenSPARC T1 Design Verification Suites

This appendix was adapted from Chapter 3 of *OpenSPARC T1 Design and Verification User's Guide*, Part Number 819-5019-12, March 2007, Revision A.

This appendix describes the following topics:

D.1 OpenSPARC T1 Verification Environment

The OpenSPARC T1 verification environment is a highly automated environment. With a simple command, you can run the entire regression suite for the OpenSPARC T1 processor, containing thousands of tests. With a second command, you can check the results of the regression.

The OpenSPARC T1 Design/Verification package comes with two testbench environments: `core1` and `chip8`. Each environment can perform either a mini-regression or a full regression. TABLE D-1 lists the testbench environment elements.

TABLE D-1 Testbench Environment Elements

`core1` Testbench Environment	`chip8` Testbench Environment
One SPARC CPU core	Full OpenSPARC T1 chip, including all 8 cores
Cache	Cache
Memory	Memory
Crossbar	Crossbar
— (No I/O subsystem)	I/O subsystem

OpenSPARC T1 Release 1.4 and later include a third regression environment for single-thread implementation of the OpenSPARC T1 core. This regression environment has all the components present in `core1` except that it supports only one hardware thread and removes the stream processing unit (SPU). This implementation is primarily developed to create a core with a footprint amenable for the FPGA map. The SPU can be added back into the design by disablement of the `FPGA_SYN_NO_SPU` flag during design compile time.

The verification environment uses source code in various languages. TABLE D-2 summarizes the types of source code and their uses.

TABLE D-2 Source Code Types in the Verification Environment

Source Code Language	Used for:
Verilog	Chip design, testbench drivers, and monitors
Vera	Testbench drivers, monitors, and coverage objects. Use of Vera is optional
Perl	Scripts for running simulations and regressions
C and C++	PLI (Programming Language Interface) for Verilog
SPARC Assembly	Verification tests

The top-level module for the testbench is called `cmp_top`. The same testbench is used for both the `core1` and `chip8` environments with compilation-time options.

D.2 Regression Tests

Each environment supports a mini-regression and a full regression. TABLE D-3 describes the regression groups.

TABLE D-3 Details of Regression Groups

Regression Group Name	Environment	No. of Tests	Disk Space Needed to Run (Mbyte)
`thread1_mini`	`thread1`	42	25
`thread1_full`	`thread1`	605	900
`core1_mini`	`core1`	68	41
`core1_full`	`core1`	900	1,680
`chip8_mini`	`chip8`	492	1,517
`chip8_full`	`chip8`	3789	29,000

To run a regression:

1. **Create the `$MODEL_DIR` directory and change to that directory. At the command line, type**

```
mkdir $MODEL_DIR
cd $MODEL_DIR
```

2 (a). Run the `sims` command with chosen parameters.

For example, to run a mini-regression for the `core1` environment using the VCS simulator, from the command line, type

```
sims -sim_type=vcs -group=core1_mini
```

This command creates two directories:

- A directory called `core1` under `$MODEL_DIR`. The regression compiles Vera and Verilog code under the `core1` directory. This is the Vera and Verilog build directory.
- A directory named with today's date and a serial number, such as `2006_01_07_0` (the format is `YYYY_MM_DD_ID`) under the current directory where simulations will run. This is the Verilog simulation's run directory. Each diagnostics test has one subdirectory under this directory

By default, the simulations are run with Vera. If you do not want to use Vera, add following option to the `sims` command:

```
-novera_build -novera_run
```

2 (b). To run regressions on multiple groups at one time, specify multiple `-group=` *parameters* at the same time.

For a complete list of command-line options for the `sims` command, see Appendix A of *OpenSPARC T1 Design and Verification User's Guide*.

3. Run the `regreport` command to get a summary of the regression. From the command line, type

```
regreport $PWD/2006_01_25_0 > report.log
```

For the `core1_mini` regression, results are reported for 68 tests.

D.2.1 The `sims` Command Actions

When running a simulation, the `sims` command performs the following steps:

1. Compiles the design into the `$MODEL_DIR/core1` or `$MODEL_DIR/chip8` directory, depending on which environment is being used.
2. Creates a directory for regression called *$PWD/DATE_ID*, where *$PWD* is the user's current directory, *DATE* is in `YYYY_MM_DD` format, and *ID* is a serial number starting with 0.

   For example, for the first regression on Jan 25, 2006, a directory called *$PWD*/`2006_01_26_0` is created. For the second regression run on the same day, the last ID is incremented to become *$PWD*/`2006_01_26_1`.
3. Creates a `master_diaglist.`*regression-group* file under the above directory, such as `master_diaglist.core1_mini` for the `core1_mini` regression group. This file is created based on diaglists under the `$DV_ROOT/verif/diag` directory.
4. Creates a subdirectory with the test name under the regression directory created in step 2 above.
5. Creates a `sim_command` file for the test based on the parameters in the diaglist file for the group.
6. Executes `sim_command` to run a Verilog simulation for the test. If the `-sas` option is specified for the test, `sim_command` also runs the SPARC Architecture Simulator (SAS) in parallel with the Verilog simulator and compares the results of the Verilog simulation with the SAS results after each instruction.

   The `sim_command` command creates many files in the test directory. Following are the sample files in the test directory:

   ```
   diag.ev       diag.s       l2way.log    perf.log
   sas.log.gz    sims.log     symbol.tbl   sim.perf.log
   diag.exe.gz   efuse.img    midas.log    sim_command
   status.log    sim.log.gz
   ```

The `status.log` file has a summary of the status, where the first line contains the name of the test and its status (`PASS/FAIL`), for example,

```
Diag: xor_imm_corner:model_core1:core1_full:0    PASS
```

7. Repeats steps 4 to 6 for each test in the regression group.

D.2.2 Running Regression With Other Simulators

To use a Verilog simulator other than VCS or NC-Verilog, use the following options for the `sims` command:

`-sim_type=`*your-simulator-name*

`-sim_build_cmd=`*your-simulator-command-to-build-compile-RTL*

`-sim_run_cmd=`*your-simulator-command-to-run-simulations*

`-sim_build_args=`*arguments-to-build-compile*

`-sim_run_args=`*arguments-to-run-simulation*`s`

The `sim_type`, `sim_build_cmd`, and `sim_run_cmd` options need be specified only once. The `sim_build_args` and `sim_run_args` can be specified multiple times to specify multiple argument options.

D.3 Verification Code

This section outlines Verilog and Vera code structures and locations.

D.3.1 Verilog Code Used for Verification

Various testbench drivers and monitors are written in Verilog. TABLE D-4 presents a list of all Verilog modules, the location of the source code, and descriptions. All verification Verilog files are in the `$DV_ROOT/verif/env` directory..

TABLE D-4 OpenSPARC T1 Verification Testbench Modules

Module Name	Type	# of instances	Instance Names	Directory Location Under `$DV_ROOT/verif/env`	Description
`OpenSPARCT1`	Chip	1	`iop`	`$DV_ROOT/design/sys/iop/rtl`	OpenSPARC T1 top level
`bw_sys`	Driver	1	`bw_sys`	`cmp`	SSI bus driver
`cmp_clk`	Driver	1	`cmp_clk`	`cmp`	Clock driver
`cmp_dram`	Model	1	`cmp_dram`	`cmp`	DRAM modules
`cmp_mem`	Driver	1	`cmp_mem`	`cmp`	Memory tasks
`cpx_stall`	Driver	1	`cpx_stall`	`cmp`	CPX stall
`dbg_port_chk`	Monitor	1	`dbg_port_chk`	`cmp`	Debug port checker
`dffrl_async`	Driver	4	`flop_ddr0|1|2|3_oe`	`$DV_ROOT/design/sys/iop/common/rtl`	Flip-flop
`err_inject`	Driver	1	`err_inject`	`cmp`	Error Injector
`jbus_monitor`	Monitor	1	`jbus_monitor`	`iss/pli/jbus_mon/rtl`	J-Bus Monitor
`jp_sjm`	Driver	2	`j_sjm_4, j_sjm_5`	`iss/pli/sjm/rtl`	J-Bus Driver
`monitor`	Monitor	1	`monitor`	`cmp`	Various monitors
`one_hot_mux_mon`	Monitor	1	`one_hot_mux_mon`	`cmp`	Hot mux monitor
`pcx_stall`	Driver	1	`pcx_stall`	`cmp`	PCX stall
`sas_intf`	SAS	1	`sas_intf`	`cmp`	SAS interface
`sas_tasks`	SAS	1	`sas_tasks`	`cmp`	SAS tasks
`slam_init`	Driver	1	`slam_init`	`cmp`	Initialization tasks
`sparc_pipe_flow`	Monitor	1	`sparc_pipe_flow`	`cmp`	SPARC pipe flow monitor
`tap_stub`	Driver	1	`tap_stub`	`cmp`	JTAG driver

D.3.2 Vera Code Used for Verification

Two types of Vera code are included in the OpenSPARC T1 verification environment:

- Testbench driver and monitor Vera code
- Vera object coverage Vera code

Vera code is in the `$DV_ROOT/verif/env/cmp/vera` directory. Each object coverage module has a corresponding subdirectory. Following is a list of Vera object coverage modules:

`cmpmss_coverage`	`coreccx_coverage`	`dram_coverage`
`err_coverage`	`exu_coverage`	`ffu_coverage`
`fpu_coverage`	`ifu_coverage`	`lsu_coverage`
`mmu_coverage`	`mt_coverage`	`spu_coverage`
`tlu_coverage`	`tso_coverage`	

Object coverage Vera code for `jbi` is in the `$DV_ROOT/verif/env/iss/vera/jbi_coverage` directory. Object coverage Vera code is used only for the `chip8_cov` regression groups.

D.4 PLI Code Used for the Testbench

Verilog's programming language interface (PLI) is used to drive and monitor the simulations of the OpenSPARC T1 design. There are eight different directories for PLI source code. Some PLI code is in C language, and some is in C++ language. The object libraries for the VCS simulator and NC-Verilog simulator are included for the PLI code in the `$DV_ROOT/tools/SunOS/sparc/lib` directory. TABLE D-5 gives the details of PLI code directories, VCS libraries, and NC-Verilog libraries.

TABLE D-5 PLI Source Code and Object Libraries

PLI Name	Source Code Location Under `$DV_ROOT`	VCS Object Library Name	NC-Verilog Object Library Name	Description
`iop`	`tools/pli/iop`	`libiob.a`	`libiob_ncv.so`	Monitors and drivers
`mem`	`tools/pli/mem`	`libmem_pli.a`	`libmem_pli_ncv.so`	Memory read/write
`socket`	`tools/pli/socket`	`libsocket_pli.a`	`libsocket_pli_ncv.so`	Sockets to SAS
`utility`	`tools/pli/utility`	`libutility_pli.a`	`libutility_ncv.so`	Utility functions
`common`	`verif/env/iss/pli/common/c`	`libjpcommon.a`	`libjpcommon_ncv.so`	Common PLI functions
`jbus_mon`	`verif/env/iss/pli/jbus_mon/c`	`libjbus_mon.a`	`libjbus_mon_ncv.so`	J-Bus monitor
`monitor`	`verif/env/iss/pli/monitor/c`	`libmonitor.a`	`libmonitor_ncv.so`	Various
`sjm`	`verif/env/iss/pli/sjm/c`	`libsjm.a`	`libsjm_ncv.so`	J-Bus driver

VCS object libraries are statically linked libraries (`.a` files) which are linked when VCS compiles the Verilog code to generate a `simv` executable. NC-Verilog object libraries are dynamically loadable libraries (`.so` files) which are linked dynamically while running the simulations.

Makefiles are provided to compile PLI code. Under each PLI directory are a `makefile` file to create a static object library (`.a` file). There is a `makefile.ncv` file under each PLI directory to create a dynamic object library.

To compile all PLI libraries, run the mkplilib script. This script has three options, as listed in TABLE D-6.

TABLE D-6 Options for the mkplilib Script

Option	Used for
vcs	Compiling PLI libraries for VCS
ncverilog	Compiling PLI libraries for NC-Verilog
clean	Deleting all PLI libraries

To compile PLI libraries with your chosen option, for example, to compile PLI libraries in VCS, at the command line, type

```
mkplilib vcs
```

Either version of this procedure, VCS or NC-Verilog, compiles C or C++ code, creates static or dynamic libraries, and copies them to the $DV_ROOT/tools/SunOS/sparc/lib directory.

D.5 Verification Test File Locations

The verification or diagnostics tests (diags) for the OpenSPARC T1 processor are written in SPARC assembly language (the file names have a .s extension). Some diags require command files for a J-Bus driver. Those command files are named sjm_4.cmd and sjm_5.cmd. Some diagnostics test cases in SPARC assembly are automatically generated by Perl scripts.

The main diaglist for core1 is core1.diaglist. The main diaglist for chip8 is chip8.diaglist. These main diaglists for each environment also include many other diaglists. The locations of various verification test files are listed in TABLE D-7.

TABLE D-7 Verification Test File Directories

Directory	Contents
`$DV_ROOT/verif/diag`	All diagnostics, various diagnostic list files with the extension `.diaglist`.
`$DV_ROOT/verif/diag/assembly`	Source code for SPARC assembly diagnostics. More than 2000 assembly test files.
`$DV_ROOT/verif/diag/efuse`	EFuse cluster default memory load files.

D.6 Compilation of Source Code for Tools

To compile source code for some Sun tools used for the OpenSPARC T1 processor, use the `mktools` script. The tools source code is located in the `$DV_ROOT/tools/src` directory.

The `mktools` script compiles the source code and copies the binaries to `$DV_ROOT/tools/`*operating-system*`/`*processor-type* directory, where:

- *operating-system* is defined by the `uname -s` command
- *processor-type* is defined by the `uname -p` command

D.7 Gate-Level Verification

OpenSPARC T1 depends heavily on cross-module references (XMRs) within the verification environment. Therefore, dropping in a netlist in place of the RTL core will produce a high number of XMR errors. In order to overcome this difficulty, a simple playback support is now added.

Although it is anticipated that this method will be useful primarily to verify FPGA synthesized netlists, it could be potentially used with netlists generated by semi-custom synthesis flows as well (for example, Synopsys).

CAUTION! Running this vector playback mechanism on RTL, although feasible, is not recommended because of some array initialization issues. In the gate playback mode, all arrays are explicitly initialized to zero while in RTL and some arrays are initialized to random values. This may result in mismatch in playback simulation. If RTL arrays are initialized correctly (zeroes), then this mechanism can be used to verify the RTL netlist as well.

To verify a netlist:

1. **Run RTL mini or full regression to generate stimuli files for netlist verification.**

   To do this, add the `-vcs_build_args=$DV_ROOT/verif/env/cmp/playback_dump.v` option to the regression command.

   For example, the `thread1_mini` regression command for the SPARC-level driver (stimuli) generation would require the following (all on one line):

   ```
   sims -sim_type=vcs -group=thread1_mini -debussy
   -vcs_build_args=$DV_ROOT/verif/env/cmp/
   playback_dump.v
   ```

   The preceding regression generates the `stimuli.txt` file under the run directory of each diagnostic. Sample `stimuli.txt` files are included under `$DV_ROOT/verif/gatesim` for the `thread1_mini` regression (file `thread1_mini_stim.tar.gz`). These files are generated with the VCS build flags: args `(-vcs_build_args)` `FPGA_SYN`, `FPGA_SYN_1THREAD`, and `FPGA_SYN_NO_SPU`.

2. **Create a Verilog file list that includes the following files:**

   - `$DV_ROOT/verif/env/cmp/playback_driver.v`
   - *SPARC-level-gate-netlist*`.v`
   - *library-used-for-synthesis*`.v`

   A sample `flist` is provided under `$DV_ROOT/verif/gatesim` for reference (file `flist.xilinx_unisims`)

3. **Compile the design to build the gate-level model.**

   A sample compile script is provided under the `$DV_ROOT/verif/gatesim` directory. To use the compile script, at the command line, type

   `$DV_ROOT/verif/gatesim/build_gates` ***flist***

4. **Run the simulation by including `+stim_file=`*path-to-stim-file*`/stimuli.txt`**
   - If the playback fails, the simulation returns with `Playback FAILED with # mismatches!`
   - If the playback passes, the simulation returns with `Playback PASSED!`

Use `fsdb` generation options in the compile script to debug failing runs.

A simple `run_gates` script is also included for reference under the `$DV_ROOT/verif/gatesim` directory. To run the script, at the command line, type

`$DV_ROOT/verif/gatesim/run_gates` *path-to-stim-file*

TABLE D-8 lists the gate netlist files and describes their content.

TABLE D-8 Gate Netlist Files

Directory/File	**Contents**
`$DV_ROOT/verif/gatesim/build_gates`	Compile script to create gate-level model of SPARC `netlist`
`$DV_ROOT/verif/gatesim/run_gates`	Run script to execute playback of vectors on gate `netlist`
`$DV_ROOT/verif/gatesim/flist.xilinx.unisims`	Sample Verilog file list with Xilinx synthesis library
`$DV_ROOT/verif/gatesim/thread1_mini_stim.tar.gz`	FOR REFERENCE ONLY: Collection of prepackaged stimulus files for `thread1_mini` regression suite

APPENDIX E

OpenSPARC T2 Design Verification Suites

This appendix was adapted from *OpenSPARC T2 Design and Verification User's Guide*, Part Number 820-2729-10, December 2007, Revision A.

This appendix describes the following topics:

E.1 System Requirements

OpenSPARC T2 regressions are currently supported to run only on SPARC systems running the Solaris 9 or Solaris 10 Operating System.

Disk space requirements are listed in TABLE E-1.

TABLE E-1 Disk Space Requirements

Disk Space Required	Required for:
3.1 Gbyte	Download, unzip or uncompress, and extract from the tar file
0.3 Gbyte	Run a mini-regression
11.5 Gbyte	Run a full regression
1.1 Gbyte	Run synthesis
17 Gbyte	Total

E.2 OpenSPARC T2 Verification Environment

The OpenSPARC T2 verification environment is a highly automated environment. With a simple command, you can run the entire regression suite for the OpenSPARC T2 processor, containing hundreds of tests. With a second command, you can check the results of the regression.

The OpenSPARC T2 Design and Verification package comes with two testbench environments: `cmp1` and `fc8`. TABLE E-2 lists the testbench environment entities.

TABLE E-2 Testbench Environment Elements

`cmp1` Testbench Environment	**`fc8` Testbench Environment**
One SPARC CPU core	Full OpenSPARC T2 chip, including all 8 cores
Cache	Cache
Memory	Memory
Crossbar	Crossbar
— (No I/O subsystem)	I/O subsystem

The verification environment uses source code in various languages. TABLE E-3 summarizes the types of source code and their uses.

TABLE E-3 Source Code Types in the Verification Environment

Source Code Language	**Used for:**
Verilog	Chip design, testbench drivers, and monitors
Vera	Testbench drivers, monitors, and coverage objects; use of Vera is optional
Perl	Scripts for running simulations and regressions
C and C++	Programming Language Interface (PLI) for Verilog
SPARC Assembly	Verification tests

To run a regression, use the `sims` command as described below. The important parameters for the `sims` command are as follows:

- `-sys`System type. Set this to `cmp1` or `fc8`. For example: **`-sys=cmp1`**
- `-group`Regression group name. The choices for `-group` are `cmp1_mini_T2`, `cmp1_all_T2`, `fc8_mini_T2`, and `fc8_all_T2`. For example, **`-group=cmp1_mini_T2`**
  - (For `cmp1` only) `-diaglist_cpp_args=-DT2`
  - (For `fc8` only) `-config_cpp_args=-DT2`

For help, type **`sims -h`**

E.3 Regression Tests

Each environment supports a mini-regression and a full regression. TABLE E-4 describes the regression groups.

TABLE E-4 Details of Regression Groups

Regression Group Name	Environment	No. of Tests	Disk Space Needed to Run (Mbytes)
`cmp1_mini_T2`	`cmp1`	51	75
`cmp1_all_T2`	`cmp1`	763	2500
`fc8_mini_T2`	`fc8`	84	246
`fc8_full_T2`	`fc8`	534	9000

To run a regression:

1. **Create the `$MODEL_DIR` directory and change directory to it. At the command line, type**

```
mkdir $MODEL_DIR
cd $MODEL_DIR
```

2. **Run the `sims` command with chosen parameters.**

   For instance, to run a mini-regression for the `cmp1` environment using the VCS simulator, from the command line, type

```
sims -sys=cmp1 -group=cmp1_mini_T2 -diaglist_cpp_args=-DT2
```

   This command creates two directories:

   - A directory called `cmp1` under `$MODEL_DIR`. The regression compiles Vera and Verilog code under the `cmp1` directory. This is the Vera and Verilog build directory. By default, the simulations are run with Vera.

- A directory named with today's date and a serial number, such as `2007_01_07_0` (the format is `YYYY_MM_DD_ID`) under the current directory where simulations will run. This is the Verilog simulation's run directory. Each diagnostics test has one subdirectory under this directory.

3. **Run the `regreport` command to get a summary of the regression. From the command line, type**

   ```
   regreport $PWD/2007_08_07_0 > report.log
   ```

When running a simulation, the `sims` command performs the following steps:

1. Compiles the design into the `$MODEL_DIR/cmp1` or `$MODEL_DIR/fc8` directory, depending on which environment is being used.

2. Creates a directory for regression called *$PWD/DATE_ID*, where *$PWD* is the current directory, *DATE* is in `YYYY_MM_DD` format, and *ID* is a serial number starting with 0. For example, for the first regression on August 7, 2007, a directory called *$PWD*/`2007_08_07_0` is created. For the second regression run on the same day, the last ID is incremented to become *$PWD*/`2007_08_07_1`.

3. Creates a `master_diaglist.`*regression-group* file under the above directory. such as `master_diaglist.cmp1_mini_T2` for the `cmp1_mini_T2` regression group. This file is created based on diaglists under the `$DV_ROOT/verif/diag` directory.

4. Creates a subdirectory with the test name under the regression directory created in step 2 above.

5. Creates a `sim_command` file for the test based on the parameters in the diaglist file for the group.

6. Executes `sim_command` to run a Verilog simulation for the test. If the `-sas` option is specified for the test, `sim_command` also runs the SPARC Architecture Simulator (SAS) in parallel with the Verilog simulator. The results of the Verilog simulation are compared with the SAS results after each instruction.

   The `sim_command` command creates many files in the test directory. Following are the sample files in the test directory.

   ```
   diag.ev       diag.s        raw_coverage     seeds.log
   status.log    vcs.log.gz    diag.exe.gz      midas.log
   sas.log.gz    sims.log      symbol.tbl       vcs.perf.log
   ```

The `status.log` file has a summary of the status, where the first line contains the name of the test and its status (`PASS/FAIL`), for example,

```
   Rundir: tlu_rand05_ind_03:cmp1_st:cmp1_mini_T2:0
PASS
```

7. Repeats steps 4 to 6 for each test in the regression group.

E.4 PLI Code Used For the Testbench

Verilog's Programming Language Interface (PLI) is used to drive and monitor the simulations of the OpenSPARC T2 design. There are eight different directories for PLI source code. Some PLI code is in C language, and some is in C++ language. TABLE E-5 gives the details of PLI code directories and VCS libraries.

TABLE E-5 PLI Source Code and Object Libraries

PLI name	Source Code Location Under `$DV_ROOT`	VCS Object Library Name	Description
`iob`	`verif/env/common/pli/cache`	`libiob.a`	Cache warming routines
`mem`	`model/infineon`	`libbwmem_pli.a`	Memory read/write
`socket`	`verif/env/common/pli/socket`	`libsocket_pli.a`	Sockets to SAS
`utility`	`verif/env/common/pli/utility`	`libbwutility_pli.a`	Utility functions
`monitor`	`verif/env/common/pli/monitor/c`	`libmonitor_pli.a`	Various
`global_chkr`	`verif/env/common/pli/global_chkr/`	`libglobal_chkr.a`	Various checkers

VCS object libraries are statically linked libraries (`.a` files) that are linked when VCS compiles the Verilog code to generate a `simv` executable.

A `makefile` file under the `$DV_ROOT/tools/pli` directory will compile static executables (`.a` files) of the PLI code.

E.5 Verification Test File Locations

The verification or diagnostics tests (diags) for the OpenSPARC T2 processor are written in SPARC assembly language (the file names have a `.s` extension). Some diagnostics test cases in SPARC assembly are automatically generated by Perl scripts.

The main diaglist for `cmp1` is `cmp1.diaglist`. The main diaglist for `fc8` is `fc8.diaglist`. These main diaglists for each environment also include many other diaglists. The locations of various verification test files are listed in TABLE E-6.

TABLE E-6 Verification Test File Directories

Directory	Contents
`$DV_ROOT/verif/diag`	All diagnostics, various diagnostic list files with the extension `.diaglist`
`$DV_ROOT/verif/diag/assembly`	Source code for SPARC assembly diagnostics. More than 1400 assembly test files
`$DV_ROOT/verif/diag/efuse`	eFuse cluster default memory load files

APPENDIX **F**

OpenSPARC Resources

This appendix provides references to additional OpenSPARC resources available on the World Wide Web.

- For general OpenSPARC information, visit:
  `http://OpenSPARC.net`
- For information on programs supporting academic uses of OpenSPARC, visit:
  `http://www.OpenSPARC.net/edu/university-program.html`
  Specific questions may be sent by email to
  `OpenSPARC-UniversityProgram@sun.com`
- For more information regarding FPGA-based OpenSPARC projects, visit:
  `http://www.opensparc.net/fpga/index.html`
  Information regarding the Xilinx OpenSPARC FPGA Board is on page:
  `http://www.OpenSPARC.net/edu/university-program.html`
- For OpenSPARC T1, UltraSPARC Architecture 2005, and T1 Hypervisor specification documents, visit:
  `http://www.opensparc.net/opensparc-t1/`
- To download the OpenSPARC T1 processor design, verification tools, simulation tools, and performance modeling tools, visit:
  `http://www.opensparc.net/opensparc-t1/downloads.html`
- For OpenSPARC T2 and UltraSPARC Architecture 2007 specification documents, visit:
  `http://www.opensparc.net/opensparc-t2/`
- To download the OpenSPARC T2 processor design, verification tools, simulation tools, and performance modeling tools, visit:
  `http://www.opensparc.net/opensparc-t2/downloads.html`

- To read (or post to) OpenSPARC Community Forums, visit:
  `http://forums.sun.com/category.jspa?categoryID=120`
- If you have OpenSPARC-related questions, the best way to get an answer (and share the answer with other community members) is to post your question in the General OpenSPARC Discussion Forum at:
  `http://forums.sun.com/forum.jspa?forumID=837`

APPENDIX **G**

OpenSPARC Terminology

This appendix defines concepts and terminology applicable to OpenSPARC T1 and OpenSPARC T2.

A

address space	A range of 2^{64} locations that can be addressed by instruction fetches and load, store, or load-store instructions. *See also* **address space identifier (ASI)**.
address space identifier (ASI)	An 8-bit value that identifies a particular address space. An ASI is (implicitly or explicitly) associated with every instruction access or data access. *See also* **implicit ASI**.
aliased	Said of each of two virtual or real addresses that refer to the same underlying memory location.
application program	A program executed with the virtual processor in nonprivileged mode.
ASI	Address space identifier.
ASR	Ancillary State register
available (virtual processor)	A virtual processor that is physically present and functional and that can be enabled and used.

B

big-endian	An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte's significance decreases as its address increases.
BIST	Abbreviation for built-in self-test
BLD	(Obsolete) abbreviation for Block Load instruction; replaced by LDBLOCKF.
BST	(Obsolete) abbreviation for Block Store instruction; replaced by STBLOCKF.
byte	Eight consecutive bits of data, aligned on an 8-bit boundary.

C

CAM	Abbreviation for content-addressable memory.
CCR	Abbreviation for Condition Codes register.
clean window	A register window in which each of the registers contain 0, a valid address from the current address space, or valid data from the current address space.
cleared	A term applied to an error when the originating incorrect signal or datum is set to a value that is not in error. An originating incorrect signal that is stored in a memory (a stored error) may be cleared automatically by hardware action or may need software action to clear it. An originating incorrect signal that is not stored in any memory needs no action to clear it. (For this definition, "memory" includes caches, registers, flip-flops, latches, and any other mechanism for storing information, and not just what is usually considered to be system memory.)
CMT	Chip-level multithreading (or, as an adjective, chip-level multithreaded). Refers to a physical processor containing more than one virtual processor.

coherence	A set of protocols guaranteeing that all memory accesses are globally visible to all caches on a shared-memory bus.
completed (memory operation)	Said of a memory transaction when an idealized memory has executed the transaction with respect to all processors. A load is considered completed when no subsequent memory transaction can affect the value returned by the load. A store is considered completed when no subsequent load can return the value that was overwritten by the store.
context	A set of translations that defines a particular address space. *See also* **memory management unit (MMU)**.
context ID	A numeric value that uniquely identifies a particular context.
copyback	The process of sending a copy of the data from a cache line owned by a physical processor core, in response to a snoop request from another device.
CPI	Cycles per instruction. The number of clock cycles it takes to execute an instruction.
core	In an OpenSPARC processor, may refer to either a virtual processor or a physical processor core.
correctable	A term applied to an error when at the time the error occurs, the error detector knows that enough information exists, either accompanying the incorrect signal or datum or elsewhere in the system, to correct the error. Examples include parity errors on clean L1s, which are corrected by invalidation of the line and refetching of the data from higher up in the memory hierarchy, and correctable errors on L2s. *See also* **uncorrectable**.
corrected	A term applied to an error when the incorrect signal or datum is replaced by the correct signal or datum, perhaps in a downstream location. Depending on the circuit, correcting an error may or may not clear it.
cross-call	An interprocessor call in a system containing multiple virtual processors.

CSR	Abbreviation for Control and Status register.
CTI	Abbreviation for control-transfer instruction.
current window	The block of 24 R registers that is presently in use. The Current Window Pointer (CWP) register points to the current window.

D

data access (instruction)	A load, store, load-store, or FLUSH instruction.
DCTI	Delayed control-transfer instruction.
demap	To invalidate a mapping in the MMU.
denormalized number	Synonym for **subnormal number**.
deprecated	The term applied to an architectural feature (such as an instruction or register) for which an OpenSPARC implementation provides support *only* for compatibility with previous versions of the architecture. Use of a deprecated feature must generate correct results but may compromise software performance. Deprecated features should not be used in new OpenSPARC software and may not be supported in future versions of the architecture.
DFT	Abbreviation for Design For Test.
disable (core)	The process of changing the state of a virtual processor to `Disabled`, during which all other processor state (including cache coherency) may be lost and all interrupts to that virtual processor will be discarded. *See also* **park** and **enable**.

disabled (core)	A virtual processor that is out of operation (not executing instructions, not participating in cache coherency, and discarding interrupts). *See also* **parked** and **enabled**.
double-word	An 8-byte datum. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.
DUT	Abbreviation for device (or unit) under test.
D-SFAR	Data Synchronous Fault Address register.

E

enable (core)	The process of moving a virtual processor from `Disabled` to `Enabled` state and preparing it for operation. *See also* **disable** and **park**.
enabled (core)	A virtual processor that is in operation (participating in cache coherency, but not executing instructions unless it is also `Running`). *See also* **disabled** and **running**.
error	A signal or datum that is wrong. The error can be created by some problem internal to the processor, or it can appear at inputs to the processor. An error can propagate through fault-free circuitry and appear as an error at the output. It can be stored in a memory, whether program-visible or not, and can later be either read out of the memory or overwritten.
ESR	Abbreviation for Error Status register.
even parity	The mode of parity checking in which each combination of data bits plus a parity bit contains an even number of '1' bits.
exception	A condition that makes it impossible for the processor to continue executing the current instruction stream. Some exceptions may be masked (that is, trap generation disabled — for example, floating-point exceptions masked by FSR.tem) so that the decision on whether or not to apply special processing can be deferred and made by software at a later time. *See also* **trap**.

explicit ASI	An ASI that is provided by a load, store, or load-store alternate instruction (either from its imm_asi field or from the ASI register).
extended word	An 8-byte datum, nominally containing integer data. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

F

fault	A physical condition that causes a device, a component, or an element to fail to perform in a required manner; for example, a short-circuit, a broken wire, or an intermittent connection.
fcc*n*	One of the floating-point condition code fields fcc0, fcc1, fcc2, or fcc3.
FGU	Floating-point and graphics unit (which most implementations specify as a superset of **FPU**).
floating-point exception	An exception that occurs during the execution of a floating-point operate (FPop) instruction. The exceptions are unfinished_FPop, unimplemented_FPop, *sequence_error*, *hardware_error*, *invalid_fp_register*, or *IEEE_754_exception*.
F register	A floating-point register. The SPARC V9 architecture includes single-, double-, and quad-precision F registers.
floating-point operate instructions	Instructions that perform floating-point calculations. FPop instructions do not include FBfcc instructions, loads and stores between memory and the F registers, or non-floating-point operations that read or write F registers.
floating-point trap type	The specific type of a floating-point exception, encoded in the FSR.ftt field.
floating-point unit	A processing unit that contains the floating-point registers and performs floating-point operations, as defined by the specification.

FPop	Abbreviation for **floating-point operate** (instructions).
FPRS	Floating-Point Register State register.
FPU	Floating-point unit.
FSR	Floating-Point Status register.

G

GL	Global Level register.
GSR	General Status register.

H

halfword	A 2-byte datum. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.
hyperprivileged	An adjective that describes: (1) the state of the processor when HPSTATE.hpriv = 1, that is, when the processor is in hyperprivileged mode; (2) processor state that is only accessible to software while the processor is in hyperprivileged mode; for example, hyperprivileged registers, hyperprivileged ASRs, or, in general, hyperprivileged state; (3) an instruction that can be executed only when the processor is in hyperprivileged mode.
hypervisor (software)	A layer of software that executes in hyperprivileged processor state. One purpose of hypervisor software (also referred to as "the Hypervisor") is to provide greater isolation between operating system ("supervisor") software and the underlying processor implementation.

I

ICE	Abbreviation for In-Circuit Emulation.

IEEE 754	IEEE Standard 754-1985, the IEEE Standard for Binary Floating-Point Arithmetic.
IEEE-754 exception	A floating-point exception, as specified by IEEE Std 754-1985. Listed within the specification as *IEEE_754_exception*.
implementation	Hardware or software that conforms to all of the specifications of an instruction set architecture (ISA).
implementation dependent	An aspect of the OpenSPARC architecture that can legitimately vary among implementations. In many cases, the permitted range of variation is specified. When a range is specified, compliant implementations must not deviate from that range.
implicit ASI	An address space identifier that is implicitly supplied by the virtual processor on all instruction accesses and on data accesses that do not explicitly provide an ASI value (from either an imm_asi instruction field or the ASI register).
initiated	Synonym for **issued**.
instruction field	A bit field within an instruction word.
instruction group	One or more independent instructions that can be dispatched for simultaneous execution.
instruction set architecture	A set that defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. Does not define clock cycle times, cycles per instruction, datapaths, etc. This book defines the OpenSPARC T1 and OpenSPARC T2 instruction set architectures.
ISS	Abbreviation for Instruction-Set Simulator.
integer unit	A processing unit that performs integer and control-flow operations and contains general-purpose integer registers and virtual processor state registers, as defined by this specification.

interrupt request	A request for service presented to a virtual processor by an external device.
inter-strand	Describes an operation that crosses virtual processor (strand) boundaries.
intra-strand	Describes an operation that occurs entirely within one virtual processor (strand).
invalid (ASI or address)	Undefined, reserved, or illegal.
ISA	Instruction set architecture.
issued	A memory transaction (load, store, or atomic load-store) is said to be "issued" when a virtual processor has sent the transaction to the memory subsystem and the completion of the request is out of the virtual processor's control. *Synonym for* **initiated**.
IU	Integer unit.

L

little-endian	An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the least significant; a byte's significance increases as its address increases.
load	An instruction that reads (but does not write) memory or reads (but does not write) location(s) in an alternate address space. Some examples of *load* include loads into integer or floating-point registers, block loads, and alternate address space variants of those instructions. *See also* **load-store** *and* **store**, the definitions of which are mutually exclusive with *load*.

load-store	An instruction that explicitly both reads and writes memory or explicitly reads and writes location(s) in an alternate address space. *Load-store* includes instructions such as CASA, CASXA, LDSTUB, and the deprecated SWAP instruction. *See also* **load** *and* **store**, the definitions of which are mutually exclusive with *load-store*.

M

MAS	Abbreviation for microarchitectural specification.
may	A keyword indicating flexibility of choice with no implied preference. **Note:** "may" indicates that an action or operation is allowed; "can" indicates that it is possible.
memory manage-ment unit	The address translation hardware in an OpenSPARC implementation that translates 64-bit virtual address into underlying physical addresses. The MMU is composed of the TLBs, ASRs, and ASI registers used to manage address translation. *See also* **context**, **physical address**, **real address**, and **virtual address**.
MMI	Abbreviation for Module Model interface.
MMU	Abbreviation for **memory management unit**.
multipro-cessor system	A system containing more than one processor.
must	A keyword indicating a mandatory requirement. Designers must implement all such mandatory requirements to ensure interoperability with other OpenSPARC architecture-compliant products. *Synonym for* **shall**.

N

Next Pro-gram Counter	Conceptually, a register that contains the address of the instruction to be executed next if a trap does not occur.

NFO	Nonfault access only.
nonfaulting load	A load operation that behaves identically to a normal load operation, except when supplied an invalid effective address by software. In that case, a regular load triggers an exception whereas a nonfaulting load appears to ignore the exception and loads its destination register with a value of zero (on an OpenSPARC processor, hardware treats regular and nonfaulting loads identically; the distinction is made in trap handler software). *Contrast with* **speculative load**.
nonprivileged	An adjective that describes (1) the state of the virtual processor when PSTATE.priv = 0 and HPSTATE.hpriv = 0, that is, when it is in nonprivileged mode; (2) virtual processor state information that is accessible to software regardless of the current privilege mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state; (3) an instruction that can be executed in any privilege mode (hyperprivileged, privileged, or nonprivileged).
nonprivileged mode	The mode in which a virtual processor is operating when executing application software (at the lowest privilege level). Nonprivileged mode is defined by PSTATE.priv = 0 and HSTATE.hpriv = 0. *See also* **privileged** *and* **hyperprivileged**.
non-translating ASI	An ASI that does not refer to memory (for example, refers to control/status register(s)) and for which the MMU does not perform address translation.
normal trap	A trap processed in `execute_state` (or equivalently, a non-`RED_state` trap). *Contrast with* **`RED_state` trap**.
NPC	Next program counter.
npt	Abbreviation for nonprivileged trap.
nucleus software	Privileged software running at a trap level greater than 0 (TL> 0).
NUMA	Abbreviation for nonuniform memory access.

N-REG_ WINDOW	The number of register windows present in a particular implementation.

O

OBP	Abbreviation for Open Boot PROM.
octlet	Eight bytes (64 bits) of data. Not to be confused with “octet,” which has been commonly used to describe eight bits of data. In this book, the term *byte*, rather than octet, is used to describe eight bits of data.
odd parity	The mode of parity checking in which each combination of data bits plus a parity bit together contain an odd number of ‘1’ bits.
opcode	A bit pattern that identifies a particular instruction.
optional	A feature not required for UltraSPARC Architecture 2005 and UltraSPARC Architecture 2007 compliance.

P

PA	Abbreviation for physical address.
park	The process of suspending a virtual processor from operation. There may be a delay until the virtual processor is parked, but no heavyweight operation (such as a reset) is required to complete the parking process. *See also* **disable** *and* **unpark**.
parked	Said of a virtual processor that is suspended from operation. When parked, a virtual processor does not issue instructions for execution but still maintains cache coherency. *See also* **disabled, enabled,** *and* **running**.
PC	Program Counter register.
PCR	Performance Control register.

physical address	An address that maps to actual physical memory or I/O device space. *See also* **real address** *and* **virtual address**.
physical core	The term *physical processor core*, or just *physical core*, is similar to the term *pipeline* but represents a broader collection of hardware that is required for performing the execution of instructions from one or more software threads. For a detailed definition of this term, see page 757 of the UltraSPARC Architecture manual. *See also* **pipeline**, **processor**, **strand**, **thread**, *and* **virtual processor**.
physical processor	Synonym for processor; used when an explicit contrast needs to be drawn between processor and virtual processor. *See also* **processor** *and* **virtual processor**.
PIC	Performance Instrumentation Counter.
PIL	Processor Interrupt Level register.
pipeline	Refers to an execution pipeline, the basic collection of hardware needed to execute instructions. For a detailed definition of this term, see page 757 of the UltraSPARC Architecture manual. *See also* **physical core**, **processor**, **strand**, **thread**, *and* **virtual processor**.
PIPT	Physically indexed, physically tagged (cache).
PLL	Abbreviation for Phase-Locked Loop.
POR	Power-on reset.
POST	Abbreviation for power-on self-test.

prefetch-able	(1) An attribute of a memory location that indicates to an MMU that PREFETCH operations to that location may be applied. (2) A memory location condition for which the system designer has determined that no undesirable effects will occur if a PREFETCH operation to that location is allowed to succeed. Typically, normal memory is prefetchable. Nonprefetchable locations include those that, when read, change state or cause external events to occur. For example, some I/O devices are designed with registers that clear on read; others have registers that initiate operations when read. *See also* **side effect**.
privileged	An adjective that describes: (1) the state of the virtual processor when PSTATE.priv = 1 and HPSTATE.hpriv = 0, that is, the virtual processor is in privileged mode; (2) processor state that is only accessible to software while the virtual processor is in hyperprivileged or privileged mode; for example, privileged registers, privileged ASRs, or, in general, privileged state; (3) an instruction that can be executed only when the virtual processor is in hyperprivileged or privileged mode.
privileged mode	The mode in which a processor is operating when PSTATE.priv = 1 and HPSTATE.hpriv = 0. *See also* **nonprivileged** *and* **hyperprivileged**.
processor	The unit on which a shared interface is provided to control the configuration and execution of a collection of strands; a physical module that plugs into a system. *Synonym for* **processor module**. For a detailed definition of this term, see page 758 of the UltraSPARC Architecture manual. *See also* **pipeline**, **physical core**, **strand**, **thread**, *and* **virtual processor**.
processor core	Synonym for **physical core**.
processor module	Synonym for **processor**.
program counter	A register that contains the address of the instruction currently being executed.

Q

quadword	A 16-byte datum. **Note:** The definition of this term is architecture dependent and may be different from that used in other processor architectures.

R

R register	An integer register. Also called a general-purpose register or working register.
RA	Real address.
RAS	Abbreviation for Reliability, Availability, and Serviceability.
RAW	Abbreviation for Read After Write (hazard).
rd	Rounding direction for floating-point operations.
real address	An address produced by a virtual processor that refers to a particular software-visible memory location, as viewed from privileged mode. Virtual addresses are usually translated by a combination of hardware and software to real addresses, which can be used to access real memory. Real addresses, in turn, are usually translated to physical addresses, which can be used to access physical memory. *See also* **physical address** *and* **virtual address**.
recoverable	A term applied to an error when enough information exists elsewhere in the system for software to recover from an uncorrectable error. Examples include uncorrectable errors on clean L2 lines, which are recovered by software, invalidating the line, and initiating a refetch from memory. *See also* **unrecoverable**.
RED_ state	Reset, Error, and Debug state. The virtual processor state when HPSTATE.red = 1. A restricted execution environment used to process resets and traps that occur when TL = *MAXTL* – 1.

`RED_state` trap A trap processed in `RED_state`. *Contrast with* **normal trap**.

reserved Describing an instruction field, certain bit combinations within an instruction field, or a register field that is reserved for definition by future versions of the architecture.

A reserved instruction field must read as 0, unless the implementation supports extended instructions within the field. The behavior of an OpenSPARC virtual processor when it encounters a nonzero value in a reserved instruction field is as defined in Section 6.3.11, *Reserved Opcodes and Instruction Fields*, of the UltraSPARC Architecture manual.

A reserved bit combination within an instruction field is defined in Chapter 7, *Instructions*, of the UltraSPARC Architecture manual. In all cases, an OpenSPARC processor must decode and trap on such reserved bit combinations.

A reserved field within a register reads as 0 in current implementations and, when written by software, should always be written with values of that field previously read from that register or with the value zero (as described in Section 5.1, *Reserved Register Fields*, of the UltraSPARC Architecture manual).

Throughout this book, figures and tables illustrating registers and instruction encodings indicate reserved fields and reserved bit combinations with a wide ("em") dash (—).

reset trap A vectored transfer of control to hyperprivileged software through a fixed-address reset trap table. Reset traps cause entry into `RED_state`.

restricted Describes an address space identifier (ASI) that may be accessed only while the virtual processor is operating in privileged or hyperprivileged mode.

retired	An instruction is said to be “retired” when one of the following two events has occurred: (1) A precise trap has been taken, with TPC containing the instruction’s address (the instruction has not changed architectural state in this case). (2) The instruction’s execution has progressed to a point at which architectural state affected by the instruction has been updated such that all three of the following are true: • The PC has advanced beyond the instruction. • Except for deferred trap handlers, no consumer in the same instruction stream can see the old values and all consumers in the same instruction stream will see the new values. • Stores are visible to all loads in the same instruction stream, including stores to noncacheable locations.
RMO	Abbreviation for Relaxed Memory Order (a memory model).
RTO	Abbreviation for Read to Own (a type of transaction, used to request ownership of a cache line).
RTS	Abbreviation for Read to Share (a type of transaction, used to request read-only access to a cache line).
`running`	A state of a virtual processor in which it is in operation (maintaining cache coherency and issuing instructions for execution) and not `Parked`.

S

SAT	Abbreviation for stand-alone testbench.
SerDes	Abbreviation for Serializer/Deserializer.
service processor	A device external to the processor that can examine and alter internal processor state. A service processor may be used to control/coordinate a multiprocessor system and aid in error recovery.
SFSR	Synchronous Fault Status register.

shall	Synonym for **must**.
should	A keyword indicating flexibility of choice with a strongly preferred implementation. *Synonym for* **it is recommended**.
side effect	The result of a memory location having additional actions beyond the reading or writing of data. A side effect can occur when a memory operation on that location is allowed to succeed. Locations with side effects include those that, when accessed, change state or cause external events to occur. For example, some I/O devices contain registers that clear on read; others have registers that initiate operations when read. *See also* **prefetchable**.
SIMD	Abbreviation for Single Instruction/Multiple Data; a class of instructions that perform identical operations on multiple data contained (or "packed") in each source operand.
SIR	Abbreviation for software-initiated reset.
snooping	The process of maintaining coherency between caches in a shared-memory bus architecture. Each cache controller monitors (snoops) the bus to determine whether it needs to copy back or invalidate its copy of each shared cache block.
SoC	Abbreviation for server-on-a-chip or system-on-a-chip.
speculative load	A load operation that is issued by a virtual processor speculatively, that is, before it is known whether the load will be executed in the flow of the program. Speculative accesses are used by hardware to speed program execution and are transparent to code. An implementation, through a combination of hardware and system software, must nullify speculative loads on memory locations that have side effects; otherwise, such accesses produce unpredictable results. *Contrast with* **nonfaulting load**.

store	An instruction that writes (but does not explicitly read) memory or writes (but does not explicitly read) location(s) in an alternate address space. Some examples of *store* include stores from either integer or floating-point registers, block stores, Partial Store, and alternate address space variants of those instructions. *See also* **load** *and* **load-store**, the definitions of which are mutually exclusive with *store*.
strand	The hardware state that must be maintained in order to execute a software thread. For a detailed definition of this term, see page 757 of the UltraSPARC Architecture manual. *See also* **pipeline**, **physical core**, **processor**, **thread**, *and* **virtual processor**.
subnormal number	A nonzero floating-point number, the exponent of which has a value of zero. A more complete definition is provided in IEEE Standard 754-1985.
sun4v	An architected interface between privileged and hyperprivileged software, used for UltraSPARC Architecture processors. See the Hypervisor API specification for details.
superscalar	An implementation that allows several instructions to be issued, executed, and committed in one clock cycle.
supervisor software	Software that executes when the virtual processor is in privileged mode.
suspend	Synonym for **park**.
suspended	Synonym for **parked**.
synchroni-zation	An operation that causes the processor to wait until the effects of all previous instructions are completely visible before any subsequent instructions are executed.
system	A set of virtual processors that share a common hardware memory physical address space.

T

taken	Said of a control-transfer instruction (CTI) when the CTI writes the target address value into NPC. A trap is *taken* when the control flow changes in response to an exception, reset, Tcc instruction, or interrupt. An exception must be detected and recognized before it can cause a trap to be taken.
TBA	Abbreviation for trap base address.
thread	A software entity that can be executed on hardware. For a detailed definition of this term, see page 757 of the UltraSPARC Architecture manual. *See also* **pipeline**, **physical core**, **processor**, **strand**, *and* **virtual processor**.
TLB	Abbreviation for **Translation Lookaside Buffer.**
TLB hit	Said when the desired translation is present in the TLB.
TLB miss	Said when the desired translation is not present in the TLB.
TNPC	Abbreviation for trap-saved next program counter.
TPC	Abbreviation for trap-saved program counter.
Translation Lookaside Buffer	A cache within an MMU that contains recently used Translation Table Entries (TTEs). TLBs speed up translations by often eliminating the need to reread TTEs from memory.
trap	The action taken by a virtual processor when it changes the instruction flow in response to the presence of an exception, reset, a Tcc instruction, or an interrupt. The action is a vectored transfer of control to more-privileged software through a table, the address of which is specified by the privileged Trap Base Address (TBA) register or the Hyperprivileged Trap Base Address (HTBA) register. *See also* **exception**.
TSB	Translation storage buffer. A table of the address translations that is maintained by software in system memory and that serves as a cache of virtual-to-real address mappings.

TSO	Abbreviation for Total Store Order (a memory model).
TTE	Translation Table Entry. Describes the virtual-to-real, virtual-to-physical, or real-to-physical translation and page attributes for a specific page in the page table. In some cases, this term is explicitly used to refer to entries in the TSB.

U

unassigned	A value (for example, an ASI number), the semantics of which are not architecturally mandated and which may be determined independently by each implementation within any guidelines given.
uncorrect-able	A term applied to an error when not enough information accompanies the incorrect signal or datum to allow correction of the error, and it is not known by the error detector whether enough such information exists elsewhere in the system. Examples include uncorrectable errors on L2s. Uncorrectable errors can be further divided into two types: **recoverable** *and* **unrecoverable**. *See also* **correctable**.
undefined	An aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, nor make any assumptions about, an undefined feature or behavior. Use of such a feature can deliver unexpected results and may or may not cause a trap. An undefined feature may vary among implementations and may also vary over time on a given implementation. Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes (such as changing the privilege state or allowing circumvention of normal restrictions imposed by the privilege state), put a virtual processor into a more-privileged mode, or put the virtual processor into an unrecoverable state.
unimple-mented	An architectural feature that is not directly executed in hardware because it is optional or is emulated in software.
uniproces-sor system	A system containing a single virtual processor.

unpark	To bring a virtual processor out of suspension. There may be a delay until the virtual processor is unparked, but no heavyweight operation (such as a reset) is required to complete the unparking process. *See also* **disable** *and* **park**.
unparked	Synonym for **running**.
unpredict-able	Synonym for **undefined**.
unrecover-able	A term applied to an error when not enough information exists elsewhere in the system for software to recover from an uncorrectable error. Examples include uncorrectable errors on dirty L2 lines. *See also* **recoverable**.
unrestrict-ed	Describes an address space identifier (ASI) that can be used in all privileged modes; that is, regardless of the value of PSTATE.priv and HPSTATE.hpriv.
user application program	Synonym for **application program**.

V

VA	Abbreviation for **virtual address**.
virtual address	An address produced by a virtual processor that refers to a particular software-visible memory location. Virtual addresses usually are translated by a combination of hardware and software to physical addresses, which can be used to access physical memory. *See also* **physical address** and **real address**.
virtual core, virtual pro-cessor core	Synonyms for **virtual processor**.

virtual processor	The term *virtual processor*, or *virtual processor core*, is used to identify each processor in a processor. At any given time, an operating system can have a different thread scheduled on each virtual processor. For a detailed definition of this term, see page 758 of the UltraSPARC Architecture manual. *See also* **pipeline**, **physical core**, **processor**, **strand**, *and* **thread**.
VIS™	Abbreviation for Visual Instruction Set.
VP	Abbreviation for **virtual processor**.

W

WDR	Watchdog reset.
word	A 4-byte datum. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

X

XIR	Abbreviation for externally initiated reset.

Index

FIRST EDITION

OpenSPARC™ Sun microsystems

OpenSPARC™ Internals

OpenSPARC T1/T2 CHIP MULTITHREADED THROUGHPUT COMPUTING

The coverage is extremely broad and deep, from the basics of the OpenSPARC architecture, the rationale for throughput optimized microprocessor design, and the microarchitecture of the T1 and T2 implementations to a roadmap for using the T1 and T2 design database and design verification suites.

—Kunle Olukotun
Stanford University
Professor, Electrical Engineering & Computer Science
Founder, Afara Websystems
Director, Pervasive Parallelism Lab

OpenSPARC internals provides an in-depth explanation of UltraSPARC T1/T2 internal architecture. It also serves as a detailed reference for guiding the implementation of a chip multithreaded microprocessor or the development of a SoC based application system. This book plays a significant role in helping promote not only the application of UltraSPARC T1/T2 but also the research and development of a chip multithreaded microprocessor as well as its applications.

—Dongsheng Wang, Ph.D.
Tsinghua University, Beijing, China
Professor, Dept. of Computer Science
Director of Microprocessor and SoC Center

Like the open-source OpenSPARC T1 and T2 projects, OpenSPARC Internals delivers a comprehensive package. The book tells a complete story behind Sun's current flagship chip multithreaded (CMT) processors -- from the design theories and internals to the development tools and methodologies. The gook is especially indispensable to anyone interested in uncovering ways to take advantage of the open-source OpenSPARC projects.

— James C. Hoe
Carnegie Mellon University
Associate Professor, Electrical & Computer Engineering

This book provides a wealth of practical tips for getting started using OpenSPARC, and OpenSPARC provides a great design to take full advantage of modern FPGAs.

— Ivo Bolsens
Chief Technology Officer, Xilinx Corporation

About *OpenSPARC Internals*

OpenSPARC Internals was largely written by the team of OpenSPARC designers, developers, and programmers to acquaint readers with OpenSPARC and to guide users as they develop their own OpenSPARC designs. Here are some highlights of the book:

- How to customize and use OpenSPARC
- How to start using OpenSPARC code
- How to make basic changes including
  - *configuring number of cores or threads*
  - *paring to a smaller size*
  - *fitting on an FPGA*
  - *adding extensions*
- How to set up
  - *simulation environment*
  - *emulation environment*
- How to verify an OpenSPARC design

About OpenSPARC

OpenSPARC is the open-source form of the leading 32- and 64-thread microprocessors "T1" and "T2," which are modern multicore, chip multithreaded (CMT) designs. These designs can be configured and customized for imaginative commercial and academic applications. As open source, OpenSPARC is freely available for download on OpenSPARC.net.

ISBN 978-0-557-01974-8
90000 >
9 780557 019748

www.OpenSPARC.net