UltraSPARC ™ IIi

User's Manual

THE NETWORK IS THE COMPUTER™

Sun Microelectronics

901 San Antonio Road
Palo Alto, CA 94303 USA
800 681-8845
http://www.sun.com/microelectronics

Part No.: 805-0087-02

Contents

Figures

Tables

Revision History

Date	Version	Description of Change
July 1999	-02	coverage of SME1430 CPU added
October 1997	-01	initial release; describes SME1040 CPU

Preface

Overview

Welcome to the UltraSPARC IIi User's Manual. This book contains information about the architecture and programming of UltraSPARC IIi, one of Sun Microsystems' family of processors that are SPARC-V9-compliant as well as meeting the requirements of the PCI specification, version 2.1. This manual describes the UltraSPARC IIi processor implementation.

This book contains information on:

- The UltraSPARC IIi system architecture
- The components that make up an UltraSPARC IIi processor
- Memory and low-level system management, including detailed information needed by operating system programmers
- Extensions to and implementation-dependencies of the SPARC-V9 architecture
- Techniques for managing the pipeline and for producing optimized code
- Instruction set, instruction grouping rules for efficient execution, address space identifiers, and event ordering
- Data and address formats
- External interfaces and their support, including PCI, memory, and UPA64S
- Interrupts and traps
- Memory models
- Debug and diagnostic provisions, including performance instrumentation
- Power management
- Performance instrumentation and Boundary Scan (IEEE 1149) support
- Compatibility considerations with regard to prior processors

A Brief History of SPARC and PCI

SPARC stands for **S**calable **P**rocessor **ARC**hitecture, which was first announced in 1987. Unlike more traditional processor architectures, SPARC is an open standard, freely available through license from SPARC International, Inc. Any company that obtains a license can manufacture and sell a SPARC-compliant processor.

By the early 1990s SPARC processors were available from over a dozen different vendors, and over 8,000 SPARC-compliant applications had been certified.

In 1994, SPARC International, Inc. published *The SPARC Architecture Manual, Version 9*, which defined a powerful 64-bit enhancement to the SPARC architecture. SPARC-V9 provided support for:

- 64-bit virtual addresses and 64-bit integer data
- Fault tolerance
- Fast trap handling and context switching
- Big- and little-endian byte orders

UltraSPARC is the first family of SPARC-V9-compliant processors available from Sun Microsystems, Inc.

The Peripheral Component Interconnect (PCI) bus specification was first issued in June 1992 (at version 1.0) by the PCI Special Interest Group to define a high-performance bus for peripheral components. In 1993 they added a connector specification. The current version 2.1 document added a 66 MHz bus specification and was released in June, 1995.

The PCI Local Bus uses multiplexed address and data lines and is well suited for connecting large bandwidth peripheral components. It is used to interconnect highly-integrated peripheral-controller components, peripheral add-in boards, and processor and memory systems and offers the following advantages:

- Peripheral compatibility with existing drivers and application software
- 32-bit or 64-bit data bus width and 64-bit addressing are supported
- Synchronous Peripheral bus
- Processor-independent bus optimized for I/O functions
- Bus operation concurrent with processor subsystem
- Peripheral access from anywhere in memory or I/O space
- Peripheral latency minimized by efficient coupling with processor bus, cache, and memory
- 33 and 66 MHz bus clock specification
- PCI peripherals contain registers with information for their configuration

Sun provides the optional Advanced PCI Bridge (APB™) ASIC for an optimized PCI interface with the UltraSPARC IIi processor.

How to Use This Book

This book is a companion to *The SPARC Architecture Manual, Version 9*, which is available from many technical bookstores or directly from its copyright holder:

SPARC International, Inc.
535 Middlefield Road, Suite 210
Menlo Park, CA 94025
(415) 321-8692

The SPARC Architecture Manual, Version 9 provides a complete description of the SPARC-V9 architecture. Since SPARC-V9 is an open architecture, many of the implementation decisions have been left to the manufacturers of SPARC-compliant processors. These "implementation dependencies" are introduced in *The SPARC Architecture Manual, Version 9*.

This book, the UltraSPARC IIi User's Manual, describes the UltraSPARC IIi implementation of the SPARC-V9 architecture. It provides specific information about UltraSPARC IIi processors, including how each SPARC-V9 implementation dependency was resolved. (See Chapter 14, *Implementation Dependencies*" for specific information.) This manual also describes extensions to SPARC-V9 that are available (currently) only on UltraSPARC IIi processors.

A great deal of background information and a number of architectural concepts are not contained in this book. You will find cross references to *The SPARC Architecture Manual, Version 9* located throughout this book. You should have a copy of that book at hand whenever you are working with the UltraSPARC IIi User's Manual. For detailed information about the electrical and mechanical characteristics of the processor, including pin and pad assignments, consult the *UltraSPARC-IIi Data Sheet*. The section: *Bibliography* on page 465 describes how to obtain the data sheet.

The UltraSPARC IIi includes the SME1041 and SME1430 CPUs. The differences are noted below and in Chapter 18. Also see *Glossary* on page 459.

Textual Conventions

This book uses the same textual conventions as *The SPARC Architecture Manual, Version 9*. They are summarized here for convenience.

Fonts are used as follows:

- Italic font is used for register names, instruction fields, and read-only register fields.
- `courier` font is used for literals and software examples.
- **Bold** font is used for emphasis.
- UPPER CASE items are acronyms, instruction names, or writable register fields.
- *Italic sans serif* font is used for exception and trap names.
- Underbar characters (_) join words in register, register field, exception, and trap names. Such words can be split across lines at the underbar without an intervening hyphen.
- The following notational conventions are used:
- Square brackets '[ ]' indicate a numbered register in a register file.
- Angle brackets '< >' indicate a bit number or colon-separated range of bit numbers within a field.
- Curly braces '{ }' are used to indicate textual substitution.
- The □ symbol designates concatenation of bit vectors. A comma '**,**' on the left side of an assignment separates quantities that are concatenated for the purpose of assignment.

Contents

This manual has the following organization:

The initial part of this book gives an overview of the UltraSPARC IIi and contains the following chapters:

- Chapter 1, *UltraSPARC IIi Basics*, describes the architecture in general terms and introduces its components.
- Chapter 2, *Processor Pipeline*, describes UltraSPARC IIi's 9-stage pipeline.
- Chapter 3, *Cache Organization*, describes the UltraSPARC IIi caches.
- Chapter 4, *Overview of I and D-MMUs*, describes the UltraSPARC IIi MMU, its architecture, how it performs virtual address translation, and how it is programmed.
- Chapter 5, *UltraSPARC IIi in a System*, briefly describes the UltraSPARC IIi configuration.
- Chapter 6, *Address Spaces, ASIs, ASRs, and Traps* discusses physical and virtual address space mapping and identifiers. It lists address and port assignments, including those for PCI, and also gives memory DIMM requirements.
- Chapter 7, *UltraSPARC IIi Memory System*, discusses DRAM memory hardware structure, selection, and addressing.

- Chapter 8, *Cache and Memory Interactions*, deals with the requirements to preserve data integrity during cache and memory operations and describes instructions used in these cases.
- Chapter 9, *PCI Bus Interface*, describes the PCI Bus Interface Module of UltraSPARC IIi which is a host PCI bridge.
- Chapter 10, *UltraSPARC IIi IOM*, details the IO Memory Management Unit (IOM), which performs virtual to physical address translation.
- Chapter 11, *Interrupt Handling*, describes how UltraSPARC IIi processes interrupts.
- Chapter 12, *Instruction Set Summary*, provides a list of all supported instructions, including SPARC-V9 core instructions and UltraSPARC IIi extensions.
- Chapter 13, *VIS™ and Additional Instructions*, contains detailed documentation of the extended instructions that UltraSPARC IIi adds to the SPARC-V9 instruction set, including those relating to power management, graphics, and memory-access and control.
- Chapter 14, *Implementation Dependencies*, discusses how UltraSPARC IIi resolves each of the implementation-dependencies defined by the SPARC-V9 architecture.

The latter part of the book presents detailed information about UltraSPARC IIi architecture and programming. This section contains the following chapters:

- Chapter 15, *MMU Internal Architecture*
- Chapter 16, *Error Handling*, discusses how UltraSPARC IIi handles system errors and describes the available error status registers.
- Chapter 17, *Reset and RED_state*, describes how UltraSPARC IIi handles the various SPARC-V9 reset conditions, and how it implements RED_state.
- Chapter 18, *MCU Control and Status Registers*,
- Chapter 19, *UltraSPARC IIi PCI Control and Status*,
- Chapter 20, *SPARC-V9 Memory Models*, describes the supported memory models (which are documented fully in *The SPARC Architecture Manual, Version 9*. Low-level programmers and operating system implementors should study this chapter to understand how their code will interact with the UltraSPARC IIi cache and memory systems.
- Chapter 21, *Code Generation Guidelines*, contains detailed information about generating optimum UltraSPARC IIi code.
- Chapter 22, *Grouping Rules and Stalls*, describes instruction interdependencies and optimal instruction ordering.
- Appendices contain low-level technical material or information not needed for a general understanding of the architecture. The manual contains the following appendices:
- Appendix A, *Debug and Diagnostics Support*, describes diagnostics registers and capabilities.
- Appendix B, *Performance Instrumentation*, describes built-in capabilities to measure UltraSPARC IIi performance.
- Appendix C, *IEEE 1149.1 Scan Interface*, contains information about the diagnostic boundary-scan interface for UltraSPARC IIi.

- Appendix D, *ECC Specification*, details the specification for the error correcting code (ECC) used in transactions between processor and DRAMs
- Appendix E, *UPA64S interface*, describes transactions and data format on the MEMDATA bus.
- Appendix F, *Pin and Signal Descriptions*, contains general information about the pins and signals of the UltraSPARC IIi and its components.
- Appendix G, *ASI Names*, contains an alphabetical listing of the names and suggested macro syntax for all supported ASIs.
- Appendix H, *Event Ordering on UltraSPARC IIi* discusses ordering of load and store operations.
- Appendix I, *Observability Bus* describes this bus that can help bring up the processor and provide performance monitoring.
- Appendix J, *List of Compatibility Notes*, provides a reference list of the compatibility notes from the various chapters of the text.
- Appendix K, *Errata*, lists errata for the UltraSPARC IIi.

A Glossary, Bibliography, and Index complete the book.

CHAPTER 1

UltraSPARC IIi Basics

1.1 Overview

The UltraSPARC IIi CPU is a high-performance, highly integrated superscalar processor implementing the 64-bit SPARC-V9 RISC architecture that also includes on-chip memory and I/O control. It supports Sun's popular Solaris operating system and is binary-compatible with all ultraSPARC software.

Each functional area on the UltraSPARC-II*i* maintains decentralized control, allowing many activities to overlap. The design supports the following features:

- Sustained issue of up to four instructions per cycle (even in the presence of conditional branches and cache misses) with a decoupled Prefetch and Dispatch Unit.
- Load buffers on the input side of the Execution Unit, together with store buffers on the output side, decouple pipeline execution from data cache misses.
- Instructions are issued in program order to multiple functional units.
- Instructions execute in parallel and may complete out of order.
- Instructions from two basic blocks (that is, instructions before and after a conditional branch) can be issued in the same group.
- Separate Memory Control and PCI I/O interface units also decouple their related key activities from the instruction pipeline.

UltraSPARC IIi includes a full implementation of the 64-bit SPARC-V9 architecture. It supports a 44-bit virtual address space and a 41-bit physical address space with 64-bit address pointers. The core instruction set is extended to include the VIS instruction set—graphics instructions that provide the most common operations related to two-dimensional image processing, two and three-dimensional graphics and image compression algorithms, and parallel operations on pixel data with 8- and 16-bit components. Support for high bandwidth memory to memory transfers also provided through 64-byte block load and block store instructions.

1.2 Design Philosophy

The execution time of an application is the product of three factors: the number of instructions generated by the compiler, the average number of cycles required per instruction, and the cycle time of the processor. The architecture and implementation of UltraSPARC IIi, coupled with new compiler techniques, makes it possible to reduce each component while not deteriorating the other two.

The number of instructions for a given task depends on the instruction set and on compiler optimizations (dead code elimination, constant propagation, profiling for code motion, and so on). Since it is based on the SPARC-V9 architecture, UltraSPARC IIi offers features that can help reduce the total instruction count:

- 64-bit integer processing
- Additional floating-point registers (beyond the number offered in SPARC-V8) that can be used to eliminate floating-point loads and stores
- Enhanced trap model with alternate global registers

The average number of cycles per instruction (CPI) depends on the architecture of the processor and on the ability of the compiler to take advantage of the hardware features offered. The UltraSPARC IIi execution units (ALUs, LD/ST, branch, two floating-point, and two graphics) allow the CPI to be as low as 0.25 (four instructions per cycle). To support this high execution bandwidth, sophisticated hardware is provided to supply:

1. Up to four instructions per cycle, even in the presence of conditional branches

2. Data at a rate of eight bytes per two cycles from the external cache to the data cache, and eight bytes per cycle into the register files.

To reduce instruction dependency stalls, UltraSPARC IIi has short latency operations and provides direct bypassing between units or within the same unit. The impact of cache misses, usually a large contributor to the CPI, is reduced significantly through the use of decoupled units: (prefetch unit, load buffer, store buffer, and memory control) that operate asynchronously with the rest of the pipeline.

The Memory Control Unit (MCU) is responsible for DRAM and UPA64S control which is accomplished in synchronism with the processor clock. The DRAM interface is expanded from 64 + 8 ECC bits to 128 + 16 ECC bits by means of external data transceivers. This configuration maximizes the EDO CAS cycle rate. The MCU specification is wide enough to embrace all major vendors' DRAM specifications.

Other features such as a fully pipelined interface to the external cache (E-Cache) and support for speculative loads, coupled with sophisticated compiler techniques such as software pipelining and cross-block scheduling also reduce the CPI significantly.

The PCI Bus Module (PBM) provides a direct interface with a 32-bit PCI bus that meets PCI specification version 2.1. This module is internally linked with the External Cache Unit (ECU) and the IOM.

The IO Memory Management Unit (IOM) manages virtual to physical memory address mapping using a 16-entry Translation Lookaside Buffer (TLB) in conjunction with a large Translation Storage Buffer (TSB) in memory.

The PCI bus can run at 66 MHz or at 33 MHz. Up to four Advanced PCI Bridge ASICs (APB)s may be used with the UltraSPARC IIi, each of which can support up to two 33 MHz secondary PCI busses. PCI DMA transfers are cache-coherent.

A balanced architecture must be able to provide a low CPI without affecting the cycle time. Several of UltraSPARC IIi's architectural features, coupled with an aggressive implementation and state-of-the-art technology, make it possible to achieve a short cycle time (see *Table 1-1*). The pipeline is organized so that large scalarity (four), short latencies, and multiple bypasses do not affect the cycle time significantly.

1.3 Component Description

Figure 1-1 shows a block diagram that illustrates the components of the UltraSPARC IIi processor. In a single-chip implementation, UltraSPARC IIi integrates these components:

- Independently clocked (132 MHz internal, 66 or 33 MHz external) PCI interfaces, fully decoupled from the main CPU
- PCI bus module (PBM)
- PCI I/O memory management unit (IOM) with 16 entries for incoming I/O to physical mapping/protection
- External (E-cache) cache control unit (ECU)
- Memory controller unit (MCU), operates both the 144-bit-wide DRAM subsystem and the UPA64S interface
- 16-Kilobyte instruction cache (I-Cache)
- 16-Kilobyte data cache (D-cache)
- Prefetch, branch prediction and dispatch unit (PDU) containing grouping logic and an instruction buffer
- A 64-entry instruction translation lookaside buffer (iTLB) and a 64-entry data translation lookaside buffer (dTLB)
- Integer execution unit (IEU) with two arithmetic logic units (ALUs)
- Floating-point unit (FPU) with independent add, multiply and divide/square root sub-units
- Graphics unit (GRU) composed of two independent execution pipelines

- Load buffer and store buffer unit (LSU), decoupling data accesses from the pipeline

Figure 1-1 UltraSPARC IIi Block Diagram

1.3.1 PCI Bus Module (PBM)

The PBM interfaces UltraSPARC IIi directly with a 32-bit PCI bus, compliant to the PCI specification, revision 2.1. The PCI bus runs at speeds up to 66 MHz, typically 33 and 66 MHz. The PBM is optimized for 16-, 32- and 64-byte transfers, and can support up to four PCI bus masters. The module also queues pending interrupts received from the interrupt concentrator (or RIC--SME2210) chip or programmable logic device (PLD).

The entire PCI address space is noncacheable for CPU references, but coherent DMA is supported. (This means that all writes to memory from PCI, and reads from memory, are cache coherent.) Interrupt handling is synchronized to the completion of all prior DMA writes. The PCI data path is illustrated in *Figure 1-2*.

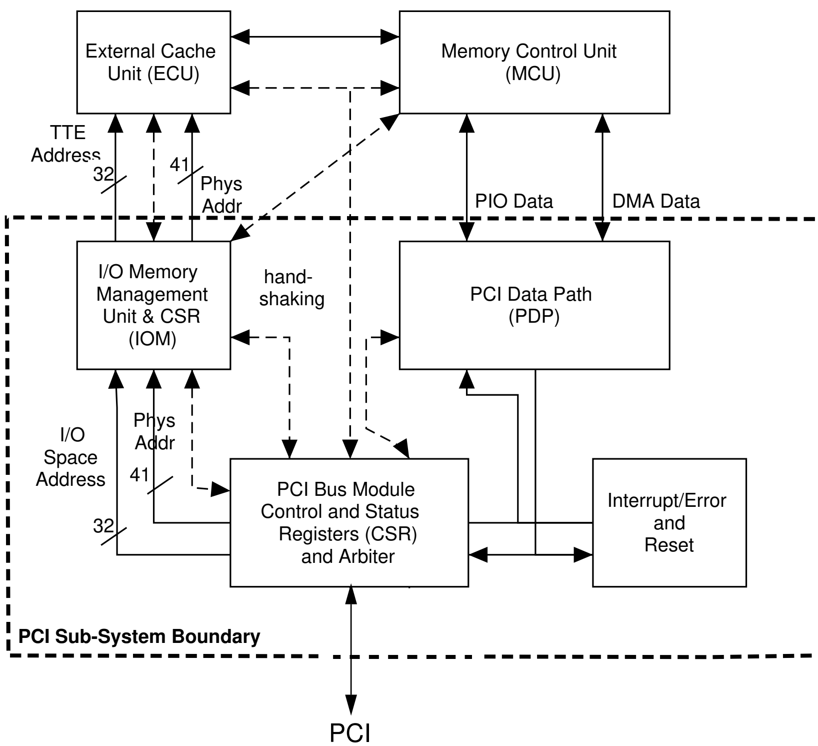


Figure 1-2 UltraSPARC IIi PCI and MCU Subsystems

1.3.2 IO Memory Management Unit (IOM)

The IOM performs address translations from 32-bit DVMA to 34-bit physical addresses when UltraSPARC IIi is a PCI target (when DVMA read/write access is required). The IOM uses a fully associative 16-entry TLB (translation lookaside buffer). In the case of a TLB miss, the IOM performs a single-level hardware tablewalk into the large TSB (translation storage buffer) in memory.

1.3.3 External Cache Control Unit (ECU)

The main role of the ECU is to handle I-cache and D-Cache misses efficiently. The ECU can handle one access every other cycle to the external cache. Loads that miss in the D-cache cause 16-byte D-cache fills using two consecutive 8-byte accesses to the E-cache. Stores are writethrough to the E-cache and are fully pipelined. Instruction prefetches that miss the I-cache cause 32-byte I-cache fills using four consecutive 8-byte accesses to the E-cache. The E-cache is parity-protected.

In addition, the ECU supports DMA accesses which hit in the external cache and maintains data coherency between the external cache and the main memory. The size of the external cache can be 256 kB, 512 kB, 1 MB, or 2 MB (where the line size is always 64 bytes). Cache lines have only 3 states: modified, exclusive and invalid.

The combination of the load buffer and the ECU is fully pipelined. For programs with large data sets, instructions are scheduled with load latencies based on the E-Cache latency, so the E-cache acts like a large primary cache. Floating-point applications use this feature to effectively “hide” D-Cache misses. Coherency is maintained between all caches and external PCI DMA references.

The ECU overlaps processing during load and store misses. Stores that hit the E-Cache can proceed while a load miss is being processed. The ECU is also capable of processing reads and writes without a costly turnaround penalty on the bidirectional E-cache data bus.

Block loads and block stores (these load or store a 64-byte line of data from memory or E-cache to the floating-point register file) provide high transfer bandwidth. By not installing into the E-cache on miss, they avoid polluting the cache with data that is only touched once.

The ECU also provides support for multiple outstanding data transfer requests to the MCU and PBM.

1.3.3.1 E-Cache SRAM Modes

The UltraSPARC IIi supports two alternative E-cache SRAM configurations that have particular operational modes:

- 2−2−2 (Pipelined) mode and
- 2−2 (Register-Latched) mode

In 2−2−2 (Pipelined) mode the E-cache SRAMs have a cycle time equal to half the processor cycle time. The name "2–2–2" indicates that it takes two processor clocks to send the address, two to access the SRAM array, and two to return the E-Cache data. 2–2–2 mode has a 6 cycle pin-to-pin latency and provides the least expensive SRAM solution at a given frequency.

In 2−2 (Register-Latched) mode the E-cache SRAMs also have a cycle time equal to half of the processor cycle time. The name "2–2" indicates that it takes two processor clocks to send the address and two clocks to access and return the E-Cache data. 2–2 mode has a 4 cycle pin-to-pin latency, which provides lower E-Cache latency. In addition, no dead cycles are necessary when alternating between reads and writes because of tighter control over turn on and turn off times in these SRAMs.

1.3.4 Memory Controller Unit (MCU)

All transactions to the DRAM and UPA64S subsystems are handled by the MCU. The external pins controlled by the MCU operate at divisions of the processor clock:

The CPU to UPA clock ratio is fixed by design. In the SME1040CGA the ratio is always 3:1. In SME1430LGA, the ratio is 4:1. The data transfer rate through the DRAM transceivers is programmable but typically occurs at 1/4 of the processor clock rate. Other options are 1/3 or 1/5 of the processor clock rate.

External data transceivers allow the DRAM data to be twice as wide as the processor's MEMDATA pins, so the EDO CAS cycle is 26.5 ns at 300 MHz. The MCU supports a composite DRAM specification which is a superset of 60 ns EDO DRAM specifications from all major vendors. These transceivers are commodity parts available from Texas Instruments. Use of faster DRAMs allow performance higher than quoted, as the various components of memory delay are programmable. A typical memory configuration is shown in *Figure 1-3*.

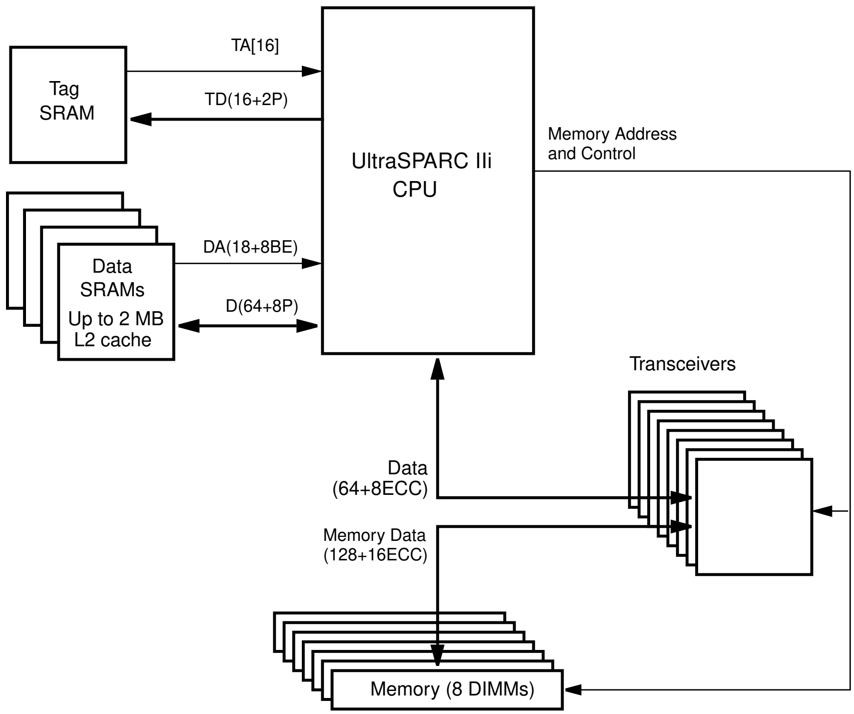


Figure 1-3 UltraSPARC IIi Memory—Typical Configuration

1.3.5 Instruction Cache (I-cache)

The I-cache is a 16 Kilobyte two-way set-associative cache with 32-byte blocks. The cache is physically indexed and physically tagged. The set is predicted as part of the "next field" so that only the index bits of an address are necessary to address the cache. (This means only 13 bits, which matches the minimum page size.) The instruction cache returns up to 4 instructions from a line that is 8 instructions wide.

1.3.6 Data Cache (D-cache)

The data cache is a write-through non-allocating 16 Kilobyte direct-mapped cache with two 16-byte sublocks per line. It is virtually indexed and physically tagged. The tag array is dual-ported so that tag updates due to line fills do not collide with tag reads for incoming loads. Snoops to the D-Cache use the second tag port so that an incoming load can proceed without being held up by a snoop.

1.3.7 Prefetch and Dispatch Unit (PDU)

The PDU fetches instructions before they are needed in the pipeline, so that the execution units do not starve for instructions. Instructions can be prefetched from all levels of the memory hierarchy, including the instruction cache, the external cache and the main memory. To prefetch across conditional branches, a dynamic branch prediction scheme is implemented in hardware, based on a two-bit history of the branch. A "next field" associated with every four instructions in the I-Cache points to the next I-Cache line to be fetched. This makes it possible to follow taken branches and provides the same instruction bandwidth achieved during sequential code. Up to 12 prefetched instructions are stored in the instruction buffer sent to the rest of the pipeline.

1.3.8 Translation Lookaside Buffers (iTLB and dTLB)

The Translation Lookaside Buffers provide mapping between 44-bit virtual addresses and 34-bit physical addresses. A 64-entry iTLB is used for instructions and a 64-entry dTLB for data, and both are fully associative. UltraSPARC IIi provides hardware support for a software-based TLB miss strategy. A trap to special software handlers installs new entries, typically with a latency of the order of an E-cache miss. A separate set of global registers is available whenever such a trap is encountered, for low latency miss handling. Page sizes of 8 kB, 64 kB, and 512 kB and 4 MB are supported.

1.3.9 Integer Execution Unit (IEU)

The IEU contains the following components:

- Two ALUs
- A multi-cycle integer multiplier
- A multi-cycle integer divider
- Eight register windows
- Four sets of global registers (normal, alternate, MMU, and interrupt globals)
- The trap registers (See *Table 1-1* for supported trap levels)

Table 1-1 shows that UltraSPARC IIi supports one more than the four trap levels mandated by the SPARC Version 9 specification

Table 1-1 Supported Trap Levels

	UltraSPARC IIi
MAXTL	4
Trap Levels	5

1.3.10 Floating-Point Unit (FPU)

The separation of the execution units in the FPU allows UltraSPARC IIi to issue and execute two floating-point instructions per cycle. Source data and results data are stored in the 32-entry register file, where each entry can contain a 32- or 64-bit value. Most instructions are fully pipelined (throughput of one per cycle), have a latency of three, and are not affected by the precision of the operands (same latency for single or double precision).

The divide and square-root instructions are not pipelined. These take 12 cycles (single precision) and 22 cycles (double precision) to execute, but they do not stall the processor. Other instructions, following the divide/square root can be issued, executed, and retired to the register file before the divide/square root finishes. A precise exception model is maintained by synchronizing the floating-point pipe with the integer pipe and by predicting traps for long-latency operations.

1.3.11 Graphics Unit (GRU)

UltraSPARC IIi includes a comprehensive set of graphics instructions (VIS) that provide industry-leading support for two-dimensional and three-dimensional image and video processing, image compression, audio processing, and similar functions. Sixteen-bit and 32-bit partitioned add, boolean and compare are provided. Eight-bit and 16-bit partitioned multiplies are supported. Single cycle pixel distance, data alignment, packing and merge operations are all supported in the GRU. The GRU may also be referred to as the VIS Instruction Unit (VIU).

1.3.12 Load/Store Unit (LSU)

The LSU is responsible for generating the virtual address of all loads and stores (including atomics and ASI loads), for accessing the data cache, for decoupling load misses from the pipeline through the load buffer, and for decoupling the stores through a store buffer. One load or one store can be issued per cycle. The store buffer can compress (or gather) multiple stores to the same 8 bytes into a single E-cache access. The UPA64S and PCI control units can compress sequential 8-byte stores into burst transactions, to improve noncacheable store bandwidth.

1.3.13 Phase Locked Loops (PLLs)

To minimize the clock skew at the system level UltraSPARC IIi has PLLs for both the processor clock and the PCI clock. The internal PCI clock runs at twice the speed of the PCI interface clock. For details, see Appendix F, *Pin and Signal Descriptions*."

1.3.14 Signals

All external cache signals are 2.6 V and exist only on the processor module. All other signals are 3.3V LVTTL. The highest frequency signal that comes from the module to the motherboard is 75 MHz. (unless the 100 MHz UPA64S interface is used). This allows cost savings in motherboard design.

Figure 1-3 on page 8 shows an UltraSPARC IIi subsystem, which consists of the UltraSPARC IIi processor and synchronous SRAM components for the E-cache tags and data.

CHAPTER 2

Processor Pipeline

2.1 Introductions

The UltraSPARC IIi processor contains a nine-stage pipeline. Most instructions go through the pipeline in exactly 9 stages. The instructions are considered terminated after they go through the last stage (W), after which changes to the processor architectural state are irreversible. *Figure 2-1* shows a simplified diagram of the integer and floating-point pipeline stages.

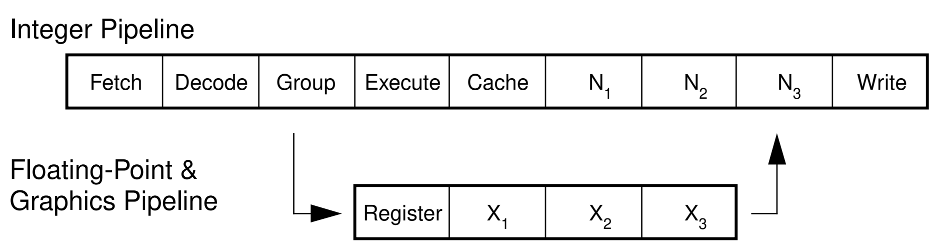


Figure 2-1 UltraSPARC IIi Pipeline Stages (Simplified)

Three additional stages are added to the integer pipeline to make it symmetrical with the floating-point pipeline. This simplifies pipeline synchronization and exception handling. It also eliminates the need to implement a floating-point queue.

Floating-point instructions with a latency greater than three (divide, square root, and inverse square root) behave differently than other instructions; the pipe is "extended" when the instruction reaches stage N_1. See Chapter 21, *Code Generation Guidelines*" for more information. Memory operations are allowed to proceed asynchronously with the pipeline in order to support latencies longer than the latency of the on-chip D-cache.

2.2 Pipeline Stages

This section describes each pipeline stage in detail. *Figure 2-2* illustrates the pipeline stages.

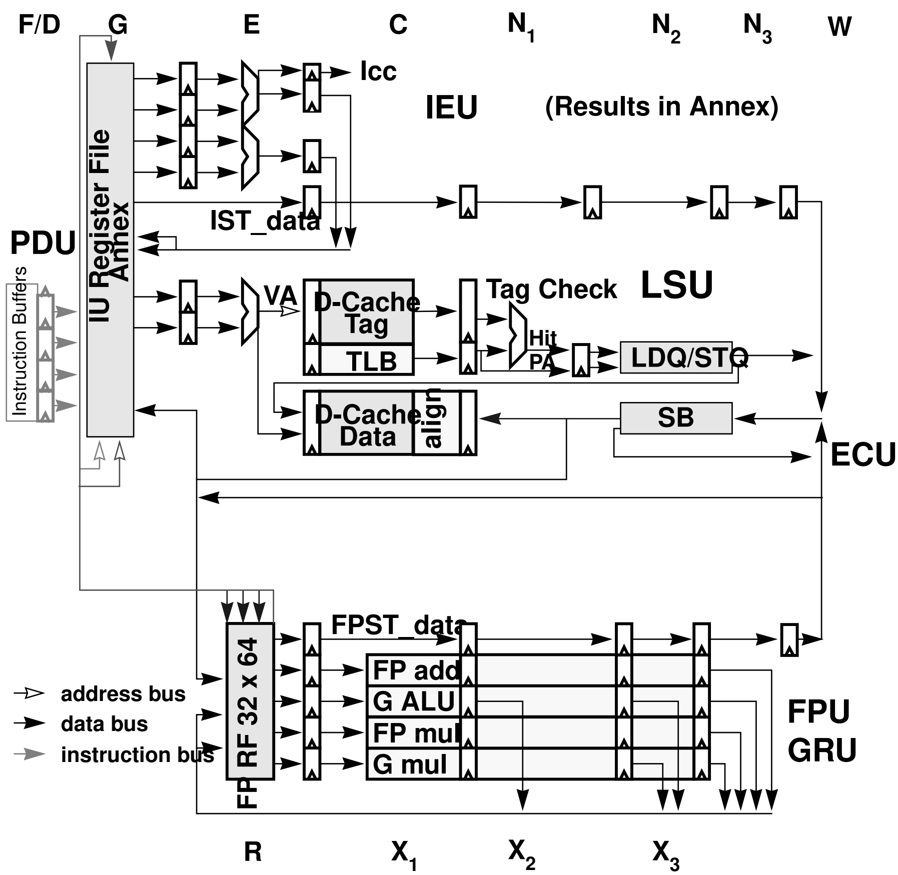


Figure 2-2 UltraSPARC IIi Pipeline Stages (Detail)

2.2.1 Stage 1: Fetch (F) Stage

Prior to their execution, instructions are fetched from the Instruction Cache (I-cache) and placed in the Instruction Buffer, where eventually they are selected to be executed. Accessing the I-cache is done during the F Stage. Up to four instructions are fetched along with branch prediction information, the predicted target address of a branch, and the predicted set of the target. The high bandwidth provided by the I-cache (4 instructions/cycle) allows the UltraSPARC IIi CPU to prefetch instructions ahead of time based on the current instruction flow and on branch prediction. Providing a fetch bandwidth greater than or equal to the maximum execution bandwidth assures that, for well behaved code, the processor does not starve for instructions. Exceptions to this rule occur when branches are hard to predict, when branches are very close to each other, or when the I-cache miss rate is high.

2.2.2 Stage 2: Decode (D) Stage

After being fetched, instructions are pre-decoded and then sent to the Instruction Buffer. The pre-decoded bits generated during this stage accompany the instructions during their stay in the Instruction Buffer. Upon reaching the next stage (where the grouping logic lives) these bits speed up the parallel decoding of up to four instructions.

While it is being filled, the Instruction Buffer also presents up to 4 instructions to the next stage. A pair of pointers manage the Instruction Buffer, ensuring that as many instructions as possible are presented *in order* to the next stage.

2.2.3 Stage 3: Grouping (G) Stage

The G-stage logic's main task is to group and dispatch a maximum of four valid instructions in one cycle. It receives a maximum of four valid instructions from the Prefetch and Dispatch Unit (PDU), it controls the Integer Core Register File (ICRF), and it routes valid data to each integer functional unit. The G-stage sends up to two floating-point or graphics instructions out of the four candidates to the Floating-Point and Graphics Unit (FGU). The G-stage logic is responsible for comparing register addresses for integer data bypassing and for handling pipeline stalls due to interlocks.

2.2.4 Stage 4: Execution (E) Stage

Data from the integer register file is processed by the two integer ALUs during this cycle (if the instruction group includes ALU operations). Results are computed and are available for other instructions (through bypasses) in the very next cycle. The virtual address of a memory operation is also calculated during the E Stage, in parallel with ALU computation.

FLOATING-POINT AND GRAPHICS UNIT: The Register (R) Stage of the FGU. The floating-point register file is accessed during this cycle. The instructions are also further decoded and the FGU control unit selects the proper bypasses for the current instructions.

2.2.5 Stage 5: Cache Access (C) Stage

The virtual address of memory operations calculated in the E-stage is sent to the tag RAM to determine if the access (load or store type) is a hit or a miss in the D-cache. In parallel the virtual address is sent to the data MMU to be translated into a physical address. On a load when there are no other outstanding loads, the data array is accessed so that the data can be forwarded to dependent instructions in the pipeline as soon as possible.

ALU operations executed in the E-stage generate condition codes in the C Stage. The condition codes are sent to the PDU, which checks whether a conditional branch in the group was correctly predicted. If the branch was mispredicted, earlier instructions in the pipe are flushed and the correct instructions are fetched. The results of ALU operations are not modified after the E Stage; the data merely propagates down the pipeline (through the annex register file), where it is available for bypassing for subsequent operations.

FLOATING-POINT AND GRAPHICS UNIT: The X_1 Stage of the FGU. Floating-point and graphics instructions start their execution during this stage. Instructions of latency one also finish their execution phase during the X_1 Stage.

2.2.6 Stage 6: N_1 Stage

A data cache (D-cache) miss/hit or a TLB miss/hit is determined during the N_1 Stage. If a load misses the D-cache, it enters the Load Buffer. The access will arbitrate for the E-cache if there are no older unissued loads. If a TLB miss is detected, a trap will be taken and the address translation is obtained through a software routine.

The physical address of a store is sent to the Store Buffer during this stage. To avoid pipeline stalls when store data is not immediately available, the store address and data parts are decoupled and sent to the Store Buffer separately.

FLOATING-POINT AND GRAPHICS UNIT: The X_2 stage of the FGU. Execution continues for most operations.

2.2.7 Stage 7: N_2 Stage

Most floating-point instructions finish their execution during this stage. After N_2, data can be bypassed to other stages or forwarded to the data portion of the Store Buffer. All loads that have entered the Load Buffer in N_1 continue their progress through the buffer; they will reappear in the pipeline only when the data comes back. Normal dependency checking is performed on all loads, including those in the load buffer.

FLOATING-POINT AND GRAPHICS UNIT: The X_3 stage of the FGU.

2.2.8 Stage 8: N_3 Stage

UltraSPARC IIi resolves traps at this stage.

2.2.9 Stage 9: Write (W) Stage

All results are written to the register files (integer and floating-point) during this stage. All actions performed during this stage are irreversible. After this stage, instructions are considered terminated.

CHAPTER 3

Cache Organization

3.1 Introduction

3.1.1 Level-1 Caches

The UltraSPARC IIi Level-1 D-cache is virtually indexed, physically tagged (VIPT). Virtual addresses are used to index into the D-cache tag and data arrays while accessing the D-MMU (that is, the dTLB). The resulting tag is compared against the translated physical address to determine D-cache hits.

A side-effect inherent in a virtual-indexed cache is *address aliasing*; this issue is addressed in Section 8.2.1, *Address Aliasing Flushing* on page 66.

The UltraSPARC IIi Level-1 I-cache is physically indexed, physically tagged (PIPT). The lowest 13 bits of instruction addresses are used to index into the I-cache tag and data arrays while accessing the I-MMU (that is, the iTLB). The resulting tag is compared against the translated physical address to determine I-cache hits.

3.1.1.1 Instruction Cache (I-cache)

The I-cache is a 16-kilobit pseudo-two-way set-associative cache with 32-byte blocks. The set is predicted based on the next fetch address; thus, only the index bits of an address are necessary to address the cache (that is, the lowest 13 bits, which matches the minimum page size of 8 kilobit). Instruction fetches bypass the instruction cache under the following conditions:

- When the I-cache enable or I-MMU enable bits in the LSU_Control_Register are clear (see Section A.6, *LSU_Control_Register* on page 370)

- When the processor is in RED_state, or
- When the I-MMU maps the fetch as noncacheable

The instruction cache snoops stores from DMA transfers, but it is not updated by stores, except for block commit stores (see Section 13.5.3, *Block Load and Store Instructions* on page 164). The FLUSH instruction can be used to maintain coherency. Block commit stores invalidate I-cache but do not flush instructions that have already been prefetched into the pipeline. A FLUSH, DONE, or RETRY instruction can be used to flush the pipeline. For block copies that must maintain I-cache coherency, it is more efficient to use block commit stores in the loop, followed by a single FLUSH instruction to flush the pipeline.

Note – The size of each I-cache set is the same as the page size in UltraSPARC IIi; thus, the virtual index bits equal the physical index bits.

3.1.1.2 Data Cache (D-cache)

The D-cache is a write-through, nonallocating-on-write-miss, 16-kb direct mapped cache with two 16-byte sub-blocks per line. Data accesses bypass the data cache when the D-cache enable bit in the LSU_Control_Register is clear (see Section A.6, *LSU_Control_Register* on page 370). Load misses will not allocate in the D-cache if the D-MMU enable bit in the LSU_Control_Register is clear or the access is mapped by the D-MMU as virtual noncacheable.

Note – A noncacheable access may access data in the D-cache from an earlier cacheable access to the same physical block, unless the D-cache is disabled. Software must flush the D-cache when changing a physical page from cacheable to noncacheable (see Section 8.2, *Cache Flushing*). In UltraSPARC IIi, the noncacheable accesses must follow the physical address space definition, so that this issue should not occur.

3.1.2 Level-2 PIPT External Cache (E-cache)

The UltraSPARC IIi E-cache (also known as level-2 cache) is physically indexed, physically tagged (PIPT). This cache has no virtual address or context information. The operating system needs no knowledge of such caches after initialization, except for stable storage management and error handling.

Memory accesses must be cacheable in the E-cache. As a result, there is no E-cache enable bit in the LSU_Control_Register.

Instruction fetches are directed to noncacheable PCI or UPA64s space when:

- The I-MMU is disabled, or
- The processor is in RED_state, or
- The access is mapped by the I-MMU as physically noncacheable

Data accesses are to noncacheable PCI or UPA64s space when:

- The D-MMU enable bit (DM) in the LSU_Control_Register is clear, or
- The access is mapped by the D-MMU as nonphysical cacheable (unless ASI_PHYS_USE_EC is used)

Note – When noncacheable accesses are used, the associated addresses must be legal according to the physical address map in *Table 6-1* on page 36.

The system must provide a noncacheable, ECC-less scratch memory for use of the booting code until the MMUs are enabled.

The E-cache is a unified, write-back, allocating, direct-mapped cache. The E-cache always includes the contents of the I-cache and D-cache. The E-cache size can range from 256 kB to 2 MB with a line size is 64 bytes. See *Table 1-1* on page 10.

Block loads and block stores, which load or store a 64-byte line of data from memory to the floating-point register file, do not allocate into the E-cache, to avoid pollution.

CHAPTER **4**

Overview of I and D-MMUs

4.1 Introduction

Instruction and Data MMUs are similar and are generically referred to as "MMU."

This chapter describes the UltraSPARC IIi Memory Management Unit as it is seen by the operating system software. The UltraSPARC IIi MMU conforms to the requirements set forth in *The SPARC Architecture Manual, Version 9*.

Note – The UltraSPARC IIi MMU does not conform to the SPARC-V8 Reference MMU Specification. In particular, the UltraSPARC IIi MMU supports a 44-bit virtual address space, software TLB miss processing only (no hardware page table walk), simplified protection encoding, and multiple page sizes. All of these differ from features required of SPARC-V8 Reference MMUs.

4.2 Virtual Address Translation

The UltraSPARC IIi MMU supports four page sizes: 8 kB, 64 kB, 512 kB, and 4 MB. It supports a 44-bit virtual address space, with 41 bits of physical address. During each processor cycle the UltraSPARC IIi MMU provides one instruction and one data virtual-to-physical address translation. In each translation, the virtual page number is replaced by a physical page number, which is concatenated with the page offset to form the full physical address, as illustrated in *Figure 4-1* on page 24. (This figure shows the full 64-bit virtual address, even though UltraSPARC IIi supports only 44 bits of VA.)

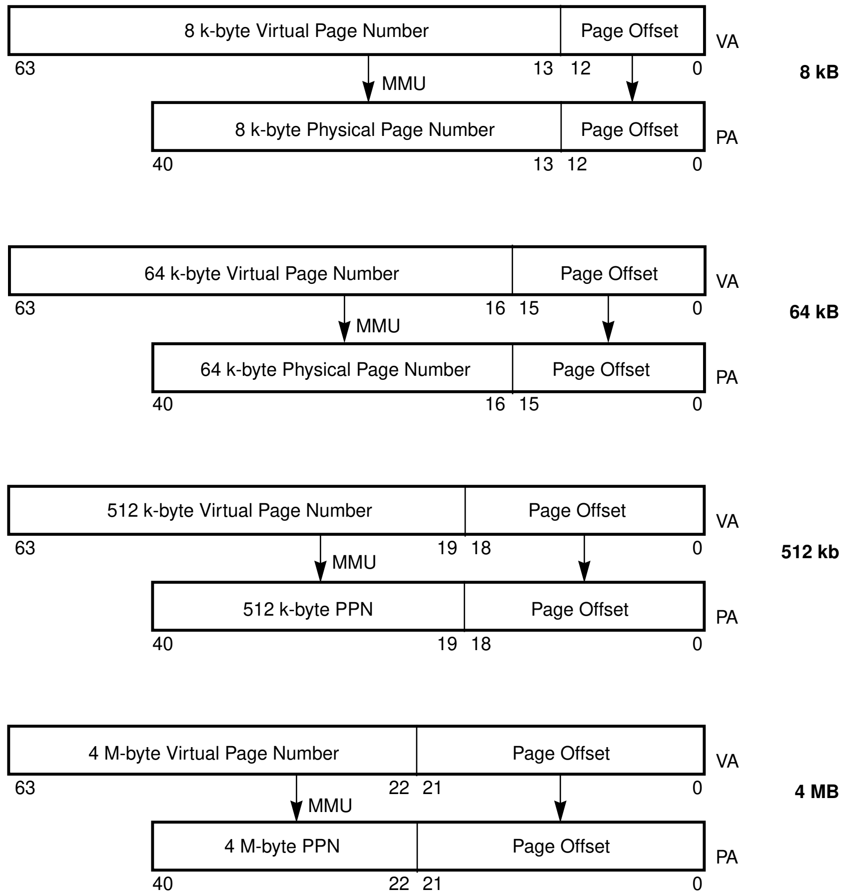


Figure 4-1 Virtual-to-physical Address Translation for all Page Sizes

UltraSPARC IIi implements a 44-bit virtual address space in two equal halves at the extreme lower and upper portions of the full 64-bit virtual address space. Virtual addresses between $0000\ 0800\ 0000\ 0000_{16}$ and $\mathrm{FFFF\ F7FF\ FFFF\ FFFF}_{16}$, inclusive, are termed "out of range" for UltraSPARC IIi and are illegal. (In other words, virtual address bits VA<63:43> must be either all zeros or all ones.) *Figure 4-2* on page 25 illustrates the UltraSPARC IIi virtual address space.

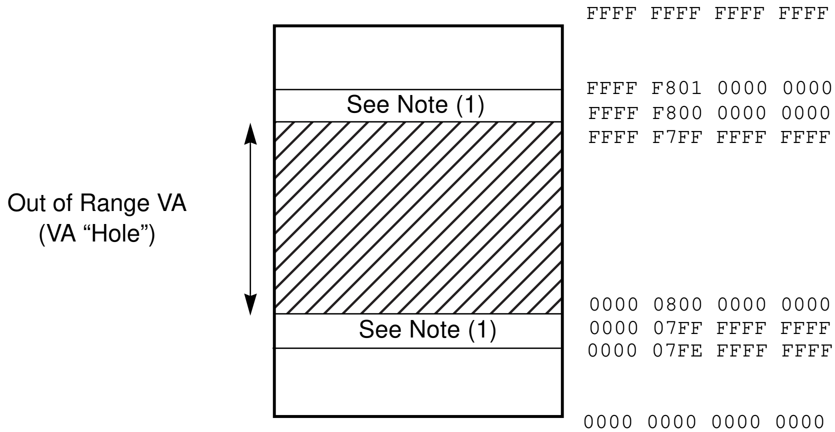


Figure 4-2 UltraSPARC IIi 44-bit Virtual Address Space, with Hole (Same as *Figure 14-2* on page 178)

Note – Throughout this document, when virtual address fields are specified as 64-bit quantities, they are assumed to be sign-extended based on VA<43>.

The operating system maintains translation information in a data structure called the Software Translation Table. The I- and D-MMU each contain a hardware Translation Lookaside Buffer (iTLB and dTLB). These buffers act as independent caches of the Software Translation Table, providing one-cycle translation for the more frequently accessed virtual pages.

Figure 4-3 on page 26 shows a general software view of the UltraSPARC IIi MMU. The TLBs, which are part of the MMU hardware, are small and fast. The Software Translation Table, which is kept in memory, is likely to be large and complex. The Translation Storage Buffer (TSB), which acts like a direct-mapped cache, is the interface between the two. The TSB can be shared by all processes running on a processor, or it can be process specific. The hardware does not require any particular scheme.

The term "TLB hit" means that the desired translation is present in the MMUs on-chip TLB. The term "TLB miss" means that the desired translation is not present in the MMUs on-chip TLB. On a TLB miss the MMU immediately traps to software for TLB miss processing. The TLB miss handler has the option of filling the TLB by any means available, but it is likely to take advantage of the TLB miss support features provided by the MMU, since the TLB miss handler is time-critical code. Hardware support is described in Section 15.3.1, *Hardware Support for TSB Access* on page 201.

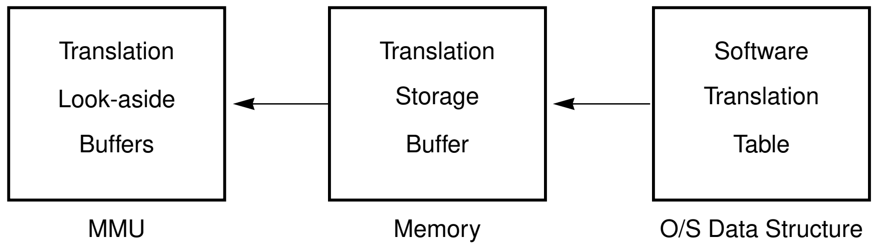


Figure 4-3 Software View of the UltraSPARC IIi MMU

Aliasing between pages of different size (when multiple VAs map to the same PA) may take place, as with the SPARC-V8 Reference MMU. The reverse case, when multiple mappings from one VA/context to multiple PAs produce a multiple TLB match, is not detected in hardware; it produces undefined results.

Note – The hardware ensures the physical reliability of the TLB on multiple matches.

CHAPTER 5

UltraSPARC IIi in a System

5.1 A Hardware Reference Platform

The elements of the hardware, the associated peripherals and their function can be presented by considering each one in the context of a hardware reference platform. *Figure 5-1* shows a typical rendering of such a platform.

This model assumes CPU and SRAM for the E-cache are provided on the same module, to keep the high-speed E-cache interface in a controlled electrical environment and away from the motherboard.

A typical module uses four, 64K x 18 register-latch SRAMs for data, to provide a 512-kilobyte E-cache or two 256K x 36 SRAMs for a 2-megabyte E-cache. A 64K x 18 or 256K x 18 tag SRAM is also used.

The reference platform provides support for two standard, 33 MHz, 32-bit, PCI buses, along with a 66 MHz, 32-bit PCI interface to a bus bridge ASIC, for example, the Advanced PCI Bridge (APB™).

Graphics can be implemented using a PCI add-in card, or by means of a custom UPA64S solution.

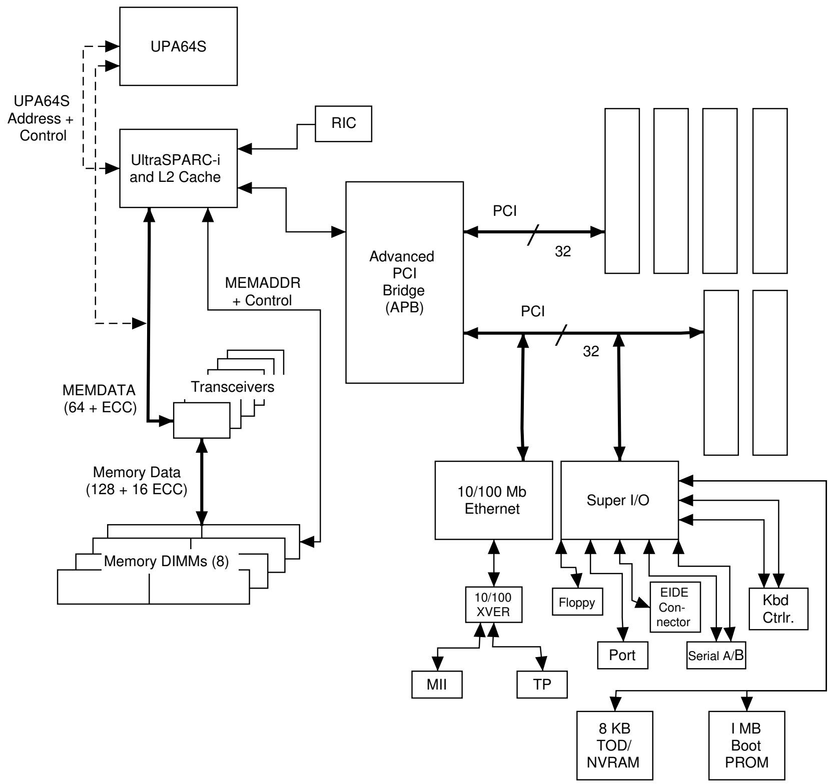


Figure 5-1 Overview of UltraSPARC IIi Reference Platform

5.2 Memory Subsystem

Figure 5-2 shows how memory is connected to, and controlled by, the UltraSPARC IIi. The memory DIMMs are arranged on a 144-bit bus to allow an entire cache line to be fetched in four CAS accesses.

UltraSPARC IIi implements ECC, with single-bit correction and multi-bit detection of errors, for all memory data transfers.

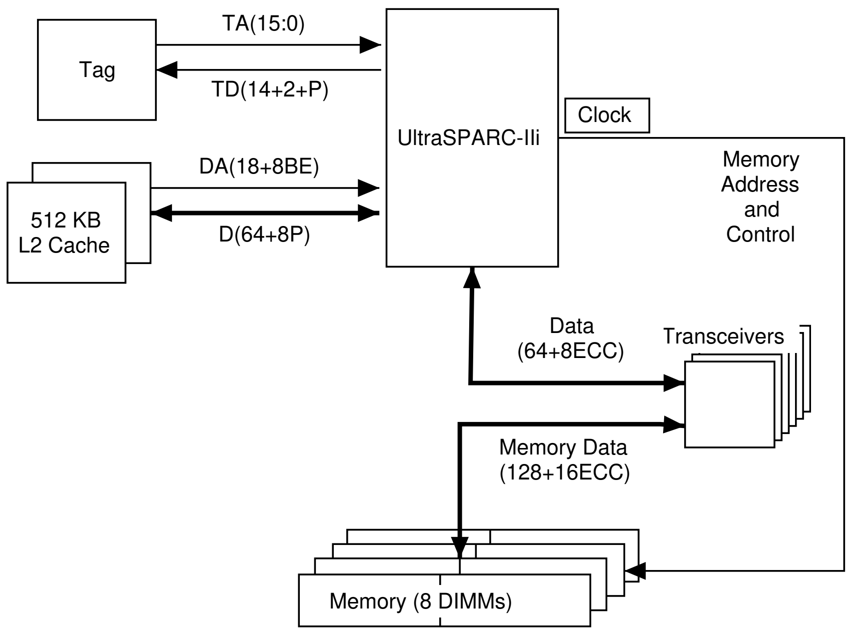


Figure 5-2 A Typical Subsystem: UltraSPARC IIi and Memory—Simplified Block Diagram

5.2.1 E-cache

Synchronous access to the E-cache (L2-cache) is made through a data bus that carries 8-bytes plus parity.

The UltraSPARC-I or UltraSPARC-II 1-1-1 style SRAMs can be used at half the processor clock rate. The UltraSPARC-II 2-2 mode SRAMS are also supported.

There are enough cache address bits to support a 2 MB E-cache, with a practical minimum of 256 kB.

E-cache can be fitted in these alternative configurations:

- 2 - 32k x 36 (data) plus 1-4k x 18 (minimally) (tag: can use 32k x 36) = 256 kilobyte
- 4 - 64k x 18 (data) plus 1-8k x 18 (minimally) (tag: can use 32k x 36) = 512 kilobyte
- 4 - 128k x 18 (data) plus 1-16k x 18 (minimally) (tag: can use 32k x 36) = 1 MB

- 2 - 128k x 36 (data) plus 1-16k x 18 (minimally) (tag: can use 32k x 36) = 1 MB
- 4 - 256k x 18 (data) plus 1-32k x 18 (minimally) (tag: can use 32k x 36) = 2 MB

As provided in UltraSPARC-II, UltraSPARC IIi supports software programming to selectively zero E-cache tag address bits, so that the same module can accommodate different sizes of SRAM IC, without the necessity of tying unused address lines low—which must be done if an over-capacity SRAM is used.

5.2.2 DRAM Memory

The following are the major features of the DRAM modules utilized in UltraSPARC IIi memory:

- Four DIMM pairs for up to 256 Megabytes, using 168-pin JEDEC DIMMs, with 16-Megabit DRAM. Up to one Gigabyte, using 64-Megabit DRAM
- 144-bit DRAM data bus with 8-bit ECC on each 64 bits of data—industry standard ECC pinout
- High performance CMOS silicon gate process
- Single +3.3V ± 0.3 V power supply
- All device pins are 3.3 V compatible
- Low power, 9 mW standby; 1,800 mW active, typical
- Refresh modes: $\overline{\text{CAS}}$-BEFORE-$\overline{\text{RAS}}$ (CBR)
- All inputs are buffered except $\overline{\text{RAS}}$
- 2,048-cycle refresh distributed across 32 ms interval
- Extended Data Out (EDO) access cycles

The UltraSPARC IIi memory design is built with JEDEC standard 168-pin DIMMs. The memory bus is 144 bits wide. RAS and CAS signals are provided that support a maximum of eight 8 - 128 megabyte DIMMs. A mode that supports 11-bit column addresses for 16M X 4, 64 megabit DRAMs allows a maximum of four 8–256 megabyte DIMMs. The memory bus width requires that the DIMMs be populated in pairs at a time. Consequently the minimum memory configuration contains 16 megabytes and the maximum memory configuration contains 1 gigabyte.

These DIMMs are available from many vendors. A composite specification was made considering typical vendor specifications. When the UltraSPARC IIi is programmed according to Chapter 18, *MCU Control and Status Registers*, for a particular frequency and DIMM loading combination, it generates signals that meet this composite specification, if the electrical and topological motherboard layout requirements are met.

5.2.3 Transceivers

The Texas Instruments SN74ALVC16268 is a bidirectional registered 12-bit-to-24-bit bus exchanger, with 3-state outputs.

The transceiver transfers data bidirectionally between the 72-bit UltraSPARC IIi memory data bus, and the 144-bit DIMM memory data bus. The DIMMs cycle data in EDO mode at 37.5 MHz maximum frequency—a period of 26.5 ns.

The transceiver has bus-hold on data inputs, eliminating the need for external pullup resistors. It is available in 56-pin Plastic Shrink Small-Outline (DL) and Thin Shrink Small-Outline (DGG) packages.

The ports connected to the DIMMs include the equivalent of 26Ω series resistors, to make external series termination resistors unnecessary.

The device provides synchronous data exchange between the two ports. Data is stored in the internal registers on the low-to-high transition of the CLK input, provided that the appropriate CLKEN inputs are low. All control inputs, including the CLK inputs, are driven by UltraSPARC IIi

5.3 PCI Interface—Advanced PCI Bridge

The PCI interface of UltraSPARC IIi can be used directly or expanded using one or more PCI bridges. *Figure 5-3* shows an example of the connection of an external PCI subsystem using Sun Microsystems, Inc. Advanced PCI Bridge (APB™).

This configuration uses PCI clocks asynchronous with the processor clock and three or more PCI buses, all compatible with the existing PCI 2.1 standard:

- One 66 MHz, 32-bit primary bus from UltraSPARC IIi to APB; note that multiple APBs can be used for multiplying PCI connectivity
- Two 33 MHz, 32-bit secondary busses from each APB

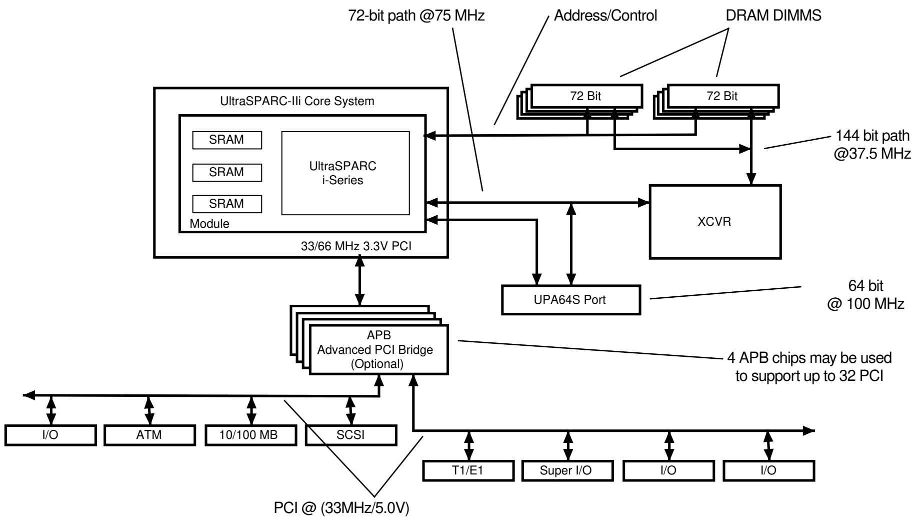


Figure 5-3 UltraSPARC-II*i* System Implementation Example

The interface from UltraSPARC IIi with its I/O subsystems is a 32-bit PCI bus, which can run at either 33 or 66 MHz. UltraSPARC IIi internal PLLs allow slower PCI bus clock rates, down to 20 MHz or 40 MHz for each range respectively. This allows use of more PCI targets than the 2.1 specification permits for full-speed operation. However, the PCI arbiters on UltraSPARC-II*i* and APB only support four master requests per bus. The Advanced PCI Bridge (APB) allows external arbiters on the secondary buses.

The UltraSPARC-II*i* PCI interface runs at 3.3 V only. To support 5 V PCI cards, the Advanced PCI Bridge (APB) must be used, which also provides expansion from one 66 MHz 32-bit PCI bus, to two 32-bit 33 MHz PCI buses. APB provides up to 64-byte write posting and data prefetching, so that the delivered throughput can be higher than a single 33 MHz bus could provide.

The secondary PCI buses have:

- 3.3 Volt operation and signalling, but are compatible with the PCI 5 V signalling environment definition.
- 32-bit data bus

- Compatibility with the PCI Rev. 2.1 Specification
- Support for up to four master devices

Interrupts are not routed through the APB. A separate Drain/Empty protocol is used to guarantee that all DMA writes temporally complete to memory, prior to receipt of an interrupt, and thus before a potential processor trap as a result of that interrupt.

The Primary bus, which can be used with or without the Advanced PCI Bridge, has the same characteristics discussed above, except it can run in the 20-33 MHz or the 40-66 MHz range. UltraSPARC-II*i* operates internally at twice the external PCI clock frequency, that is, up to 132 MHz. This helps reduce the latency involved in crossing clock domains and manipulating state machines.

5.4 RIC Chip

The RIC Chip (SME2210) supports the system resets, system interrupts, system scan, and system clock control functions. Its features include:

- Support for resets from power supply, reset buttons, and scan
- Concentration of all of the interrupts; it sends interrupt numbers to the UltraSPARC IIi
- Direction of SCAN inputs and outputs through scan chains

5.5 UPA64S interface (FFB)

UPA64S is a slave-only interface protocol used, for instance, by proprietary graphics boards. It can be used for any high bandwidth control or data transfers between the processor and a dedicated subsystem. Transfers to and from the UPA64S interface are fully synchronous, since UPA64S receives a PECL clock that is aligned with the processor's clock. The processor transfers data on clock edges that correspond to the UPA64S clock edges. The SME1430 UPA interface runs at 1/4 of the processor clock rate (1/3 for the SME1040 CPU), that is, up to 120 MHz.

UltraSPARC IIi drives the SYSADR (system address), ADR_VLD (address valid) signals, the S_REPLY handshake, and reset (RST_L) to the UPA64S. The data bus (64 bits out of 72) is shared with the transceiver connection to the UltraSPARC IIi. The internal memory controller of the UltraSPARC IIi transfers data aligned to processor clocks, but guarantees that UPA64S transfers appear aligned to the UPA64S clock. In other words, these are valid for three processor clock cycles, and only sampled on the UPA clock edge when UPA64S is driving.

Note that, although the transceivers only cycle the 72-bit MEMDATA at 70 to 90 MHz, the FFB/UPA64S cycle this bus at up to 120 MHz.

5.6 Alternate RMTV support

UltraSPARC IIi has a pin to select a second RMTV to allow use of PC compatible SuperIO chips on a PCI bus—see Section 17.3.2, *RED_state Trap Vector* on page 260.

5.7 Power Management

See Section 13.6.2, *SHUTDOWN* on page 172.

CHAPTER 6

Address Spaces, ASIs, ASRs, and Traps

6.1 Overview

A SPARC-V9 processor provides an Address Space Identifier (ASI) with every address sent to memory. The ASI is used to distinguish between different address spaces, provide an attribute that is unique to an address space, and to map internal control and diagnostics registers within a processor.

SPARC-V9 also extends the limit of virtual addresses from 32 to 64 bits for each address space. SPARC-V9 continues to support 32-bit addressing by masking the upper 32-bits of the 64-bit address to zero when the address mask (AM) bit in the PSTATE register is set.

Both big- and little-endian byte orderings are supported in the UltraSPARC IIi CPU. The default data access byte ordering after a Power-On Reset (POR) is big-endian. Instruction fetches are always big-endian.

6.2 Physical Address Space

The UltraSPARC IIi memory management hardware uses a 44-bit virtual address and an 8-bit ASI to generate a 41-bit physical address. This physical address space can be accessed using either virtual-to-physical address mapping or the MMU bypass mode. For details of this mode See Section 15.10, *MMU Bypass Mode*.

6.2.1 Port Allocations

UltraSPARC IIi divides its physical address space among:

- DRAM
- UPA64S (for a graphics device – FFB)
- PCI, that is further subdivided into PCI A and B bus spaces, when the Advanced PCI Bridge (APB) is used.

Table 6-1 UltraSPARC IIi Address Map

Address Range in PA<40:0>	Size	Port Addressed	Access Type
0x000.0000.0000 - 0x000.3FFF.FFFF	1 GB	Main Memory	Cacheable
0x000.4000.0000 - 0x1FF.FFFF.FFFF	Do not use	Undefined	Cacheable
0x000.0000.0000 - 0x1FB.FFFF.FFFF	Do not use	Undefined	Noncacheable
0x1FC.0000.0000 - 0x1FD.FFFF.FFFF	8 GB	UPA64S (FFB)	Noncacheable
0x1FE.0000.0000 - 0x1FF.FFFF.FFFF	8 GB	PCI	Noncacheable

Only the Cacheability attribute and PA[33:32] are used for steering transactions.

Note that, for compatibility with prior UltraSPARC systems, software should use PA[40:34] equal to all ‘1’s for noncacheable space, and all ‘0’s for cacheable space. UltraSPARC IIi does not detect any errors associated with using a PA[40:34] that violates this convention. UltraSPARC IIi also does not detect the error of using PA[33:32] in violation of the above cacheable/noncacheable partitioning.

Consequently, all possible PA’s decode to some destination. DRAM accesses wrap at the 1 GB boundary, although 4 GB of cacheable space is supported by the L2 cache tags, so the L2 cache will wrap at 4 GB. Noncacheable destinations are determined only by PA[33:32].

6.2.2 Memory DIMM requirements

There can be eight DIMMs ranging in size from eight MB to 128 MB. An alternate mode for supporting DRAM with 11-bit column addressing allows four DIMMs ranging in size from 8 MB to 256 MB. Each DIMM can have two banks of DRAM, controlled by separate RAS# signals.

The Memory Controller timing is programmable, The assumption is that ADDR, CAS#, and WE# are buffered on the DIMM, and that RAS#, CAS# and WE# are buffered on the motherboard.

Note the prior address/cacheability map implies that it is impossible to cause noncacheable access to main memory.

Parameters that affect the address assignments of each DIMM module are DIMM size and the pair in which the DIMM is installed. DIMMs must be loaded in pairs. If the same size memory DIMMs are not installed within a pair, software should either disable the pair, or configure it to match the smaller sized DIMM. Any mixture of sizes is permitted among pairs.

Software can identify the type and size of a DIMM in the system from its address range which is unique for each DIMM type and size. See *Table 7-2* on page 61 or *Table 7-4* on page 64 for the DIMM to PA mapping.

6.2.3 PCI Address Assignments

Table 6-2 Physical address space to PCI space

PCI Address Space	PA[40:0]	CPU Commands Supported	PCI Commands Generated
PCI Configuration Space	0x1FE.0100.0000- 0x1FE.01FF.FFFF	NC read (any) NC write (any)	Configuration Read Configuration Write (may also be Special Cycle)
PCI Bus I/O Space	0x1FE.0200.0000- 0x1FE.02FF.FFFF	NC read (any) NC write (any)	I/O Read I/O Write
Don't Use	0x1FE.0300.0000- 0x1FE.FFFF.FFFF		May wrap to Configuration or I/O Space behavior
PCI Bus Memory Space	0x1FF.0000.0000- 0x1FF.FFFF.FFFF	NC read (4 byte) NC read (8 byte) NC Block read NC write NC Block write NC Instruction fetch	Memory Read Memory Read Multiple Memory Read Line Memory Write Memory Write Memory Read

Table 6-3 Additional Internal UltraSPARC IIi CSR space (noncacheable)

PA[40:0]	Owner
0x1FE.0000.0000 - 0x1FE.0000.01FF	PBM
0x1FE.0000.0200 - 0x1FE.0000.03FF	IOM
0x1FE.0000.0400 - 0x1FE.0000.1FFF	PIE

Table 6-3 Additional Internal UltraSPARC IIi CSR space (noncacheable) *(Continued)*

PA[40:0]	Owner
0x1FE.0000.2000 - 0x1FE.0000.5FFF	PBM
0x1FE.0000.6000 - 0x1FE.0000.9FFF	PIE
0x1FE.0000.A000 - 0x1FE.0000.A7FF	IOM
0x1FE.0000.A800 - 0x1FE.0000.EFFF	PIE
0x1FE.0000.F000 - 0x1FE.00FF.F018	MCU
0x1FE.00FF.F020	PIE
0x1FE.0000.F028 - 0x1FE.00FF.FFFF	MCU

6.2.4 Probing the address space

Generally, systems are configurable, and the boot prom needs to determine what exact configuration is present. There are three address spaces to interrogate: DRAM, UPA64S and PCI.

DRAM probing is explained in detail by Section A.10.2, *Memory Probing* on page 383.

Probing for PCI devices is done using PCI Configuration space accesses. To handle non-response to some of these accesses, software should synchronize on traps as described by Section 16.2.1, *Probing PCI during boot using deferred errors* on page 233. Also see Section 16.5, *Summary of Error Reporting* on page 240

Unlike as for PCI, there is no trapping for non-reply to UPA64S transactions.

If the motherboard ties the P_REPLY[1:0] (UPA64S acknowledgment signals) high during power-on reset, the MCU will assume it received a handshake for all loads and stores targeting the UPA64S address space. This allows software to look for a specific known data pattern being returned by a UPA64S device to report existence. The MCU behavior prevents the software from hanging if no UPA64S device is present.

APB existence can be determined by probing APB-specific registers. See the APB specification for details.

UltraSPARC IIi does not support any UPA-compliant probing algorithm, other than as described.

6.3 Alternate Address Spaces

The SPARC-V9 Address Space Identifier (ASI) is divided into restricted and nonrestricted halves. ASIs in the range $00_{16}..7F_{16}$ are restricted; ASIs in the range $80_{16}..FF_{16}$ are non-restricted. An attempt by non-privileged software to access a restricted ASI causes a *data_access_exception* trap.

ASIs in the ranges $04_{16}..11_{16}$, $18_{16}..19_{16}$, $24_{16}..2C_{16}$, $70_{16}..73_{16}$, $78_{16}..79_{16}$ and $80_{16}..FF_{16}$ are called "normal" or "translating" ASIs. These ASIs are translated by the MMU.

Bypass ASIs are in the range $14_{16}..15_{16}$ and $1C_{16}..1D_{16}$. These ASIs are not translated by the MMU; instead, they pass through their virtual addresses as physical addresses.

UltraSPARC IIi Internal ASIs (also called "Nontranslating ASIs") are in the ranges $45_{16}..6F_{16}$, $76_{16}..77_{16}$ and $7E_{16}..7F_{16}$. These ASIs are not translated by the MMU; instead, they pass through their virtual addresses as physical addresses. Accesses made using these ASIs are always made in "big-endian" mode, regardless of the setting of the D-MMU's IE bit. Accesses to Internal ASIs with invalid virtual address have undefined behavior; they may or may not cause a *data_access_exception* trap. They may or may not alias onto a valid virtual address. Software should not rely on any specific behavior.

Note – MEMBAR `#Sync` is generally needed after stores to internal ASIs. A FLUSH, DONE, or RETRY is needed after stores to internal ASIs that affect instruction accesses. See Section 8.3.8, *Instruction Prefetch to Side-Effect Locations* on page 77.

6.3.1 Supported SPARC-V9 ASIs

The SPARC-V9 architecture defines several address spaces that must be supported by a conforming processor. They are listed in *Table 6-4*. All operand sizes are supported in these accesses. See Appendix G, *ASI Names*" for an alphabetical listing of ASI names and macro syntax.

Table 6-4 Mandatory SPARC-V9 ASIs

ASI Value	ASI Name (Suggested Macro Syntax)	Access	Description	Section
04_{16}	ASI_NUCLEUS (ASI_N)	RW	Implicit address space; nucleus privilege; TL>0	V9
$0C_{16}$	ASI_NUCLEUS_LITTLE (ASI_NL)	RW	Implicit address space; nucleus privilege; TL>0; little endian	V9
10_{16}	ASI_AS_IF_USER_PRIMARY (ASI_AIUP)	RW[2]	Primary address space; user privilege	V9
11_{16}	ASI_AS_IF_USER_SECONDARY (ASI_AIUS)	RW[2]	Secondary address space; user privilege	V9
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE (ASI_AIUPL)	RW[2]	Primary address space; user privilege; little endian	V9
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE (ASI_AIUSL)	RW[2]	Secondary address space; user privilege; little endian	V9
80_{16}	ASI_PRIMARY (ASI_P)	RW	Implicit primary address space	V9
81_{16}	ASI_SECONDARY (ASI_S)	RW	Implicit secondary address space	V9
82_{16}	ASI_PRIMARY_NO_FAULT (ASI_PNF)	R[1]	Primary address space; no fault	V9, Section 1 4.4.6
83_{16}	ASI_SECONDARY_NO_FAULT (ASI_SNF)	R[1]	Secondary address space; no fault	V9, Section 1 4.4.6
88_{16}	ASI_PRIMARY_LITTLE (ASI_PL)	RW	Implicit primary address space; little endian	V9
89_{16}	ASI_SECONDARY_LITTLE (ASI_SL)	RW	Implicit secondary address space; little endian	V9
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE (ASI_PNFL)	R[1]	Primary address space; no fault; little endian	V9, Section 1 4.4.6
$8B_{16}$	ASI_SECONDARY_NO_FAULT_LITTLE (ASI_SNFL)	R[1]	Secondary address space; no fault; little endian	V9, Section 1 4.4.6

[1] Read-only access; causes a *data_access_exception* trap if written respectively.

[2] Causes a *data_access_exception* trap if the page being accessed is privileged.

6.3.2 UltraSPARC IIi (Non-SPARC-V9) ASI Extensions

Table 6-5 on page 41 defines all non-SPARC-V9 ASI extensions supported in UltraSPARC IIi. These ASIs may be used with LDXA, STXA, LDDFA, STDFA instructions only, unless otherwise noted. Other length accesses will cause a *data_access_exception* trap. See Appendix G, *ASI Names*" for an alphabetical listing of ASI names and macro syntax.

Table 6-5 UltraSPARC IIi Extended (non-SPARC-V9) ASIs

ASI Value	ASI Name (Suggested Macro Syntax)	VA	Access	Description	Section
14_{16}	ASI_PHYS_USE_EC (ASI_PHYS_USE_EC)	—	RW [2,5]	Physical address; external cacheable only	Section 15. 10
15_{16}	ASI_PHYS_BYPASS_EC_WITH_EBIT (ASI_PHYS_BYPASS_EC_WITH_EBIT)	—	RW[2]	Physical address; non-cacheable; with side effect	Section 15. 10
$1C_{16}$	ASI_PHYS_USE_EC_LITTLE (ASI_PHYS_USE_EC_L)	—	RW [2,5]	Physical address; external cacheable only; little endian	Section 15. 10
$1D_{16}$	ASI_PHYS_BYPASS_EC_WITH_EBIT_LIT TLE (ASI_PHYS_BYPASS_EC_WITH_EBIT_L)	—	RW [2]	Physical address; non-cacheable; with side-effect; little endian	Section 15. 10
24_{16}	ASI_NUCLEUS_QUAD_LDD (ASI_NUCLEUS_QUAD_LDD)	—	R [1,3]	Cacheable; 128-bit atomic LDDA	Section 13. 6.1
$2C_{16}$	ASI_NUCLEUS_QUAD_LDD_LITTLE (ASI_NUCLEUS_QUAD_LDD_L)	—	R [1,3]	Cacheable; 128-bit atomic LDDA; little endian	Section 13. 6.1
45_{16}	ASI_LSU_CONTROL_REG (ASI_LSU_CONTROL_REG)	0_{16}	RW	Load/store unit control register	Section A. 6
46_{16}	ASI_DCACHE_DATA (ASI_DCACHE_DATA)	—	RW	D-cache data RAM diagnostics access	Section A. 8.1
47_{16}	ASI_DCACHE_TAG (ASI_DCACHE_TAG)	—	RW	D-cache tag/valid RAM diagnostics access	Section A. 8.2
48_{16}	ASI_INTR_DISPATCH_STATUS (ASI_INTR_DISPATCH_STATUS)	0_{16}	R[1]	Interrupt vector dispatch status	Section 11. 10.3
49_{16}	ASI_INTR_RECEIVE (ASI_INTR_RECEIVE)	0_{16}	RW	Interrupt vector receive status	Section 11. 10.5
$4A_{16}$	ASI_UPA_CONFIG_REG (ASI_UPA_CONFIG_REG)	0_{16}	RW	UPA configuration register	Section 18. 5
$4B_{16}$	ASI_ESTATE_ERROR_EN_REG (ASI_ESTATE_ERROR_EN_REG)	0_{16}	RW	E-cache error enable register	Section 16. 6.1
$4C_{16}$	ASI_ASYNC_FAULT_STATUS (ASI_ASYNC_FAULT_STATUS)	0_{16}	RW	ECU Asynchronous fault status register	Section 16. 6.2
$4D_{16}$	ASI_ASYNC_FAULT_ADDRESS (ASI_ASYNC_FAULT_ADDRESS)	0_{16}	RW	ECU Asynchronous fault address register	Section 16. 6.3

Table 6-5 UltraSPARC IIi Extended (non-SPARC-V9) ASIs *(Continued)*

ASI Value	ASI Name (Suggested Macro Syntax)	VA	Access	Description	Section
$4E_{16}$	ASI_ECACHE_TAG_DATA (ASI_EC_TAG_DATA)	0_{16}	RW	E-cache tag/valid RAM data diagnostic access	Section A. 9.2
50_{16}	ASI_IMMU (ASI_IMMU)	0_{16}	R[1]	I-MMU Tag Target Register	Section 15. 9.2
50_{16}	ASI_IMMU (ASI_IMMU)	18_{16}	RW	I-MMU Synchronous Fault Status Register	Section 15. 9.4
50_{16}	ASI_IMMU (ASI_IMMU)	28_{16}	RW	I-MMU TSB Register	Section 15. 9.6
50_{16}	ASI_IMMU (ASI_IMMU)	30_{16}	RW	I-MMU TLB Tag Access Register	Section 15. 9.7
51_{16}	ASI_IMMU_TSB_8KB_PTR_REG (ASI_IMMU_TSB_8KB_PTR_REG)	0_{16}	R[1]	I-MMU TSB 8KB Pointer Register	Section 15. 9.8
52_{16}	ASI_IMMU_TSB_64KB_PTR_REG (ASI_IMMU_TSB_64KB_PTR_REG)	0_{16}	R[1]	I-MMU TSB 64KB Pointer Register	Section 15. 9.8
54_{16}	ASI_ITLB_DATA_IN_REG (ASI_ITLB_DATA_IN_REG)	0_{16}	W[1]	I-MMU TLB Data In Register	Section 15. 9.9
55_{16}	ASI_ITLB_DATA_ACCESS_REG (ASI_ITLB_DATA_ACCESS_REG)	$0_{16}..1F8_{16}$	RW	I-MMU TLB Data Access Register	Section 15. 9.9
56_{16}	ASI_ITLB_TAG_READ_REG (ASI_ITLB_TAG_READ_REG)	$0_{16}..1F8_{16}$	R[1]	I-MMU TLB Tag Read Register	Section 15. 9.9
57_{16}	ASI_IMMU_DEMAP (ASI_IMMU_DEMAP)	0_{16}	W[1]	I-MMU TLB demap	Section 15. 9.10
58_{16}	ASI_DMMU (ASI_D-MMU)	0_{16}	R[1]	D-MMU Tag Target Register	Section 15. 9.2
58_{16}	ASI_DMMU (ASI_DMMU)	8_{16}	RW	I/D MMU Primary Context Register	Section 15. 9.3
58_{16}	ASI_DMMU (ASI_DMMU)	10_{16}	RW	D-MMU Secondary Context Register	Section 15. 9.3
58_{16}	ASI_DMMU (ASI_DMMU)	18_{16}	RW	D-MMU Synch. Fault Status Register	Section 15. 9.4
58_{16}	ASI_DMMU (ASI_DMMU)	20_{16}	R[1]	D-MMU Synch. Fault Address Register	Section 15. 9.5
58_{16}	ASI_DMMU (ASI_DMMU)	28_{16}	RW	D-MMU TSB Register	Section 15. 9.6
58_{16}	ASI_DMMU (ASI_DMMU)	30_{16}	RW	D-MMU TLB Tag Access Register	Section 15. 9.7
58_{16}	ASI_DMMU (ASI_DMMU)	38_{16}	RW	D-MMU VA Data Watchpoint Register	Section A. 5.3
58_{16}	ASI_DMMU (ASI_DMMU)	40_{16}	RW	D-MMU PA Data Watchpoint Register	Section A. 5.4

Table 6-5 UltraSPARC IIi Extended (non-SPARC-V9) ASIs *(Continued)*

ASI Value	ASI Name (Suggested Macro Syntax)	VA	Access	Description	Section
59_{16}	ASI_DMMU_TSB_8KB_PTR_REG (ASI_DMMU_TSB_8KB_PTR_REG)	0_{16}	R[1]	D-MMU TSB 8K Pointer Register	Section 15. 9.8
$5A_{16}$	ASI_DMMU_TSB_64KB_PTR_REG (ASI_DMMU_TSB_64KB_PTR_REG)	0_{16}	R[1]	D-MMU TSB 64K Pointer Register	Section 15. 9.8
$5B_{16}$	ASI_DMMU_TSB_DIRECT_PTR_REG (ASI_DMMU_TSB_DIRECT_PTR_REG)	0_{16}	R[1]	D-MMU TSB Direct Pointer Register	Section 15. 9.8
$5C_{16}$	ASI_DTLB_DATA_IN_REG (ASI_DTLB_DATA_IN_REG)	0_{16}	W[1]	D-MMU TLB Data In Register	Section 15. 9.9
$5D_{16}$	ASI_DTLB_DATA_ACCESS_REG (ASI_DTLB_DATA_ACCESS_REG)	$0_{16}..1F8_{16}$	RW	D-MMU TLB Data Access Register	Section 15. 9.9
$5E_{16}$	ASI_DTLB_TAG_READ_REG (ASI_DTLB_TAG_READ_REG)	$0_{16}..1F8_{16}$	R[1]	D-MMU TLB Tag Read Register	Section 15. 9.9
$5F_{16}$	ASI_DMMU_DEMAP (ASI_DMMU_DEMAP)	0_{16}	W[1]	DMMU TLB demap	Section 15. 9.10
66_{16}	ASI_ICACHE_INSTR (ASI_IC_INSTR)	—	RW[3]	I-cache instruction RAM diagnostic access	Section A. 7.1
67_{16}	ASI_ICACHE_TAG (ASI_IC_TAG)	—	RW[3]	I-cache tag/valid RAM diagnostic access	Section A. 7.2
$6E_{16}$	ASI_ICACHE_PRE_DECODE (ASI_IC_PRE_DECODE)	—	RW[3]	I-cache pre-decode RAM diagnostics access	Section A. 7.3
$6F_{16}$	ASI_ICACHE_NEXT_FIELD (ASI_IC_NEXT_FIELD)	—	RW[3]	I-cache next-field RAM diagnostics access	Section A. 7.4
70_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY (ASI_BLK_AIUP)	—	RW[4,6]	Primary address space; block load/store; user privilege	Section 13. 5.3
71_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY (ASI_BLK_AIUS)	—	RW[4,6]	Secondary address space; block load/store; user privilege	Section 13. 5.3
76_{16}	ASI_ECACHE_W (ASI_EC_W)	<40:39>=1	W[1]	E-cache data RAM diagnostic write access	Section A. 9.1
76_{16}	ASI_ECACHE_W (ASI_EC_W)	<40:39>=2	W[1]	E-cache tag/valid RAM diagnostic write access	Section A. 9.2
77_{16}	ASI_SDBH_ERROR_REG_WRITE (ASI_SDB_ERROR_W)	0_{16}	W[1]	External UDB Error Register; write high	Section 16. 6.4
77_{16}	ASI_SDBL_ERROR_REG_WRITE (ASI_SDB_ERROR_W)	18_{16}	W[1]	External UDB Error Register; write low	Section 16. 6.5
77_{16}	ASI_SDBH_CONTROL_REG_WRITE (ASI_SDB_CONTROL_W)	20_{16}	W[1]	External UDB Control Register; write high	Section 16. 6.6
77_{16}	ASI_SDBL_CONTROL_REG_WRITE (ASI_SDB_CONTROL_W)	38_{16}	W[1]	External UDB Control Register; write low	Section 16. 6.7

Table 6-5 UltraSPARC IIi Extended (non-SPARC-V9) ASIs *(Continued)*

ASI Value	ASI Name (Suggested Macro Syntax)	VA	Access	Description	Section
77_{16}	ASI_SDB_INTR_W (ASI_SDB_INTR_W)	<18:14>=MID, <13:0>= 70_{16}	W[1]	Interrupt vector dispatch	Section 11.10.2
77_{16}	ASI_SDB_INTR_W (ASI_SDB_INTR_W)	40_{16}	W[1]	Outgoing interrupt vector data register 0	Section 11.10.1
77_{16}	ASI_SDB_INTR_W (ASI_SDB_INTR_W)	50_{16}	W[1]	Outgoing interrupt vector data register 1	Section 11.10.1
77_{16}	ASI_SDB_INTR_W (ASI_SDB_INTR_W)	60_{16}	W[1]	Outgoing interrupt vector data register 2	Section 11.10.1
78_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE (ASI_BLK_AIUPL)	—	RW[4]	Primary address space; block load/store; user privilege; little endian	Section 13.5.3
79_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE (ASI_BLK_AIUSL)	—	RW[4]	Secondary address space; block load/store; user privilege; little endian	Section 13.5.3
$7E_{16}$	ASI_ECACHE_R (ASI_EC_R)	<40:39>=1	R[1]	E-cache data RAM diagnostic read access	Section A.8.1
$7E_{16}$	ASI_ECACHE_R (ASI_EC_R)	<40:39>=2	R[1]	E-cache tag/valid RAM diagnostic read access	Section A.8.2
$7F_{16}$	ASI_SDBH_ERROR_REG_READ (ASI_SDBH_ERROR_R)	0_{16}	R[1]	External SDB Error Register; read high	Section 16.6.4
$7F_{16}$	ASI_SDBL_ERROR_REG_READ (ASI_SDBL_ERROR_R)	18_{16}	R[1]	External SDB Error Register; read low	Section 16.6.5
$7F_{16}$	ASI_SDBH_CONTROL_REG_READ (ASI_SDBH_CONTROL_R)	20_{16}	R[1]	External SDB Control Register; read high	Section 16.6.6
$7F_{16}$	ASI_SDBL_CONTROL_REG_READ (ASI_SDBL_CONTROL_R)	38_{16}	R[1]	External SDB Control Register; read low	Section 16.6.7
$7F_{16}$	ASI_SDB_INTR_R	40_{16}	R[1]	Incoming interrupt vector data register 0	Section 11.10.4
$7F_{16}$	ASI_SDB_INTR_R	50_{16}	R[1]	Incoming interrupt vector data register 1	Section 11.10.4
$7F_{16}$	ASI_SDB_INTR_R	60_{16}	R[1]	Incoming interrupt vector data register 2	Section 11.10.4
$7F_{16}$	ASI_INT_ACK	—	R	PCI interrupt acknowledge register	Section 9.2.6
$C0_{16}$	ASI_PST8_PRIMARY (ASI_PST8_P)	—	W[1,4]	Primary address space, 8 8-bit partial store	Section 13.5.1
$C1_{16}$	ASI_PST8_SECONDARY (ASI_PST8_S)	—	W[1,4]	Secondary address space. 8 8-bit partial store	Section 13.5.1
$C2_{16}$	ASI_PST16_PRIMARY (ASI_PSY16_P)	—	W[1,4]	Primary address space, 4 16-bit partial store	Section 13.5.1

Table 6-5 UltraSPARC IIi Extended (non-SPARC-V9) ASIs *(Continued)*

ASI Value	ASI Name (Suggested Macro Syntax)	VA	Access	Description	Section
$C3_{16}$	ASI_PST16_SECONDARY (ASI_PST16_S)	—	W[1,4]	Secondary address space,4; 16-bit partial store	Section 13.5.1
$C4_{16}$	ASI_PST32_PRIMARY (ASI_PST32_P)	—	W[1,4]	Primary address space, 2; 32-bit partial store	Section 13.5.1
$C5_{16}$	ASI_PST32_SECONDARY (ASI_PST32_S)	—	W[1,4]	Secondary address space, 2; 32-bit partial store	Section 13.5.1
$C8_{16}$	ASI_PST8_PRIMARY_LITTLE (ASI_PST8_PL)	—	W[1,4]	Primary address space, 8; 8-bit partial store, little endian	Section 13.5.1
$C9_{16}$	ASI_PST8_SECONDARY_LITTLE (ASI_PST8_SL)	—	W[1,4]	Secondary address space, 8; 8-bit partial store, little endian	Section 13.5.1
CA_{16}	ASI_PST16_PRIMARY_LITTLE (ASI_PST16_PL)	—	W[1,4]	Primary address space,4; 16-bit partial store, little endian	Section 13.5.1
CB_{16}	ASI_PST16_SECONDARY_LITTLE (ASI_PST16_SL)	—	W[1,4]	Secondary address space,4; 16-bit partial store, little endian	Section 13.5.1
CC_{16}	ASI_PST32_PRIMARY_LITTLE (ASI_PST32_PL)	—	W[1,4]	Primary address space, 2; 32-bit partial store; little endian	Section 13.5.1
CD_{16}	ASI_PST32_SECONDARY_LITTLE (ASI_PST32_SL)	—	W [1,4]	Secondary address space, 2; 32-bit partial store; little endian	Section 13.5.1
$D0_{16}$	ASI_FL8_PRIMARY (ASI_FL8_P)	—	RW [4]	Primary address space, one; 8-bit floating point load/store	Section 13.5.2
$D1_{16}$	ASI_FL8_SECONDARY (ASI_FL8_S)	—	RW [4]	Secondary address space, one; 8-bit floating point load/store	Section 13.5.2
$D2_{16}$	ASI_FL16_PRIMARY (ASI_Fl16_P)	—	RW [4]	Primary address space, one; 16-bit floating point load/store	Section 13.5.2
$D3_{16}$	ASI_FL16_SECONDARY (ASI_FL16_S)	—	RW [4]	Secondary address space, one; 16-bit floating point load/store	Section 13.5.2
$D8_{16}$	ASI_FL8_PRIMARY_LITTLE (ASI_FL8_PL)	—	RW [4]	Primary address space, one; 8-bit floating point load/store, little endian	Section 13.5.2
$D9_{16}$	ASI_FL8_SECONDARY_LITTLE (ASI_FL8_SL)	—	RW [4]	Secondary address space, one; 8-bit floating point load/store, little endian	Section 13.5.2

Table 6-5 UltraSPARC IIi Extended (non-SPARC-V9) ASIs *(Continued)*

ASI Value	ASI Name (Suggested Macro Syntax)	VA	Access	Description	Section
DA_{16}	ASI_FL16_PRIMARY_LITTLE (ASI_FL16_PL)	—	RW [4]	Primary address space, one; 16-bit floating point load/store, little endian	Section 13.5.2
DB_{16}	ASI_FL16_SECONDARY_LITTLE (ASI_FL16_SL)	—	RW [4]	Secondary address space, one; 16-bit floating point load/store; little endian	Section 13.5.2
$E0_{16}$	ASI_BLK_COMMIT_PRIMARY (ASI_BLK_COMMIT_P)	—	W[1,4]	Primary address space; block store commit operation	Section 13.5.3
$E1_{16}$	ASI_BLK_COMMIT_SECONDARY (ASI_BLK_COMMIT_S)	—	W[1,4]	Secondary address space; block store commit operation	Section 13.5.3
$F0_{16}$	ASI_BLOCK_PRIMARY (ASI_BLK_P)	—	RW [4]	Primary address space; block load/store	Section 13.5.3
$F1_{16}$	ASI_BLOCK_SECONDARY (ASI_BLK_S)	—	RW [4]	Secondary address space; block load/store	Section 13.5.3
$F8_{16}$	ASI_BLOCK_PRIMARY_LITTLE (ASI_BLK_PL)	—	RW [4]	Primary address space; block load/store; little endian	Section 13.5.3
$F9_{16}$	ASI_BLOCK_SECONDARY_LITTLE (ASI_BLK_SL)	—	RW [4]	Secondary address space; block load/store; little endian	Section 13.5.3

1. Read-/write-only accesses cause a *data_access_exception* trap if written/read respectively.
2. 8-/16-/32-/64-bit accesses allowed.
3. LDDA, STDFA or STXA only. Other types of access cause a *data_access_exception* trap.
4. LDDFA/STDFA only. Other types of access cause a *data_access_exception* trap.
5. Can be used with LDSTUBA, SWAPA, CAS(X)A.
6. Causes a *data_access_exception* trap if the page being accessed is privileged.

6.4 Summary of CSRs mapped to the Noncacheable address space

Table 6-6 CSRs Mapped to Non-cacheable Address Space

PA	Register	Access Size	Section
0x1FE.0000.0000	Undefined (alias to other csrs); was UPA PortID	8 bytes	
0x1FE.0000.0008	Undefined (alias to other csrs); was UPA Config	8 bytes	
0x1FE.0000.0010	Reserved	8 bytes	
0x1FE.0000.0020	Reserved	8 bytes	
0x1FE.0000.0030	DMA UE AFSR	8 bytes	19.4.3.1
0x1FE.0000.0038	DMA UE/CE AFAR	8 bytes	19.4.3.2
0x1FE.0000.0040	DMA CE AFSR	8 bytes	19.4.3.3
0x1FE.0000.0048	DMA UE/CE AFAR (aliases to 0x1fe.0000.0038)	8 bytes	19.4.3.2
0x1FE.0000.0100	Reserved	8 bytes	
0x1FE.0000.0108	Reserved	8 bytes	
0x1FE.0000.0200	IOMMU Control Register	8 bytes	19.3.2.1
0x1FE.0000.0208	IOMMU TSB Base Address Reg	8 bytes	19.3.2.2
0x1FE.0000.0210	IOMMU Flush Register	8 bytes	19.3.2.3
0x1FE.0000.0C00	PCI Bus A Slot 0 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C08	PCI Bus A Slot 1 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C10	PCI Bus A Slot 2 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C18	PCI Bus A Slot 3 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C20	PCI Bus B Slot 0 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C28	PCI Bus B Slot 1 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C30	PCI Bus B Slot 2 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.0C38	PCI Bus B Slot 3 Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1000	SCSI Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1008	Ethernet Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1010	Parallel Port Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1018	Audio Record Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1020	Audio Playback Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1028	Power Fail Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1030	Kbd/mouse/serial Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1038	Floppy Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1040	Spare HW Int Mapping Reg	8 bytes	19.3.3.1

Table 6-6 CSRs Mapped to Non-cacheable Address Space *(Continued)*

PA	Register	Access Size	Section
0x1FE.0000.1048	Keyboard Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1050	Mouse Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1058	Serial Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1060	Reserved		19.3.3.1
0x1FE.0000.1068	Reserved		19.3.3.1
0x1FE.0000.1070	DMA UE Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1078	DMA CE Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1080	PCI Error Int Mapping Reg	8 bytes	19.3.3.1
0x1FE.0000.1088	Reserved	8 bytes	
0x1FE.0000.1090	Reserved	8 bytes	
0x1FE.0000.1098	On board graphics Int Mapping Reg (also mapped at 0x1FE.0000.6000)	8 bytes	19.3.3.2
0x1FE.0000.10A0	Expansion UPA64S Int Mapping Reg (also mapped at 0x1FE.0000.8000)	8 bytes	19.3.3.2
0x1FE.0000.1400- 0x1FE.0000.1418	PCI Bus A Slot 0 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.1420- 0x1FE.0000.1438	PCI Bus A Slot 1 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.1440- 0x1FE.0000.1458	PCI Bus A Slot 2 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.1460- 0x1FE.0000.1478	PCI Bus A Slot 3 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.1480- 0x1FE.0000.1498	PCI Bus B Slot 0 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.14A0- 0x1FE.0000.14B8	PCI Bus B Slot 1 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.14C0- 0x1FE.0000.14D8	PCI Bus B Slot 2 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.14E0- 0x1FE.0000.14F8	PCI Bus B Slot 3 Clear Int Regs	8 bytes	19.3.3.3
0x1FE.0000.1800	SCSI Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1808	Ethernet Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1810	Parallel Port Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1818	Audio Record Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1820	Audio Playback Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1828	Power Fail Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1830	Kbd/mouse/serial Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1838	Floppy Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1840	Spare HW Clear Int Reg	8 bytes	19.3.3.3

Table 6-6 CSRs Mapped to Non-cacheable Address Space *(Continued)*

PA	Register	Access Size	Section
0x1FE.0000.1848	Keyboard Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1850	Mouse Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1858	Serial Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1860	Reserved	8 bytes	19.3.3.3
0x1FE.0000.1868	Reserved	8 bytes	19.3.3.3
0x1FE.0000.1870	DMA UE Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1878	DMA CE Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1880	PCI Error Clear Int Reg	8 bytes	19.3.3.3
0x1FE.0000.1888	Reserved	8 bytes	
0x1FE.0000.1890	Reserved	8 bytes	
0x1FE.0000.1A00	Reserved	8 bytes	
0x1FE.0000.1C00	Reserved	8 bytes	
0x1FE.0000.1C08	Reserved	8 bytes	
0x1FE.0000.1C10	Reserved	8 bytes	
0x1FE.0000.1C18	Reserved	8 bytes	
0x1FE.0000.1C20	PCI DMA Write Synchronization Register	8 bytes	19.3.0.5
0x1FE.0000.2000	PCI Control/Status Register	8 bytes	19.3.0.1
0x1FE.0000.2010	PCI PIO Write AFSR	8 bytes	19.3.0.2
0x1FE.0000.2018	PCI PIO Write AFAR	8 bytes	19.3.0.2
0x1FE.0000.2020	PCI Diagnostic Register	8 bytes	19.3.0.3
0x1FE.0000.2028	PCI Target Address Space Register	8 bytes	19.3.0.4
0x1FE.0000.2800	Reserved	8 bytes	
0x1FE.0000.2808	Reserved	8 bytes	
0x1FE.0000.2810	Reserved	8 bytes	
0x1FE.0000.4800	Reserved	8 bytes	
0x1FE.0000.4808	Reserved	8 bytes	
0x1FE.0000.4810	Reserved	8 bytes	
0x1FE.0000.5000 - 0x1FE.0000.5038	PIO Buffer Diag Access	8 bytes	19.3.0.6
0x1FE.0000.5100 - 0x1FE.0000.5138	DMA Buffer Diag Access	8 bytes	19.3.0.7
0x1FE.0000.51C0	DMA Buffer Diag Access (72:64)	8 bytes	19.3.0.8
0x1FE.0000.6000	On board graphics Int Mapping Reg (also mapped at 0x1FE.0000.1098)	8bytes	19.3.3.2
0x1FE.0000.8000	Expansion UPA64S Int Mapping Reg (also mapped at 0x1FE.0000.10A0)	8bytes	19.3.3.2
0x1FE.0000.A000	Reserved	8 bytes	

Table 6-6 CSRs Mapped to Non-cacheable Address Space *(Continued)*

PA	Register	Access Size	Section
0x1FE.0000.A008	Reserved	8 bytes	
0x1FE.0000.A400	IOMMU Virtual Address Diag Reg	8 bytes	19.3.2.6
0x1FE.0000.A408	IOMMU Tag Compare Diag	8 bytes	19.3.2.7
0x1FE.0000.A500- 0x1FE.0000.A57F	Reserved	8 bytes	
0x1FE.0000.A580- 0x1FE.0000.A5FF	IOMMU Tag Diag	8 bytes	19.3.2.4
0x1FE.0000.A600- 0x1FE.0000.A67F	IOMMU Data RAM Diag	8 bytes	19.3.2.5
0x1FE.0000.A800	PCI Int State Diag Reg	8 bytes	19.3.3.4
0x1FE.0000.A808	OBIO and Misc Int State Diag Reg	8 bytes	
0x1FE.0000.B000- 0x1FE.0000.B3FF	Reserved	8 bytes	
0x1FE.0000.B400- 0x1FE.0000.B7FF	Reserved	8 bytes	
0x1FE.0000.B800- 0x1FE.0000.B87F	Reserved	8 bytes	
0x1FE.0000.B900- 0x1FE.0000.B97F	Reserved	8 bytes	
0x1FE.0000.C000- 0x1FE.0000.C3FF	Reserved	8 bytes	
0x1FE.0000.C400- 0x1FE.0000.C7FF	Reserved	8 bytes	
0x1FE.0000.C800- 0x1FE.0000.C87F	Reserved	8 bytes	
0x1FE.0000.C900- 0x1FE.0000.C97F	Reserved	8 bytes	
0x1FE.0000.F000	FFB_Config	8 bytes	
0x1FE.0000.F010	MC_Control0	8 bytes	
0x1FE.0000.F018	MC_Control1	8 bytes	
0x1FE.0000.F020	Reset_Control	8 bytes	
0x1FE.0100.0000	PCI Configuration Space: Vendor ID	2 bytes	19.3.1.1
0x1FE.0100.0002	PCI Configuration Space: Device ID	2 bytes	19.3.1.2
0x1FE.0100.0004	PCI Configuration Space: Command	2 bytes	19.3.1.3
0x1FE.0100.0006	PCI Configuration Space: Status	2 bytes	19.3.1.4
0x1FE.0100.0008	PCI Configuration Space: Revision ID	2 bytes	19.3.1.5
0x1FE.0100.0009	PCI Configuration Space: Programming I/F Code	1 byte	19.3.1.6
0x1FE.0100.000A	PCI Configuration Space: Sub-class Code	1 byte	19.3.1.7
0x1FE.0100.000B	PCI Configuration Space: Base Class Code	1 byte	19.3.1.8

Table 6-6 CSRs Mapped to Non-cacheable Address Space *(Continued)*

PA	Register	Access Size	Section
0x1FE.0100.000D	PCI Configuration Space: Latency Timer	1 byte	19.3.1.9
0x1FE.0100.000E	PCI Configuration Space: Header Type	1 byte	19.3.1.10
0x1FE.0100.0040	PCI Configuration Space: Bus Number	1 byte	19.3.1.11
0x1FE.0100.0041	PCI Configuration Space: Subordinate Bus Number	1 byte	19.3.1.11
0x1FE.0100.0042-0x1FE.0100.07FF	Reserved	Any	
0x1FE.0200.0000-0x1FE.02FF.FFFF	PCI Bus I/O Space	Any	
0x1FF.0000.0000-0x1FF.FFFF.FFFF	PCI Bus Memory Space	Any	

Compatibility Note – A read of any addresses labelled “Reserved” above returns zeros, and writes have no effect.

Caution – Reads to noncacheable addresses not listed above may return zeroes or alias an existing CSR in the table. Writes to noncacheable addresses not listed above may result in a no-op or invoke an alias to an existing CSR in the table and modify it unexpectedly. Software should protect addresses over the full range of 0x1FE.0000.0000 through 0x1FE.00FF.FFFF to prevent back-door access.

6.5 Ancillary State Registers

6.5.1 Overview of ASRs

SPARC-V9 provides up to 32 Ancillary State Registers (ASRs 0..31). ASRs 0..6 are defined by the SPARC-V9 ISA; ASRs 7..15 are reserved for future use by the architecture. ASRs 16..31 are available for use by an implementation.

6.5.2 SPARC-V9-Defined ASRs

Table 6-7 defines the SPARC-V9 ASRs that must be supported by a conforming processor implementation. *Table 6-8* suggests the assembly language syntax for accessing these registers.

Table 6-7 Mandatory SPARC-V9 ASRs

ASR Value	ASR Name	Access	Description	Section
00_{16}	Y_REG	RW	Y register	V9
02_{16}	COND_CODE_REG	RW	Condition code register	V9
03_{16}	ASI_REG	RW	ASI register	V9
04_{16}	TICK_REG	R[1,2]	TICK register	V9
05_{16}	PC	R[2]	Program Counter	V9
06_{16}	FP_STATUS_REG	RW	Floating-point status register	V9

[1.] An attempt to read this register by non-privileged software with NPT = 1 causes a *privileged_action* trap. The tick register can only be written with the privileged wrpr instruction.

[2.] Read-only—an attempt to write this register causes an *illegal_instruction* trap.

Table 6-8 Suggested Assembler Syntax for Mandatory ASRs

Operation	Syntax
rd	%y, reg_{rd}
wr	reg_{rs1}, *reg_or_imm*, %y
rd	%ccr, reg_{rd}
wr	reg_{rs1}, *reg_or_imm*, %ccr
rd	%asi, reg_{rd}
wr	reg_{rs1}, *reg_or_imm*, %asi
rd	%tick, reg_{rd}
rd	%pc reg_{rd}
rd	%fprs, reg_{rd}
wr	reg_{rs1}, *reg_or_imm*, %fprs

6.5.3 Non-SPARC-V9 ASRs

Non-SPARC-V9 ASRs are listed in Section Table 6-9.

Table 6-9 Non-SPARC-V9 ASRs

ASR Value	ASR Name/Syntax	Access	Description	Section
10_{16}	PERF_CONTROL_REG	RW[3]	Performance Control Reg (PCR)	Section B .2
11_{16}	PERF_COUNTER	RW[4]	Performance Instrumentation Counters (PIC)	Section B .4
12_{16}	DISPATCH_CONTROL_REG	RW[3]	Dispatch Control Register (DCR)	Section A.3
13_{16}	GRAPHIC_STATUS_REG	RW[2]	Graphics Status Register (GSR)	Section 1 3.3
14_{16}	SET_SOFTINT	W[1]	Set bit(s) in per-processor Soft Interrupt register	Section 1 1.11
15_{16}	CLEAR_SOFTINT	W[1]	Clear bit(s) in per-processor Soft Interrupt register	Section 1 1.11
16_{16}	SOFTINT_REG	RW[3]	Per-processor Soft Interrupt register	Section 1 1.11
17_{16}	TICK_CMPR_REG	RW[3]	TICK compare register	Section 1 4.5.1

[1]Read accesses cause an *illegal_instruction* trap. Nonprivileged write accesses cause a *privileged_opcode* trap.

[2]Accesses cause an *fp_disabled* trap if PSTATE.PEF or FPRS.FEF are zero.

[3]Nonprivileged accesses cause a *privileged_opcode* trap.

[4]Nonprivileged accesses with PCR.PRIV=0 cause a *privileged_action* trap.

Table 6-10 Suggested Assembler Syntax for Non-SPARC V9 ASRs

Operation	Syntax
rd	`%pcr`, reg_{rd}
wr	reg_{rs1},`%pcr`
rd	`%pic`, reg_{rd}
wr	reg_{rs1},`%pic`
rd	`%gsr`, reg_{rd}
wr	reg_{rs1},`%gsr`
wr	reg_{rs1},`%clear_softint`
wr	reg_{rs1},`%set_softint`
rd	`%softint`, reg_{rd}
wr	reg_{rs1},`%softint`
rd	`%tick_cmpr`, reg_{rd}

Table 6-10 Suggested Assembler Syntax for Non-SPARC V9 ASRs

Operation	Syntax
wr	reg_{rs1},`%tick_cmpr`
rd	`%dcr`, reg_{rd}
wr	reg_{rs1},`%dcr`

6.6 Other UltraSPARC IIi Registers

Table 6-11 lists additional sets of 64-bit global registers supported by UltraSPARC IIi

Table 6-11 Other UltraSPARC IIi Registers

Register Name	Access	Description	Section
INTERRUPT_GLOBAL_REG	RW	8 Interrupt handler globals	Section 14 .5.9
MMU_GLOBAL_REG	RW	8 MMU handler globals	Section 14 .5.9

6.7 Supported Traps

Table 6-12 lists the traps supported by UltraSPARC IIi.

Table 6-12 Traps Supported in UltraSPARC IIi

Exception or Interrupt Request	Globals[9]	TT	Priority
Reserved	—	000_{16}	*n/a*
power_on_reset	AG	001_{16}	0
watchdog_reset	AG	002_{16}	1[1]
externally_initiated_reset	AG	003_{16}	1[1]
software_initiated_reset	AG	004_{16}	1[1]
RED_state_exception	AG	005_{16}	1[1]
instruction_access_exception	MG	008_{16}	5
instruction_access_error	AG	$00A_{16}$	3

Table 6-12 Traps Supported in UltraSPARC IIi *(Continued)*

Exception or Interrupt Request	Globals[9]	TT	Priority
illegal_instruction	AG	010_{16}	7[10]
privileged_opcode	AG	011_{16}	6
fp_disabled	AG	020_{16}	8
fp_exception_ieee_754	AG	021_{16}	11[2]
fp_exception_other	AG	022_{16}	11[2]
tag_overflow	AG	023_{16}	14
clean_window	AG	$024_{16}..027_{16}$	10
division_by_zero	AG	028_{16}	15
data_access_exception	MG	030_{16}	12[3]
data_access_error	AG	032_{16}	12[3]
mem_address_not_aligned	AG	034_{16}	10[4, 10]
LDDF_mem_address_not_aligned	AG	035_{16}	10[4]
STDF_mem_address_not_aligned	AG	036_{16}	10[4]
privileged_action	AG	037_{16}	11[2]
interrupt_level_n (n=1..15)	AG	$041_{16}..04F_{16}$	$32-n$
interrupt_vector	IG	060_{16}	16[5]
PA_watchpoint	AG	061_{16}	12[5]
VA_watchpoint	AG	062_{16}	11[2]
corrected_ECC_error	AG	063_{16}	33
fast_instruction_access_MMU_miss	MG	$064_{16}..067_{16}$	2[6]
fast_data_access_MMU_miss	MG	$068_{16}..06B_{16}$	12[3,7]
fast_data_access_protection	MG	$06C_{16}..06F_{16}$	12[3,8]
spill_n_normal (n=0..7)	AG	$080_{16}..09F_{16}$	9
spill_n_other (n=0..7)	AG	$0A0_{16}..0BF_{16}$	9
fill_n_normal (n=0..7)	AG	$0C0_{16}..0DF_{16}$	9
fill_n_other (n=0..7)	AG	$0E0_{16}..0FF_{16}$	9
trap_instruction	AG	$100_{16}..17F_{16}$	16[5]

[1] Priority 1 traps are processed in the following order: XIR>WDR>SIR>RED.

[2] *Fp_exception_ieee_754, fp_exception_other* are mutually exclusive with memory access traps such as *privileged_action* and *VA_watchpoint*. *Privileged_action* has higher priority than *VA_watchpoint*.

[3] Priority 12 traps are processed in the following program order: *data_access_exception* > *fast_data_access_MMU_miss* / *fast_data_access_protection* > *PA_watchpoint* > *data_access_error*.

[4] Priority 10 traps are processed in the following order: *LDDF/STDF_mem_address_not_aligned* > *mem_address_not_aligned* trap. *LDDF/STDF_mem_address_not_aligned* traps are mutually exclusive.

[5] Priority 16 traps are processed in the following order: *trap instruction* > *interrupt_vector*.

[6] When an MMU fault is detected during an instruction access, a *fast_instruction_access_MMU_miss* trap is generated instead of an *instruction_access_MMU_miss* trap.

[7] A *fast_data_access_MMU_miss* trap is generated instead of *a data_access_MMU_miss* trap.

[8] A *fast_data_access_protection* trap is generated instead of a *data_access_protection* trap.

[9] AG = alternate globals, MG = MMU globals, IG = interrupt globals

[10] Some ASIs must be used with specific types of loads and stores; for example, block ASIs can be used only with LDDFA/STDFA. When these ASIs are used with incorrect opcodes, they do not take *mem_address_not_aligned* or *illegal_instruction* traps for memory and register alignment required by the ASI. For example, block ASIs require 64-byte alignment, but an LDFA opcode with a block ASI checks only for 4-byte alignment.

CHAPTER 7

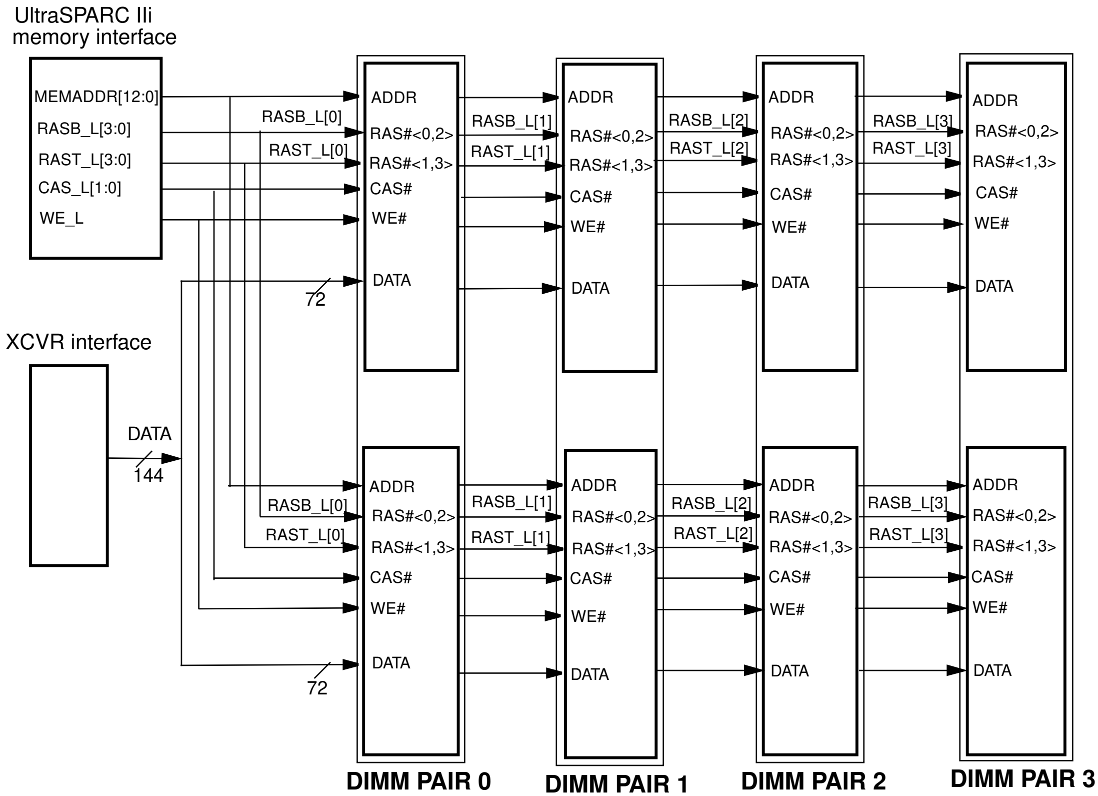
UltraSPARC IIi Memory System

7.1 Overview

The UltraSPARC IIi Memory system is designed to provide overall comparable performance with existing UltraSPARC systems, while using a narrower memory interface. Using EDO DRAMs achieves a CAS cycle half as long as that possible using FPM. Control signals are asserted on processor clock boundaries to allow precise control of DRAM signal transitions.

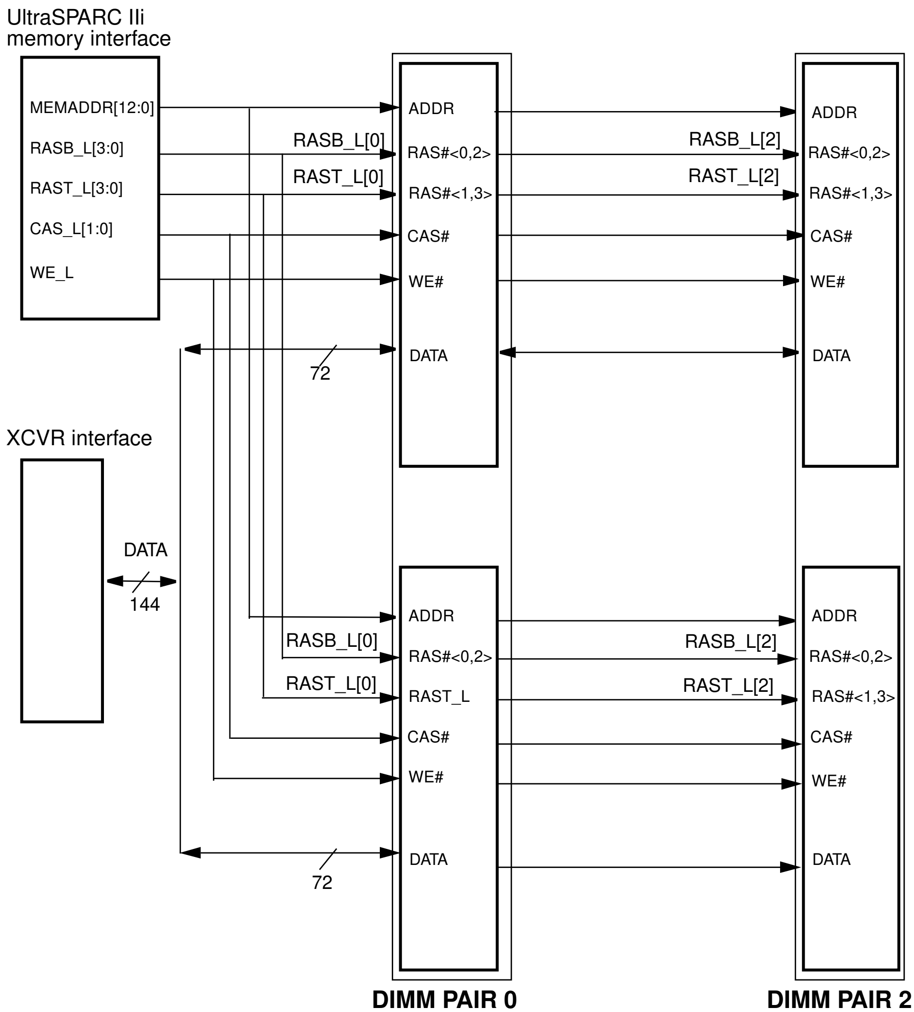
In addition to addressing that supports 10-bit column address DRAMs, an additional mode supports 11-bit column addressing. Since the total available address bits in the memory controller is constant, at 1 GB maximum addressable, the maximum number of DIMM pairs in this mode is halved in 11-bit column address mode.

The connectivity of RASB_L/RAST_L is critical and non-intuitive given the JEDEC standard pin names for the DIMMs. Exactly follow the schematics in *Figure 7-1* and *Figure 7-2*. The B and T versions of RAS must go to the same DIMM since there are not separate B and T versions of the refresh enable/disable bits for each DIMM. See Section 18.2, *Mem_Control0 Register* on page 267.



Two copies of CAS_L are provided only to reduce loading. Both are always asserted together.
Real configuration needs buffers on RAS/CAS/WE.
See design guide for requirements for min/max delays and skew relationships.

Figure 7-1 Memory RAS Wiring with 10-bit Column, 8-128 MB DIMM



Two copies of CAS_L are provided only to reduce loading. Both are always asserted together.
Real configuration needs buffers on RAS/CAS/WE.
See design guide for requirements for min/max. delays and skew relationships.

Figure 7-2 Memory RAS Wiring with 11-bit Column, 8-256MB DIMM

7.2 10-bit Column Addressing

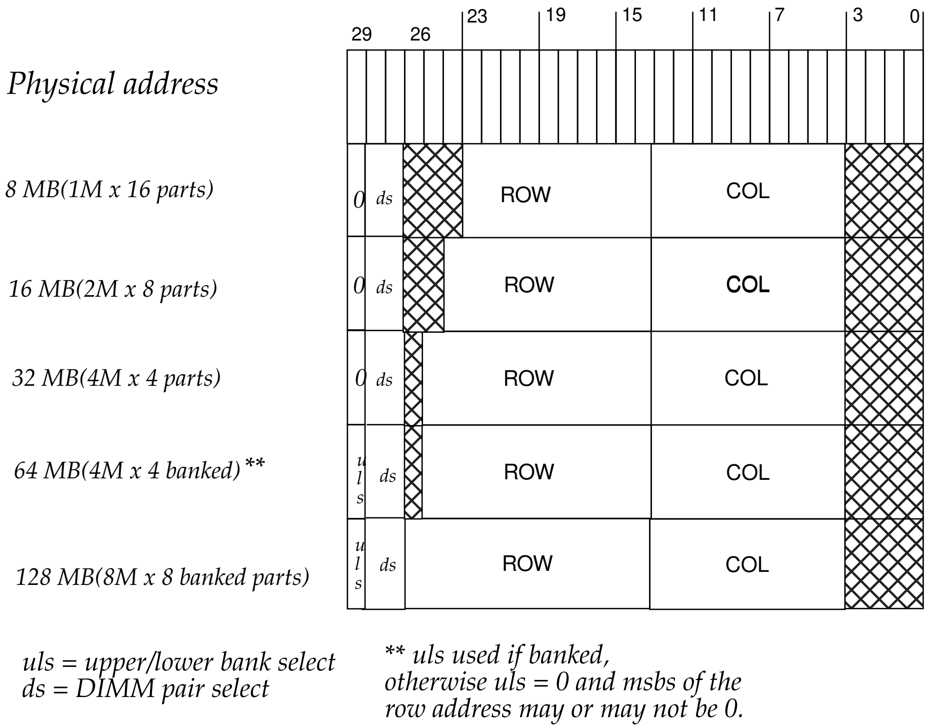


Figure 7-3 UltraSPARC IIi Memory Addressing for 10-bit Column Address Mode

In this scheme, PA[28:27] is used as a DIMM select; it selects a DIMM-pair. PA[29] is used as a upper/lower bank select: 0 = bottom bank, 1 = top bank. DIMMs that contain only a single (bottom) bank must have PA[29] = 0 to be accessed. Mapping of PA[29:27] to RAS assertion is shown in *Table 7-3*.

Table 7-1 PA[29:27] to RASX_L Mapping for 10-bit Column Address Mode

PA[29:27]	RAS_L Asserted
000	RASB_L[0]
001	RASB_L[1]
010	RASB_L[2]

Table 7-1 PA[29:27] to RASX_L Mapping for 10-bit Column Address Mode *(Continued)*

PA[29:27]	RAS_L Asserted
011	RASB_L[3]
100	RAST_L[0]
101	RAST_L[1]
110	RAST_L[2]
111	RAST_L[3]

Table 7-2 Memory Address Map for 10-bit Column Address Mode

DIMM Pair	Individual DIMM size	Address Range (PA[29:0])
0	8MB	0x0000_0000 to 0x00FF_FFFF
0	16MB	0x0000_0000 to 0x01FF_FFFF
0	32MB	0x0000_0000 to 0x03FF_FFFF
0	64MB	0x0000_0000 to 0x07FF_FFFF
0	64MB (banked)	0x0000_0000 to 0x03FF_FFFF and 0x2000_0000 to 0x23FF_FFFF
0	128MB (banked)	0x0000_0000 to 0x07FF_FFFF and 0x2000_0000 to 0x27FF_FFFF
1	8MB	0x0800_0000 to 0x08FF_FFFF
1	16MB	0x0800_0000 to 0x09FF_FFFF
1	32MB	0x0800_0000 to 0x0BFF_FFFF
1	64MB	0x0800_0000 to 0x0FFF_FFFF
1	64MB (banked)	0x0800_0000 to 0x0BFF_FFFF and 0x2800_0000 to 0x2BFF_FFFF
1	128MB (banked)	0x0800_0000 to 0x0FFF_FFFF and 0x2800_0000 to 0x2FFF_FFFF
2	8MB	0x1000_0000 to 0x10FF_FFFF
2	16MB	0x1000_0000 to 0x11FF_FFFF
2	32MB	0x1000_0000 to 0x13FF_FFFF
2	64MB	0x1000_0000 to 0x17FF_FFFF
2	64MB (banked)	0x1000_0000 to 0x13FF_FFFF and 0x3000_0000 to 0x33FF_FFFF
2	128MB (banked)	0x1000_0000 to 0x17FF_FFFF and 0x3000_0000 to 0x37FF_FFFF
3	8MB	0x1800_0000 to 0x18FF_FFFF
3	16MB	0x1800_0000 to 0x19FF_FFFF

Table 7-2 Memory Address Map for 10-bit Column Address Mode *(Continued)*

DIMM Pair	Individual DIMM size	Address Range (PA[29:0])
3	32MB	0x1800_0000 to 0x1BFF_FFFF
3	64MB	0x1800_0000 to 0x1FFF_FFFF
3	64MB (banked)	0x1800_0000 to 0x1BFF_FFFF and 0x3800_0000 to 0x3BFF_FFFF
3	128MB (banked)	0x1800_0000 to 0x1FFF_FFFF and 0x3800_0000 to 0x3FFF_FFFF

7.3 11-bit Column Addressing

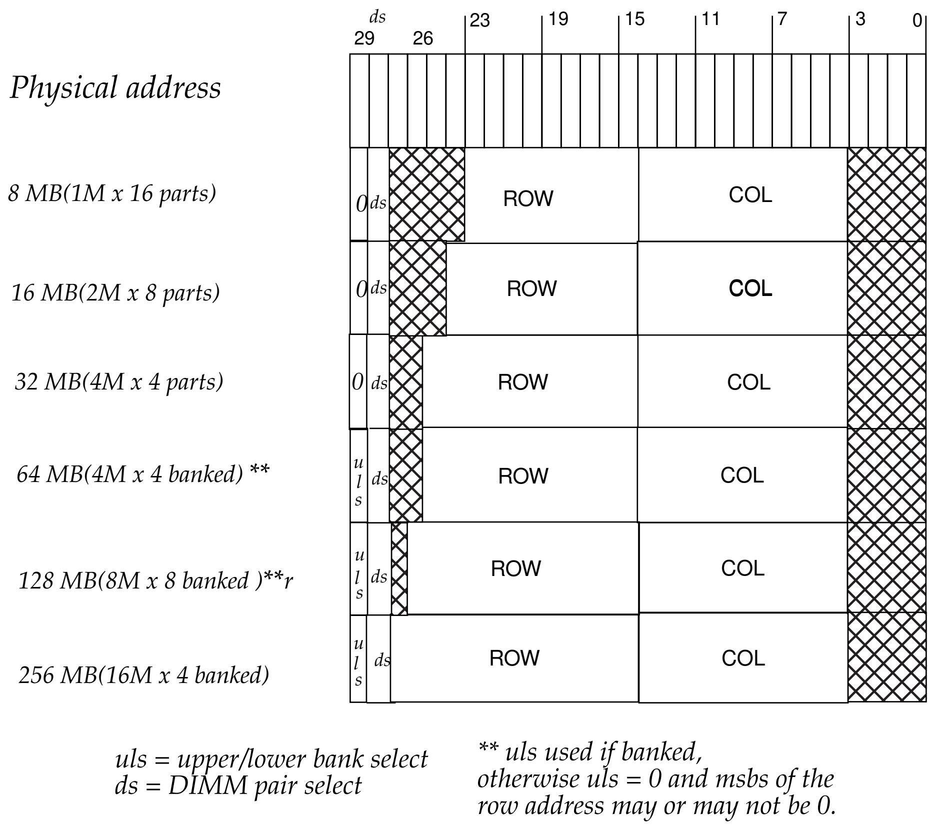


Figure 7-4 UltraSPARC IIi Memory Addressing for 11-bit Column Address Mode

In this scheme, PA[28] is used as a DIMM select; it selects a DIMM-pair. PA[29] is used as a upper/lower bank select: 0 = bottom bank, 1 = top bank. DIMMs that contain only a single (bottom) bank must have PA[29] = 0 in order to be accessed. The mapping of PA[29:28]into RASX_L[?] is shown in *Table 7-3*.

Table 7-3 PA[29:28] to RASX_L Mapping for 11-bit Column Address Mode

PA[29:28]	RAS_L Asserted
00	RASB_L[0]
01	RASB_L[2]
10	RAST_L[0]
11	RAST_L[2]

Table 7-4 Memory Address Map for 11-bit Column Address Mode

DIMM Pair	Individual DIMM size	Address Range (PA[29:0])
0	8MB	0x0000_0000 to 0x00FF_FFFF
0	16MB	0x0000_0000 to 0x01FF_FFFF
0	32MB	0x0000_0000 to 0x03FF_FFFF
0	64MB	0x0000_0000 to 0x07FF_FFFF
0	64MB (banked)	0x0000_0000 to 0x03FF_FFFF and 0x2000_0000 to 0x23FF_FFFF
0	128MB	0x0000_0000 to 0x0FFF_FFFF
0	128MB (banked)	0x0000_0000 to 0x07FF_FFFF and 0x2000_0000 to 0x27FF_FFFF
0	256MB (banked)	0x0000_0000 to 0x0FFF_FFFF and 0x2000_0000 to 0x2FFF_FFFF
2	8MB	0x1000_0000 to 0x10FF_FFFF
2	16MB	0x1000_0000 to 0x11FF_FFFF
2	32MB	0x1000_0000 to 0x13FF_FFFF
2	64MB	0x1000_0000 to 0x17FF_FFFF
2	64MB (banked)	0x1000_0000 to 0x13FF_FFFF and 0x3000_0000 to 0x33FF_FFFF
2	128MB	0x1000_0000 to 0x1FFF_FFFF
2	128MB (banked)	0x1000_0000 to 0x17FF_FFFF and 0x3000_0000 to 0x37FF_FFFF
2	256MB (banked)	0x1000_0000 to 0x1FFF_FFFF and 0x3000_0000 to 0x3FFF_FFFF

CHAPTER 8

Cache and Memory Interactions

8.1 Introduction

This chapter describes various interactions between the caches and memory, and the management processes that an operating system must perform to maintain data integrity in these cases. In particular, it discusses:

- Invalidation of one or more cache entries – when and how to do it
- Differences between cacheable and non-cacheable accesses
- Ordering and synchronization of memory accesses
- Accesses to addresses that cause side effects (I/O accesses)
- Non-faulting loads
- Instruction prefetching
- Load and store buffers

This chapter only addresses coherence in a uniprocessor environment. For more information about coherence in multi-processor environments, see Chapter 20, *SPARC-V9 Memory Models*.

8.2 Cache Flushing

Data in the level-1 (read-only or write-through) caches can be flushed by invalidating the entry in the cache. Modified data in the level-2 (writeback) cache—subsequently referred to as the External or E-cache—must be written back to memory when flushed.

Cache flushing is required in the following cases:

- **I-cache:** Flush is needed before executing code that is modified by a local store instruction other than block commit store, see Section 3.1.1.1, *Instruction Cache (I-cache)*. This is done with the FLUSH instruction or using ASI accesses. See Section A.7, *I-cache Diagnostic Accesses* on page 373. When ASI accesses are used, software must ensure that the flush is done on the same processor as the stores that modified the code space.
- **D-cache:** Flush is needed when a physical page is changed from (virtually) cacheable to (virtually) noncacheable, or when an illegal address alias is created. (see Section 8.2.1, *Address Aliasing Flushing* on page 66.) This is done with a displacement flush. (see Section 8.2.3, *Displacement Flushing* on page 67.) or using ASI accesses. (See Section A.8, *D-cache Diagnostic Accesses* on page 378.)
- **E-cache:** Flush is needed for stable storage. Examples of stable storage include battery-backed memory and transaction logs. This is done with either a displacement flush. (see Section 8.2.3, *Displacement Flushing* on page 67.) or a store with ASI_BLK_COMMIT_{PRIMARY,SECONDARY}. Flushing the E-cache flushes the corresponding blocks from the I- and D-caches, because UltraSPARC IIi maintains inclusion between the external and internal caches. See Section 8.2.2, *Committing Block Store Flushing* on page 67.

8.2.1 Address Aliasing Flushing

A side-effect inherent in a virtual-indexed cache is *illegal address aliasing*. Aliasing occurs when multiple virtual addresses map to the same physical address. Since UltraSPARC IIi's D-cache is indexed with the virtual address bits and is larger than the minimum page size, it is possible for the different aliased virtual addresses to end up in different cache blocks. Such aliases are illegal because updates to one cache block will not be reflected in aliased cache blocks.

Normally, software avoids illegal aliasing by forcing aliases to have the same address bits (*virtual color*) up to an *alias boundary*. For UltraSPARC IIi, the minimum alias boundary is 16 kB; this size may increase in future designs. When the alias boundary is violated, software must flush the D-cache if the page was virtual cacheable. In this case, only one mapping of the physical page can be allowed in the D-MMU at a time. Alternatively, software can turn off virtual caching of illegally aliased pages. This allows multiple mappings of the alias to be in the D-MMU and avoids flushing the D-cache each time a different mapping is referenced.

Note – A change in virtual color when allocating a free page does not require a D-cache flush, because the D-cache is write-through.

8.2.2 Committing Block Store Flushing

In UltraSPARC IIi, stable storage must be implemented by software cache flush. Data that is present and modified in the E-cache must be written back to the stable storage.

Two ASIs: (ASI_BLK_COMMIT_{PRIMARY,SECONDARY}) are implemented by UltraSPARC IIi to perform these writebacks efficiently when software can ensure exclusive write access to the block being flushed. Using these ASIs, software can write back data from the floating-point registers to memory and invalidate the entry in the cache. The data in the floating-point registers must first be loaded by a block load instruction. A MEMBAR `#Sync` instruction is needed to ensure that the flush is complete. See also Section 13.5.3, *Block Load and Store Instructions* on page 164.

8.2.3 Displacement Flushing

Cache flushing also can be accomplished by a displacement flush. This is done by 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 starting a displacement flush, otherwise the TLB miss handler may put new data into the caches.

Note – Diagnostic ASI accesses to the E-cache can be used to invalidate a line, but they are generally not an alternative to displacement flushing. Modified data in the E-cache will not be written back to memory using these ASI accesses. See Section A.9, *E-cache Diagnostics Accesses* on page 380.

8.3 Memory Accesses and Cacheability

Note – Atomic load-store instructions are treated as both a load and a store; they can be performed only in cacheable address spaces.

8.3.1 Coherence Domains

Two types of memory operations are supported in UltraSPARC IIi: 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. In TSO mode, UltraSPARC IIi maintains TSO ordering, regardless of the cacheability of the accesses. For SPARC-V9 compatibility while in PSO or RMO mode, a MEMBAR `#Lookaside` should be used between a store and a subsequent load to the same noncacheable address. See *The SPARC Architecture Manual, Version 9* for more information about the SPARC-V9 memory models.

Note – On UltraSPARC IIi, a MEMBAR `#Lookaside` executes more efficiently than a MEMBAR `#StoreLoad`.

8.3.1.1 Cacheable Accesses

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

- Data resides in real memory locations.
- They observe supported cache coherence protocol.
- The unit of coherence is 64 bytes.

8.3.1.2 Non-Cacheable 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 UltraSPARC IIi 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 with the E-bit set (that is, those having side-effects) are all strongly ordered with respect to other noncacheable accesses with the E-bit set. In addition, store buffer compression is disabled for these accesses. Speculative loads with the E-bit set cause a *data_access_exception* trap (with SFSR.FT=2, speculative load to page marked with E-bit).

Note – The side-effect attribute does not imply noncacheability.

8.3.1.3 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 8-1* illustrates the issues involved in mixing cacheable and noncacheable accesses.

Code Example 8-1 Memory Ordering and MEMBAR Examples

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

Process A:

While (1)

{

      Store D1:data produced

1     MEMBAR #StoreStore (needed in PSO, RMO)

      Store F1:set flag

      While F1 is set (spin on flag)

      Load F1

2  MEMBAR #LoadLoad | #LoadStore (needed in RMO)

   Load D2

}


Process B:

While (1)

{

   While F1 is cleared (spin on flag)

      Load F1

2  MEMBAR #LoadLoad | #LoadStore (needed in RMO)

   Load D1

   Store D2

1  MEMBAR #StoreStore (needed in PSO, RMO)

   Store F1:clear flag

}
```

Note – A MEMBAR `#MemIssue` or MEMBAR `#Sync` is needed if ordering of cacheable accesses following noncacheable accesses must be maintained in PSO or RMO.

Due to load and store buffers implemented in UltraSPARC IIi, *Code Example 8-1* may not work in PSO and RMO modes without the MEMBARs shown in the program segment.

In TSO mode, loads and stores (except block stores) cannot pass earlier loads, and stores cannot pass earlier stores; therefore, no MEMBAR is needed.

In PSO mode, loads are completed in program order, but stores are allowed to pass earlier stores; therefore, only the MEMBAR at #1 is needed between updating data and the flag.

In RMO mode, there is no implicit ordering between memory accesses; therefore, the MEMBARs at both #1 and #2 are needed.

8.3.2 Memory Synchronization: MEMBAR and FLUSH

The MEMBAR (STBAR in SPARC-V8) and FLUSH instructions are provide for explicit control of memory ordering in program execution. MEMBAR has several variations; their implementations in UltraSPARC IIi 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.

8.3.2.1 MEMBAR #LoadLoad

Forces all loads after the MEMBAR to wait until all loads before the MEMBAR have reached global visibility.

8.3.2.2 MEMBAR #StoreLoad

Forces all loads after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility.

8.3.2.3 MEMBAR #LoadStore

Forces all stores after the MEMBAR to wait until all loads before the MEMBAR have reached global visibility.

8.3.2.4 MEMBAR #StoreStore and STBAR

Forces all stores after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility.

Note – STBAR has the same semantics as MEMBAR `#StoreStore`; it is included for SPARC-V8 compatibility.

Note – The above four MEMBARs do not guarantee ordering between cacheable accesses after noncacheable accesses.

8.3.2.5 MEMBAR #Lookaside

SPARC-V9 provides this variation for implementations having virtually tagged store buffers that do not contain information for snooping.

Note – For SPARC-V9 compatibility, this variation should be used before issuing a load to an address space that cannot be snooped.

8.3.2.6 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 non-cacheable accesses. For example, I/O accesses must be followed by a MEMBAR `#MemIssue` before subsequent cacheable stores; this ensures that the I/O accesses reach global visibility before the cacheable stores after the MEMBAR.

Note – MEMBAR `#MemIssue` is different from the combination of MEMBAR `#LoadLoad | #LoadStore | #StoreLoad | #StoreStore`. MEMBAR `#MemIssue` orders cacheable and noncacheable domains; it prevents memory accesses after it from issuing until it completes.

8.3.2.7 MEMBAR #Sync (Issue Barrier)

Forces all outstanding instructions and all deferred errors to be completed before any instructions after the MEMBAR are issued.

Note – MEMBAR `#Sync` is a costly instruction; unnecessary usage may result in substantial performance degradation.

8.3.2.8 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. LISP programs and dynamic linking require this behavior. SPARC-V9 provides the FLUSH instruction to synchronize instruction and data memory after code space has been modified.

In UltraSPARC IIi, a 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.

8.3.3 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 *data_access_exception* trap (with SFSR.FT=4, atomic to page marked non-cacheable). An atomic access with an unsupported ASI causes a *data_access_exception* trap (with SFSR.FT=8, illegal ASI value or virtual address). *Table 8-1* lists the ASIs that support atomic accesses

Table 8-1 ASIs that Support SWAP, LDSTUB, and CAS

ASI Name	Access
ASI_NUCLEUS{_LITTLE}	Restricted
ASI_AS_IF_USER_PRIMARY{_LITTLE}	Restricted
ASI_AS_IF_USER_SECONDARY{_LITTLE}	Restricted
ASI_PRIMARY{_LITTLE}	Unrestricted
ASI_SECONDARY{_LITTLE}	Unrestricted
ASI_PHYS_USE_EC{_LITTLE}	Unrestricted

Note – Atomic accesses with non-faulting ASIs are not allowed, because these ASIs have the load-only attribute.

8.3.3.1 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. A cache miss allocates the corresponding line.

Note – If a page is marked as virtually-non-cacheable but physically cacheable, allocation is done to the E-cache only.

8.3.3.2 LDSTUB Instruction

LDSTUB behaves like SWAP, except that it loads a byte from memory into an integer register and atomically writes all ones (FF_{16}) into the addressed byte.

8.3.3.3 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.

8.3.4 Non-Faulting Load

A non-faulting load behaves like a normal load, except that:

- It does not allow side-effect access. An access with the E-bit set causes a *data_access_exception* trap (with SFSR.FT=2, Speculative Load to page marked E-bit).
- It can be applied to a page with the NFO-bit set; other types of accesses will cause a *data_access_exception* trap (with SFSR.FT=10_{16}, Normal access to page marked NFO).

Non-faulting loads are issued with ASI_PRIMARY_NO_FAULT{_LITTLE}, or ASI_SECONDARY_NO_FAULT{_LITTLE}. A store with a NO_FAULT ASI causes a *data_access_exception* trap (with SFSR.FT=8, Illegal RW).

When a non-faulting 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 non-faulting 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, non-faulting loads allow the null pointer to be accessed safely in a read-ahead fashion if the OS can ensure that the page at virtual address 0_{16} is accessed with no penalty. The NFO (non-fault access only) bit in the MMU marks pages that are mapped for safe access by non-faulting 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 benefitting from the acceleration of non-faulting access in debugged library routines.

8.3.5 PREFETCH Instructions

UltraSPARC IIi has extensions to support the v9 Prefetch instruction. These extensions primarily address floating-point vector code, in which the software (compiler) can accurately schedule the prefetch of data sufficiently ahead of its usage, and in which execution is bounded by (E-cache) miss throughput.

UltraSPARC IIi allows loads and stores (E-cache-hits) to continue while a prefetch (E-cache-miss) is outstanding. An outstanding Prefetch does not block subsequent load or store hits.

This extension from UltraSPARC allows greater miss throughput. The UltraSPARC Load Buffer is designed such that a load with an E-cache-miss blocks subsequent load hits; these load-hits in turn block subsequent load misses. This tends to serialize load-misses.

However, Prefetch misses do *not* block subsequent load hits. Hence prefetches can be scheduled sufficiently far in advance of the associated Load (or Store) instruction, without interfering with subsequent loads and stores.

Prefetches appear as Loads that do not return data to a register. A prefetch request that is sent to the ECU checks the E-cache for the block. If the Prefetch hits in the E-cache, the operation will be complete; if it does not hit, the ECU requests that block from the Memory Control Unit (MCU). When the MCU returns the requested data, it is only written into the E-cache, not into the D-cache.

8.3.5.1 PREFETCH Behavior and Limitations

- All PREFETCH instructions are enqueued on the load buffer, except as noted below.
- Some conditions, noted below, cause an otherwise supported PREFETCH to be treated as a no-op and removed from the load buffer when it reaches the front of the queue.
- No PREFETCH will cause a trap except:
  - PREFETCH with *fcn*=5..15 causes an *illegal_instruction* trap, as defined in *The SPARC Architecture Manual, Version 9*.
  - Watchpoint, as defined in Section A.5, *Watchpoint Support* on page 368.
- Any PREFETCHA that specifies an internal ASI in the following ranges is not enqueued on the load buffer and is not executed:
  - 40_{16}..$4F_{16}$, 50_{16}..$5F_{16}$, 60_{16}..$6F_{16}$, 76_{16}, 77_{16}
- The following conditions cause a PREFETCH{A} to be treated as a NOP:
  - PREFETCH with *fcn*=16..31, as defined in *The SPARC Architecture Manual, Version 9*.
  - A *data_access_MMU_miss* exception
  - D-MMU disabled
  - For PREFETCHA, any ASI other than the following 04_{16}, $0C_{16}$, 10_{16}, 11_{16}, 18_{16}, 19_{16}, 80_{16}..83_{16}, 88_{16}..$8B_{16}$
  - Attempt to PREFETCH to a noncacheable page
  - fcn==16_{16}..31_{16}
- Alignment is not checked on PREFETCH{A}. The five least-significant address bits are ignored.

8.3.5.2 Implemented fcn Values

Table 8-2 lists the supported values for *fcn* and their meanings.

Table 8-2 PREFETCH{A} Variants

fcn	Prefetch function	Action
0	Prefetch for several reads	Generate DRAM read if the desired line is not E-cache-resident
1	Prefetch for one read	
4	Prefetch page	
2	Prefetch for several writes	Generate DRAM read if the desired line is not E-cache-resident
3	Prefetch for one write	
5-15	reserved	illegal-instruction trap
16-31	Implementation-dependent	no-op

For more information, including an enumeration of the bus transaction that each *fcn* value causes, see Section 14.4.5, *PREFETCH{A} (Impdep #103, 117)* on page 189.

8.3.6 Block Loads and Stores

Block load and store instructions work like normal floating-point load and store instructions, except that the data size (granularity) is 64 bytes per transfer. See Section 13.5.3, *Block Load and Store Instructions* on page 164 for a full description of the instructions.

8.3.7 I/O (PCI or UPA64S) and Accesses with Side-effects

I/O locations may not behave with memory semantics. Loads and stores may have side-effects; for example, a read access may clear a register or pop an entry off a FIFO. A write access may set a register address port so that the next access to that address will read or write a particular internal registers, etc. Such devices are considered order sensitive. Also, such devices may only allow accesses of a fixed size, so store buffer merging of adjacent stores or stores within a 16-byte region will cause an access error.

The UltraSPARC IIi MMU includes an attribute bit (the E-Bit) in each page translation, which, when set, indicates that access to this page cause side effects. Accesses other than block loads or stores to pages that have this bit set have the following behavior:

- Noncacheable accesses are strongly ordered with respect to each other
- Noncacheable loads with the E-bit set will not be issued until all previous control transfers (including exceptions) are resolved.
- Store buffer compression is disabled for noncacheable accesses.

- Non-faulting loads are not allowed and will cause a *data_access_exception* trap (with SFSR.FT = 2, speculative load to page marked E-bit).
- A MEMBAR may be needed between side-effect and non-side-effect accesses while in PSO and RMO modes.

8.3.8 Instruction Prefetch to Side-Effect Locations

UltraSPARC IIi does instruction prefetching and follows branches that it predicts will be taken. Addresses mapped by the I-MMU may be accessed even though they are not actually executed by the program. Normally, locations with side effects or those that generate time-outs or bus errors will not be mapped by the I-MMU, so prefetching will not cause problems. When running with the I-MMU disabled, however, software must avoid placing data in the path of a control transfer instruction target or sequentially following a trap or conditional branch instruction. Data can be placed sequentially following the delay slot of a *BA(,pt)*, *CALL*, or *JMPL* instruction. Instructions should not be placed within 256 bytes of locations with side effects. See Section 21.2.10, *Return Address Stack (RAS)* on page 335 for other information about JMPLs and RETURNs.

8.3.9 Instruction Prefetch When Exiting RED_state

Exiting RED_state by writing 0 to PSTATE.RED in the delay slot of a JMPL is not recommended. A noncacheable instruction prefetch may be made to the JMPL target, which may be in a cacheable memory area. This may result in a bus error on some systems, which will cause an *instruction_access_error* trap. The trap can be masked by setting the NCEEN bit in the ESTATE_ERR_EN register to zero, but this will mask all non-correctable error checking. To avoid this problem exit RED_state with DONE or RETRY, or with a JMPL to a noncacheable target address.

8.3.10 UltraSPARC IIi Internal ASIs

ASIs in the ranges $46_{16}..6F_{16}$ and $76_{16}..7F_{16}$ are used for accessing internal UltraSPARC IIi states. Stores to these ASIs do not follow the normal memory model ordering rules. Correct operation requires the following:

- A MEMBAR `#Sync` is needed after an internal ASI store other than MMU ASIs before the point that side effects must be visible. This MEMBAR must precede the next load or noninternal store. The MEMBAR also must be in or before the delay slot of a delayed control transfer instruction of any type. This is necessary to avoid corrupting data.

- A FLUSH, DONE, or RETRY is needed after an internal store to the MMU ASIs (ASI $50_{16}..52_{16}$, $54_{16}..5F_{16}$) or to the IC bit in the LSU control register before the point that side effects must be visible. Stores to D-MMU registers other than the context ASIs may also use a MEMBAR `#Sync`. One of these instructions must precede the next load or noninternal store. They also must be in or before the delay slot of a delayed control transfer instruction. This is necessary to avoid corrupting data.

8.4 Load Buffer

The load buffer allows the load and execution pipelines in UltraSPARC IIi to be decoupled; thus, loads that cannot return data immediately will not stall the pipeline but, rather, will be buffered until they can return data. For example, when a load misses the on-chip D-cache and must access the E-cache, the load will be placed in the load buffer and the execution pipelines will continue moving as long as they do not require the register that is being loaded. An instruction that attempts to use the data that is being loaded by an instruction in the load buffer is called a 'use' instruction.

The pipelines are not fully decoupled, because UltraSPARC IIi still supports the notion of precise traps, and loads that are younger than a trapping instruction must not execute, except in the case of deferred traps. Loads themselves can take precise traps, when exceptions are detected in the pipeline. For example, address misalignment or access violations detected in the translation process will both be reported as precise traps. However, when a load has a hardware problem on the external bus (for example, a parity error), it will generate a deferred trap since younger instructions, unblocked by the D-cache miss, could have been retired and modified the machine state. This may result in termination of the user thread or reset. UltraSPARC IIi does not support recovery from such hardware errors, and they are fatal. See Chapter 16, *Error Handling*.

8.5 Store Buffer

All store operations (including atomic and STA instructions) and barriers or store completion instructions (MEMBAR and STBAR) are entered into the Store Buffer.

8.5.1 Stores Delayed by Loads

The store buffer normally has lower priority than the load buffer when arbitrating for the D-cache or E-cache, since returning load data is usually more critical than store completion. To ensure that stores complete in a finite amount of time as required by SPARC-V9, UltraSPARC IIi eventually will raise the store buffer priority above load buffer priority if the store buffer is continually locked out by subsequent loads (other than internal ASI loads). Software using a load spin loop to wait for a signal from another processor following a store that signals that processor waits for the store to time out in the store buffer. For this type of code, it is more efficient to put a MEMBAR `#StoreLoad` between the store and the load spin loop.

8.5.2 Store Buffer Compression

Consecutive non-side-effect stores may be combined into aligned 8-byte entries in the store buffer to improve store bandwidth. Cacheable stores can only be compressed with adjacent cacheable stores, Likewise, noncacheable stores can only be compressed with adjacent noncacheable stores. In order to maintain strong ordering for I/O accesses, stores with the side-effect attribute (E-bit set) cannot be combined with any other stores.

The memory control unit can also compress consecutive 8-byte stores into single 16-byte UPA64S transactions.

8.6 Use of CP==1, CV==0 to Bypass the D-cache

The D-cache can return stale data if CP==1, CV==0 is used to bypass the cache, after use of CP==1 and CV==1, for loads and stores to a particular address.

The D-cache should be flushed after mixing use of any CP/CV settings for a physical address, including cacheable (DRAM) and noncacheable (IO) physical addresses. The term “noncacheable” in the user’s manual does not refer to “non-D-cacheable”. The term “virtually noncacheable” does refer to the “non-D-cacheable” CP==1, CV==0 case.

CP==1, CV==1:	Cacheable, Virtually-cacheable
CP==1, CV==0:	Cacheable, Virtually-noncacheable
CP==0, CV==1:	Not Used

CP==0, CV==0: Noncacheable

Only two entries in the D-cache need be flushed for each physical address {VA[13]==0,PA[12:0]} and {VA[13]==1,PA[12:0]}.

When a load with a physical address occurs, using ASI=0x14 (ASI_PHYS_USE_EC), causing CP==1 and CV==0, and the address hits in the D-cache, the data can come from the D-cache instead of from the E-cache .

Note that the manual has a caveat that is similar to this case: If CP==0 and CV==0, which indicates a "noncacheable" access, and the address is in the D-cache, data can be returned from the D-cache. Section 3.1.1.2, *Data Cache (D-cache)* on page 20 warns that the address should be flushed from the D-cache before changing its mapping.

Similarly, if CP==1, and CV==0, and the data is in the D-cache, data may be returned from the D-cache. However there are corner cases where it may not be returned.

For instance, with ASI_PHYS_USE_EC, the physical PA[13] is used to index the D-cache, where VA[13] would ordinarily be used. So the data might not be correctly returned if the real data were in VA[13]==0, but PA[13]==1. Ordinarily the rest of the PA bits will show a difference, so there is a miss in the D-cache, and a correct reference to the E-cache. This takes advantage of knowing that a valid PA can only exist in one VA[13] mapping at a time in the D-cache. Note that this depends on how the addresses were mapped earlier, when the line was installed in the D-cache.

This ASI_PHYS_USE_EC load hitting on the D-cache behavior is not defined or tested, so software should not rely on it.

When a store is done with a physical address, using ASI=0x14 (ASI_PHYS_USE_EC), causing CP==1 and CV==0, and the address hits in the D-cache D-cache, the D-cache apparently does get updated. However, this behavior is not verified or guaranteed. Again, software should make sure the physical address is not in the D-cache, before accessing that address using CP==1, CV==0, whether by a TLB mapping, or using one of the special ASIs.

CHAPTER 9

PCI Bus Interface

9.1 Introduction

This chapter describes the PCI Bus Interface Module (PBM) of UltraSPARC IIi.

The PBM is a 0–66 MHz 32-bit host-PCI bridge. The Advanced PCI Bridge (APB) provides an external connection to two 32-bit 0-33 MHz PCI busses. APB forwards transactions in both directions, between these primary and secondary PCI busses. Main features:

- Operates with a 2x PCI clock. (40–132 MHz)
- Single 64-byte DMA read/write buffers, single 64-byte PIO read/write buffer
- Little-endian to the bus and internal configuration space

9.1.1 Supported PCI features:

- 64-bit Addressing (Dual Address Cycle) for DMA bypass
- Required adapter and host-bridge configuration space header registers
- Fast Back-to-Back cycles as a DMA target
- Arbitrary byte enables (Consistent DMA)
- Ability to generate memory, I/O, and configuration read and write cycles
- Ability to generate special cycles
- Ability to receive memory cycles
- Peer-to-peer DMA on a single segment

9.1.2 Unsupported PCI features:

- Exclusive Access to main memory (LOCK)
- Peer-to-peer transfers between bus segments
- Cache support
- Cache-line Wrap Addressing Mode
- Fast Back-to-Back cycles as a PIO master
- Address/Data Stepping
- Subtractive decode
- Any DOS compatibility features

9.2 PCI Bus Operations

9.2.1 Basic Read/Write Cycles

Read and write transactions occur as specified in the PCI specification.

When a DMA burst transfer goes over a line (64 B) boundary, UltraSPARC IIi generates a disconnect. This disconnect normally causes the master device to reattempt the transaction at the address of the next untransferred data.

UltraSPARC IIi is capable of generating arbitrary byte enables on PIO writes. It can also generated aligned PIO reads of 1, 2, 4, 8, 16, and 64 bytes. A target device is required to drive all data bytes on reads, but is not required to support arbitrary byte enables on writes and may terminate the cycle with a target-abort if an illegal byte enable combination is signalled. UltraSPARC IIi supports arbitrary byte enables for all DMA transactions.

The PBM can accept Dual-Address-Cycles, using the 64-bit address in bypass mode. UltraSPARC IIi does not generate 64-bit PIO cycles or PIOs with DACs.

9.2.2 Transaction Termination Behavior

- **Retries:** For PIO transactions, a count is kept of the number of retries for a given transaction. When this value exceeds the Retry Limit Count the PBM ceases to attempt the transaction and issues an interrupt to the processor. The Retry Limit Count is fixed at 512.

- **Disconnects:** The difference between a disconnect and a retry is that there is no data transferred during a retry; otherwise, the signalling is the same. No count is kept of disconnects. The transaction is restarted with the next untransferred data.
- **Master-aborts:** A *master-abort* typically happens when no device responds to the PIO address.
- **Target-aborts:** A *target-abort* may be received for a variety of error conditions. All cases for which UltraSPARC IIi may signal a target-abort are given in Chapter 16, *Error Handling*.

9.2.3 Addressing Modes

Only the Linear Incrementing addressing mode is supported. Reserved and Cache Line Wrap address mode accesses are disconnected after the first data phase, allowing the master to complete the transfer one data word at a time.

9.2.4 Configuration Cycles

UltraSPARC IIi generates both Type 0 and Type 1 configuration accesses. The type generated depends on the bus number field within the configuration address. UltraSPARC IIi hardwires its Bus Number to 0. See Section 19.3.1, *PCI Configuration Space* on page 289 for details.

Compatibility Note – If Configuration cycles are generated with compressed (E-bit==0) byte or halfword stores, or with random byte enable patterns using the PSTORE instruction, UltraSPARC IIi does not guarantee that AD[1:0] points to the first byte with a BE asserted.

Also, while not addressed by the PCI 2.1 specification UltraSPARC IIi can generate multi-databeat configuration reads and writes.

9.2.5 Special Cycles

UltraSPARC IIi ignores Special Cycles and does not generate them.

9.2.6 PCI INT_ACK Generation

UltraSPARC IIi can generate an interrupt acknowledge in response to a PCI Interrupt. See Section 19.3.4, *PCI INT_ACK Generation* on page 309 for the method of generating this transaction.

9.2.7 Exclusive Access

UltraSPARC IIi does not implement locking and the LOCK# signal is not connected. Any exclusive access proceeds as if it were a non-exclusive access.

9.2.8 Fast Back-to-Back Cycles

UltraSPARC IIi is capable of handling Fast Back-to-Back DMA transactions as a target device. The Fast Back-to-Back Capable bit in the Status register is hardwired to '1'. It handles the master-based mechanism (as required) and is capable of decoding the target-based mechanism as well. The address is checked and UltraSPARC IIi does not reply to masters presenting an invalid address.

The specification requires that TRDY#, DEVSEL#, and STOP# be delayed by one cycle unless this device were the target of the previous transaction. This delay causes writes to be extended by a cycle but is hidden on reads.

There is little performance gain except for reads that follow writes, but support is provided for third party devices that choose to implement this feature.

UltraSPARC IIi is not capable of generating Fast Back-to-Back PIO transactions and does not implement the Fast Back-to-Back enable bit in the Command Register in the configuration header.

A Fast Back-to-Back PIO would remove the idle cycle between two transactions to the same target as long as the first transaction were a write. Alternately stated, it would insert an idle cycle between transactions to different targets and after read transactions. UltraSPARC IIi does not support this sequence.

9.3 Functional Topics

9.3.1 PCI Arbiter

9.3.1.1 Arbitration Schemes

Two arbitration schemes are implemented in the UltraSPARC IIi and APB on-chip PCI arbiters. The default condition is fair arbitration, where all enabled requests are serviced in “round-robin” fashion. The second condition (enabled by the ARB_PRIO bits in the PCI Control Register) gives higher priority to a specific request. This allows the device attached to that pair to claim, at most, every other PCI transaction.

Additionally, a transaction that is Retried gets the highest priority the next time it asserts its request. Only one request at a time is given this high priority. The high priority remains in effect until the request is accepted without Retry.

9.3.1.2 Bus Parking

The ARB_PARK bit in the PCI Control Register causes the last GNT to remain asserted when no other requests are asserted. This results in a saving of one clock cycle for bursts of transactions from the same device.

9.3.2 PCI Commands

Table 9-1 lists the commands that the UltraSPARC IIi PBM generates.

Table 9-1 PCI Command Generation

Command	C/BE#	Generate?	Notes
Interrupt Acknowledge	0000	Yes	
Special Cycle	0001	Yes	
I/O Read	0010	Yes	
I/O Write	0011	Yes	
Reserved	0100	No	
Reserved	0101	No	

Table 9-1 PCI Command Generation *(Continued)*

Command	C/BE#	Generate?	Notes
Memory Read	0110	Yes	Perform read access, no prefetch
Memory Write	0111	Yes	Perform write access
Reserved	1000	No	
Reserved	1001	No	
Configuration Read	1010	Yes	
Configuration Write	1011	Yes	
Memory Read Multiple	1100	Yes	Perform read with 8 byte prefetch
Dual Address Cycle	1101	No	
Memory Read Line	1110	Yes	Perform read with 64 byte prefetch
Memory Write & Invalidate	1111	No	

Table 9-2 lists the commands to which UltraSPARC IIi responds as a Target.

Table 9-2 PCI Command Response

Command	C/BE#	Response
Interrupt Acknowledge	0000	Ignored
Special Cycle	0001	Ignored
I/O Read	0010	Ignored
I/O Write	0011	Ignored
Reserved	0100	Ignored
Reserved	0101	Ignored
Memory Read	0110	Perform read access. 64-byte prefetch if to memory; 16-byte prefetch if to UPA64S
Memory Write	0111	Perform write access
Reserved	1000	Ignored
Reserved	1001	Ignored
Configuration Read	1010	Ignored
Configuration Write	1011	Ignored
Memory Read Multiple	1100	Perform read with 64 byte prefetch

Table 9-2 PCI Command Response *(Continued)*

Command	C/BE#	Response
Dual Address Cycle	1101	Bypass access
Memory Read Line	1110	Perform read with 64 byte prefetch
Memory Write & Invalidate	1111	Equivalent to Memory Write command

Note – All PCI DMA reads to UPA64S address space cause 64-byte read transactions on the UPA64S. This action may cause unwanted prefetch effects. All DMA writes to UPA64S address space cause a succession of 1-16-byte UPA64S writes.

9.4 Little-endian Support

9.4.1 Endian-ness

The UltraSPARC IIi internal, UPA64S, and DRAM system interfaces are big-endian, That is, the address of a word (or quadword, doubleword, or halfword) is the address of its most significant byte. The PCI bus is little-endian, where the word (or quadword, doubleword, or halfword) address is the address of the least significant byte. See the section *Addressing Conventions* in Chapter 6 of *The SPARC Architecture Manual, Version 9* for a detailed explanation of this topic. To route the byte lanes logically correctly, the UltraSPARC IIi main internal data busses are connected to the PCI bus in a "byte-twisted" fashion. In particular, UltraSPARC IIi data bits [63:56] are connected to the PCI data bits [7:0], UltraSPARC IIi bits [55:48] map to PCI bits [15:8], an so on. The PBM internal control registers, which are big-endian, are byte-twisted again internally.

This implementation causes all byte-sized PIOs and byte-stream DMA to be handled correctly. It, along with other features built into SPARC V9 processors, allows all PIO and DMA activity to and from the PCI bus to take place correctly.

9.4.2 Big- and Little-endian regions

9.4.2.1 Address Space

The UltraSPARC IIi 8-gigabyte address space consists of several regions. The lower 16 MB, from 0x1FE.0000.0000 to 0x1FE.00FF.FFFF allows access to internal registers within UltraSPARC IIiIO This portion of the address space is big-endian and there is no byte twisting done for accesses within this range.

There is a large region of unused/reserved address space from 0x1FE.0202.0000 to 0x1FE.FFFF.FFFF. Reads to this address range return zero and writes are simply ignored.

The remaining address regions are little-endian. The upper 4 gigabytes, from 0x1FF.0000.0000 to 0x1FF.FFFF.FFFF are used for accesses to PCI bus memory space. The 16-megabyte region from 0x0.0100.0000 to 0x0.01FF.FFFF is used for access to PCI configuration space, and there are two 64-kilobyte regions from 0x0.0200.0000 to 0x0.02FF.FFFF that are used to access PCI bus I/O space. All of these address ranges are little-endian, and all accesses to them use byte twisting.

Note – This means that any configuration and status registers in the APB ASIC must be accessed with little-endian loads and stores, or they will appear byte twisted. All configuration and status registers within UltraSPARC IIi are accessed with big-endian loads and stores, except for those used to access the PCI configuration space.

If the UltraSPARC IIi PCI bridge ASIC provides the path to the system PROM, the PROM is found between offsets 0x1FF.F000.0000 and 0x1FF.F0FF.FFFF. This range falls in the upper 4-gigabyte region, that UltraSPARC IIi considers as little-endian, and subjects to byte-twisting. In spite of the byte-twisting, and because of the way the PROM is programmed, this PROM appears to the system correctly as a big-endian device. An explanation of this mechanism is detailed in succeeding sections.

9.4.2.2 Byte Twisting

Figure 9-1 shows how data is manipulated from a 32-bit little-endian PCI bus to 64-bit big-endian UltraSPARC IIi busses.

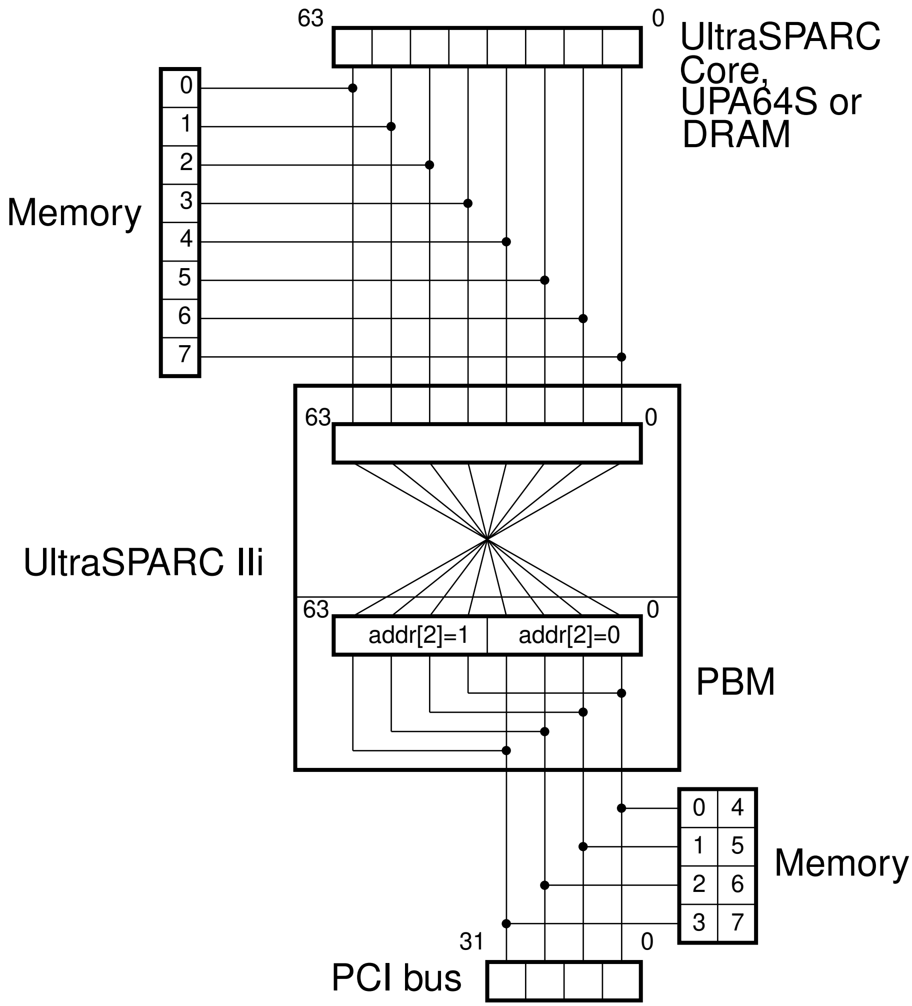


Figure 9-1 UltraSPARC IIi Byte Twisting

9.4.3 Specific Cases

9.4.3.1 PIOs

Normal

All byte sized PIOs work correctly. The byte lane used for a given address on the big-endian side is directly wired to the byte lane used for that address on the little-endian side.

Byte twisting is insufficient for any access larger than a byte. For example, if the 32-bit value 0x12345678 is written to a 32-bit register on a PCI device, the PCI device sees the value 0x78563412 instead.

The UltraSPARC core has special support to correct this By either marking the page containing the PCI register as little-endian in the processor's MMU, or by using one of the little-endian ASIs, UltraSPARC IIi will alter its ordering of the bytes so that the PCI device correctly sees 0x12345678.

PROM accesses

Instruction fetches from the PROM are a special case because they are unable to use the little-endian features. PROM instruction fetches, like all instruction fetches, are always done in big-endian mode.

In UltraSPARC IIi systems, the PROM could be a byte device on an 8 byte bus, controlled by an integrated IO controller (or SuperIO) IC. This SuperIO could stack the bytes in little-endian format, such that the byte at address 0 in the PROM appears on PCI bus data bits 7:0, byte 1 on bits 15:8, and so on. To function correctly with the byte-twisting of UltraSPARC IIi, and in the absence of any other byte reordering by the processor, the PROM must be programmed in big-endian order – byte 0 in the PROM should be the MSB of the first instruction.

Because of this required byte programming ordering for the PROM, data accesses to the PROM should not use the little-endian byte reordering of the processor, even though the PROM is located within the little-endian PCI space.

If only big-endian accesses are made to the PROM, PIOs of any size will return data with the correct byte order.

Note that use of a SuperIO IC may require different ordering of the bytes in the PROM to make UltraSPARC IIi references work correctly.

9.4.3.2 DMA

Data streams

DMA of byte streams works correctly without further intervention. A PCI device that receives the byte stream (01,02,03,04) packs the bytes into a 32-bit register starting with the LSB of the register, that is, 0x04030201. After transferring to memory on the PCI bus, the value 0x01 occurs at the lowest memory location, as required.

After byte twisting, the value given to the UltraSPARC core would be 0x01020304. Since the MSB is the lowest memory location, the value 0x01 is still stored at the lowest memory location, as required.

Descriptors

Byte twisting is insufficient for any access larger than a byte, just as for PIOs. With byte twisting used alone, a DMA descriptor access would retrieve the wrong byte ordering. For example, if the value 0x12345678 were set up as an address in a descriptor, the PCI device interprets this value as 0x78563412 instead.

To avoid this, the UltraSPARC core little-endian features are used again. Processor loads and stores to the descriptors should be specified as little-endian. This will re-order the bytes in memory so that after byte twisting, the PCI device sees the correct value.

CHAPTER **10**

UltraSPARC IIi IOM

The IO Memory Management Unit (IOM) performs virtual to physical address translation during DVMA cycles. PCI master devices provide a 32-bit virtual address at the beginning of a DVMA transfer, which the IOM translates into 34 bits of physical address.

The UltraSPARC IIi CPU contains 16-entry fully-associative Translation Lookaside Buffers (TLBs) and a a one-level, software-managed data structure called a Translation Storage Buffer(TSB). The TLB stores recently used translation information. Hardware performs a TSB lookup (also known as hardware table walk) when the translation cannot be found in the TLB. If a TSB lookup fails to locate a valid mapping, the IOM returns an error to the PCI master device.

The IOM supports alternative page sizes of 8K and 64K. Mixed page sizes can be used in the system but the TSB table lookup assumes the smaller page size. No page overlapping is allowed. Operation in Bypass mode allows devices with their own translation facility to bypass IOM.

10.1 Block Diagram

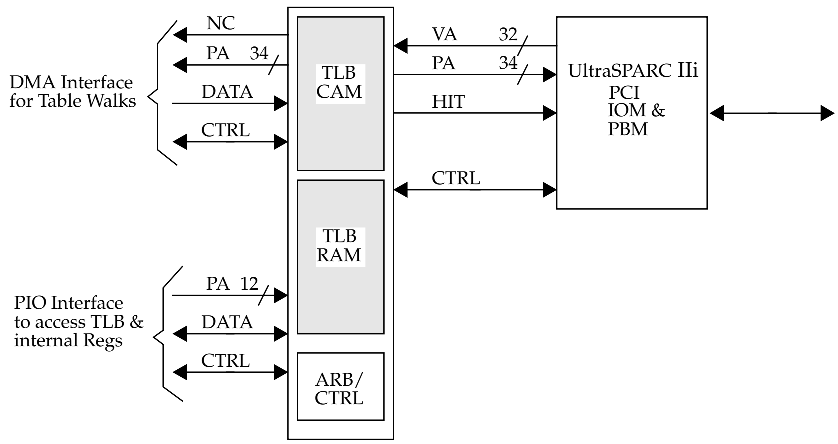


Figure 10-1 IOM Top Level Block Diagram

10.2 TLB Entry Formats

A TLB entry consists of TLB tag in the CAM and TLB data in the RAM.

10.2.1 TLB CAM Tag

24 23	22 21	20	19	18	0
ERRSTS	ERR	W	S	SIZE	VA[31:13]

Figure 10-2 TLB CAM Tag Format

Figure 10-2 shows the bit fields of the TLB CAM Tag. These assignments are explained in *Table 10-1*.

Table 10-1 Description of TLB Tag Fields

Field	Bits	Description	Type
ERRSTS	24:23	Error Status: 00 = Reserved 01 = Invalid Error 10 = Reserved 11 = UE Error (on TTE read)	RW
ERR	22	When set to 1, indicates that there is an error associated with this entry	RW
W	21	Writable; when set, the page mapped by this TLB has write permission.	RW
S	20	Stream; Ignored by UltraSPARC IIi	RW
SIZE	19	0 means 8K page, 1 means 64K page	RW
VA [31:13]	18:0	19-bit VPN (Virtual Page Number)	RW

For an IOM miss, if the returned TTE data has Valid = 0, or lacks the appropriate write privilege, or has an uncorrectable ECC error (UE), the IOM adjusts the ERR_STS[1:0] to reflect the error, and sets ERR == 1 and Valid == 1.

The error is reported by the DMA master as a Target Abort. The PBM will also log its target-abort generation with the STA bit in the PCI Configuration Space Status Register.

The Valid bit for the entry is set, regardless of the state of the valid bit in the TTE data, so the DMA transaction does not cause another IOM miss.

Software is responsible for flushing the IOM entry when it rectifies the missing TSB entry or bad DMA address.

If a VA hit results in a protection error, the IOM state is not modified.

10.2.2 TLB RAM Data

30	29	28	27:21	20 — 0
V	U	C	*0s or 1s*	PA[33:13]

Figure 10-3 TLB RAM Data Format

Table 10-2 TLB Data Format

Field	Bits	Description	Type
V	30	Valid bit; when set, the TLB data field is meaningful	RW
U	29	Used bit; affects the LRU replacement.	RW
C	28	Cacheable bit; 1=Cacheable access; 0=Non-cacheable.	RW
PA[40:34]	27:21	Not stored; all 1s if Noncacheable; all 0s if cacheable.	R
PA[33:13]	20:0	21-bit physical page number	RW

10.3 DMA Operational Modes

There are three different operational DMA IOM modes: translation, bypass, and pass-through. The applicable mode depends upon:

- The value of the "MMU_EN" bit of the IOM Control Register
- The PCI addressing mode used: DAC using 64 bits or SAC using 32 bits
- The PCI virtual address – bits 31:29 in SAC mode or bits 63:50 in DAC mode

Table 10-3 PCI DMA Modes of Operation

Mode	ad[31:29]	MMU_EN	Addr<63:50>	Result
SAC	miss	X	N/A	PCI peer-to-peer (Ignored by UltraSPARC IIi)
SAC	hit	0	N/A	Pass-through
SAC	hit	1	N/A	IOM Translation (DMA)
DAC	X	X	0x0000-0x3FFE	Ignored by UltraSPARC IIi
DAC	X	X	0x3FFF	Bypass (DMA)

The Target Address Space Register is used to decide if AD[31:29] is a hit.

10.3.1 Translation Mode

The PBM block initiates the translation by providing a 32-bit virtual address. The IOM hardware performs the following actions in order, beginning with a TLB lookup, until a valid mapping or an error results.

1. If the lookup results in TLB hit, the IOM returns a 34 bit physical address.
2. If a TLB miss occurs, hardware automatically starts a TSB lookup.
3. If the TSB lookup locates a valid mapping for the virtual page, information in the TSB entry is loaded into the TLB and translation continued.
4. If the TSB lookup results in a miss, an error is returned to the PBM.

The virtual address consists of two fields: virtual page number and page offset. Page offset is from virtual address to physical address. The conversion of virtual address to physical address for page sizes 8K and 64K is shown below.

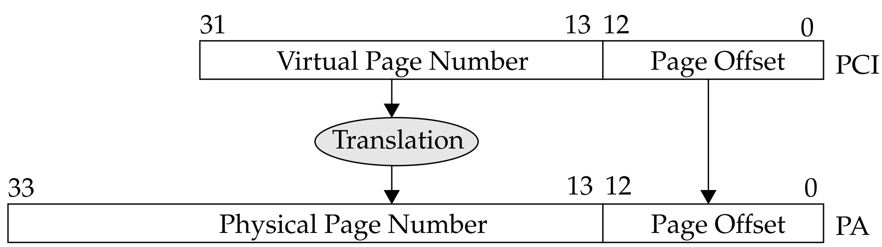


Figure 10-4 Virtual to Physical Address Translation for 8K Page Size

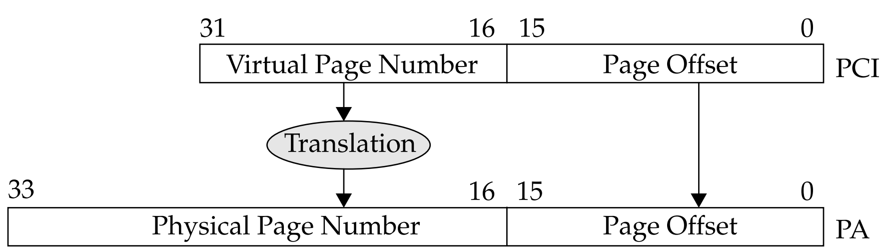


Figure 10-5 Virtual to Physical Address Translation for 64K Page Size

10.3.2 Bypass Mode

The IOM allows PCI devices to have their own MMU and bypass the IOM supported by the system. A PCI device is operating in bypass mode if all conditions in the last row in *Table 10-3* are met. In this mode, the physical address PA[33:0] = PCI_ADDR[33:0].

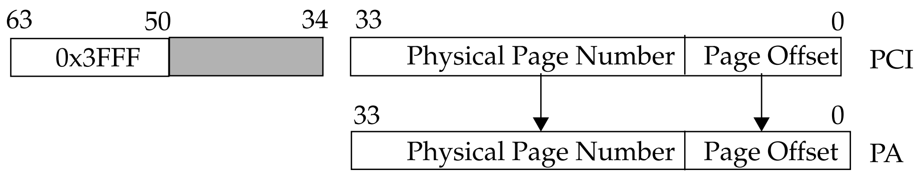


Figure 10-6 Physical Address Formation in Bypass Mode (8K and 64K)

A PCI device operating in bypass mode has direct access to the entire physical address space. Bit [34] of PCI_ADDR indicates whether the PCI device is accessing the coherent space, where (PA[34] = 0), or the UPA64S or IO space, where (PA[34] = 1).

10.3.3 Pass-through Mode

The IOM operates in pass-through mode if all conditions listed in the first row in *Table 10-3* are met. Pass-through mode allows access to the coherent address space (DRAM) only. Higher bits of physical address are padded with 0.

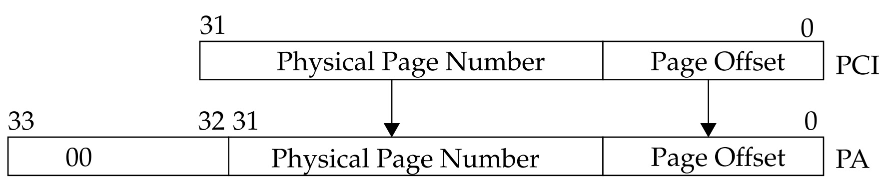


Figure 10-7 Physical Address Formation in Pass-through Mode (8K and 64K)

10.4 Translation Storage Buffer

The Translation Storage Buffer, or TSB, is a translation table in memory. It contains one-level mapping information for the virtual pages. IOM hardware looks up this table if a translation cannot be found in the TLB. A TSB entry is called Translation Table Entry, or TTE, and is eight bytes long.

The system.supports several TSB table sizes and specifies the size with the TSB_SIZE field of the IOM Control Register. The possible table sizes are 1K, 2K, 4K, 8K, 16K, 32K, 64K and 128K entries (not bytes) which supports DMA address space of 8M to 1G for an 8K page, and 64K to 2G for a 64K page (128K and 64K TSB sizes are not supported with a 64K page). Software must set up the TSB before it allows translation to start.

10.4.1 Translation Table Entry

Translation Table Entries (TTE) contain translation information for virtual pages. The IOM hardware reads one TTE during a table walk and stores it in the TLB. A TTE entry has valid information only when the DATA_V bit is set. *Table 10-4* shows the contents of the TTE.

Table 10-4 TTE Data Format

Field	Bits	Description
DATA_V	<63>	Valid bit (1 = TTE entry has valid mapping)
DATA_SIZE	<61>	Page size of the mapping (0 = 8K; 1 = 64K)
STREAM	<60>	Stream bit (1 = streamable page; 0 = consistent page)
LOCALBUS	<59>	Local bus bit; not used
DATA_SOFT_2	<58:51>	Reserved for software use
DATA_PA	<40:13>	Contains bits <33:13> of physical address; bits 15:13 are not used for 64K page; bits <40:34> are not used and implied to be 1 if noncacheable, 0 if cacheable.
DATA_SOFT	<12:7>	Reserved for software use
CACHEABLE	<4>	Cacheable (1 = cacheable page, 0 = non-cacheable page); not used
DATA_W	<1>	Set if this page is writable

TTE data is stored in main memory, in the software-managed TSB. All other bits are reserved.

10.4.2 TSB Lookup

During the TSB lookup, the physical address for the TTE entry is formed based on the following information.

- Base address of the TSB table
- Page size assumed during TSB lookup (as specified by the TBW_SIZE bit in IOM Control Register)
- TSB table size

The TSB Base Address Register contains the physical address of the first TTE entry in the TSB table. The lower order 13 bits of this register are all zeros because the TSB table must be aligned on an 8K boundary regardless of TSB size. Physical address for an entry in TSB table is formed by adding the base address and an offset generated as shown in *Table 10-5*. The lower order three bits of the offset are set to 0x0 because each TTE entry is eight bytes long.

Table 10-5 Offset to TSB Table

TSB Table Size	N	Offset (8K TSB lookup page size) (TBW_SIZE=0)	Offset (64K TSB lookup page size) (TBW_SIZE=1)
1K	12	[VA<22:13>, 000]	[VA<25:16>, 000]
2K	13	[VA<23:13>, 000]	[VA<26:16>, 000]
4K	14	[VA<24:13>, 000]	[VA<27:16>, 000]
8K	15	[VA<25:13>, 000]	[VA<28:16>, 000]
16K	16	[VA<26:13>, 000]	[VA<29:16>, 000]
32K	17	[VA<27:13>, 000]	[VA<30:16>, 000]
64K	18	[VA<28:13>, 000]	Not allowed[1]
128K	19	[VA<29:13>, 000]	Not allowed[1]

1. UltraSPARC IIi does not detect illegal combinations, and its behavior is unspecified for such combinations. Software must ensure they do not occur.

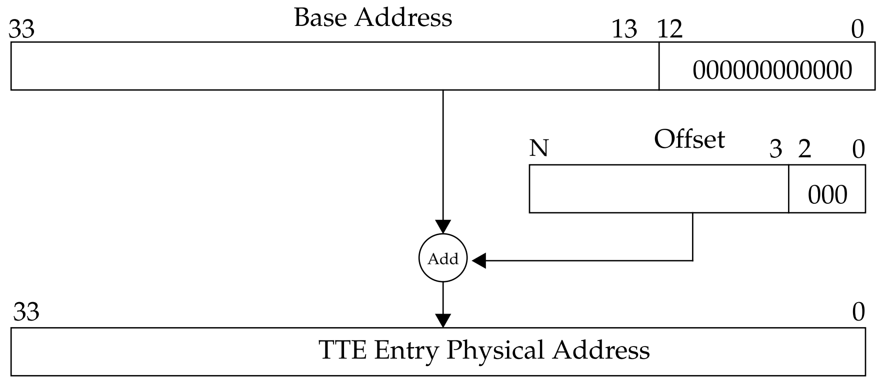


Figure 10-8 Computation of TTE Entry Address

TBW_SIZE should be set to 0 if 8K page size or mixed (8K and 64K) page sizes is used for DMA mappings. If mixed page sizes is used, each 64K page will use up 8 entries of TTE. Software must fill all 8 entries with the same information.

10.5 PIO Operations

To prevent random PIO operations from interfering with the internal states of the translation, the IOM implements an interlocking mechanism. This mechanism is described below.

- No PIO operation to the IOM is allowed during address translation for any DMA operation.
- No PIO operation to the IOM is allowed during service of TLB Miss.
- For a pending PIO request, the IOM begins the PIO operation once it completes the current translation or TLB miss service. In other words
- When the IOM is in idle state, it gives higher priority to PIO requests than address translations.

10.6 Translation Errors

Translation errors detected by the IOM are:

- **Invalid Errors:** An invalid error happens if bit DATA_V in the TTE read by IOM hardware indicates that the TTE is invalid (DATA_V = 0).
- Protection Errors: A protection error is detected if the PCI device is doing DMA write to a page which is mapped as read-only (bit W = 0 in the TLB tag or bit DATA_W = 0 in the TTE).
- **TTE UE Error:** If a correctable ECC error occurred during table walk, the MCU will correct the error and the TTE received by the IOM is error free. If the ECC error is uncorrectable, the received TTE will be invalid and the IOM will flag an error.

Compatibility Note – There are no time out errors during table walk for the UltraSPARC IIi IOM.

Compatibility Note – Bits in the DMA UE AFSR/AFAR are set, and the PA of the TTE entry is saved on Invalid, Protection (IOM miss), and TTE UE errors. This should aid debugging of software errors. If the Protection error had an IOM hit, the translated PA from the IOM is saved instead of the PA of the TTE entry. This may occur if a prior DMA read caused the IOM entry to be installed.

10.7 IOM Demap

After establishing mapping between virtual and physical addresses, implementing a change must include a demap of this existing mapping before a new mapping can be used by the device. Demap is required when taking down existing mapping to make physical memory available to other virtual addresses, or when changing access permission for a page.

During IOM demap, the PCI device is not allowed to use the page being demapped. If a device attempts to access a page currently being demapped, unexpected results may occur. The following events are needed to demap a page in the IOM.

- TSB entry properly updated with new information
- TLB flush performed with virtual page number

TLB flush is initiated by writing to the IOM Flush Address Register with the specified virtual page number. Match criteria are different for 8K and 64K page sizes. Hardware performing the flush adjusts matching criteria based on the page size. The matched entry in the TLB will be marked invalid.

10.8 Pseudo-LRU replacement algorithm

Compatibility Note – Prior PCI-based UltraSPARC systems implemented a true LRU scheme.

The UltraSPARC IIi IOM uses a 1-bit LRU scheme, just like the UltraSPARC MMUs. Each TLB entry has an associated “Valid,” and “Used” bit. On an automatic write to the TLB after a hardware tablewalk, the TLB picks the entry to write based on the following rules:

1. If any entry is not Valid, the first such entry is replaced (measuring from TLB entry 0). If not, then:
2. If any entry is not Used, the first such entry is replaced (measuring from TLB entry 0). If not, then:
3. All but one Used bit will be reset, then the process is repeated from Step 2 above.

All replacements can also be forced to a single entry.

10.9 TLB Initialization and Diagnostics

The IOM provides direct access to its internal resources, such as TLB Tag, TLB Data, and Match Comparison Logic.

After power is turned on, the contents of the IOM are undefined. Before any DMA is allowed to use the IOM, all TLB entries must be invalidated by writing 0s to them.

CHAPTER 11

Interrupt Handling

11.1 Overview

The "Mondo" interrupt transfer mechanism for Sun4u systems reduces interrupt service overhead by directly identifying the unique interrupter, without polling multiple status registers.

SPARC V9 CPUs provide a dedicated set of registers to be used exclusively for servicing interrupts. This eliminates the need for the processor to save its current register set to service an interrupt, and then restore it later.

An interrupt packet contains a Mondo vector which has three double words designed to assist the processor in servicing the interrupt.

Limitations of the Mondo vector approach include:

- Only one interrupt request packet can be serviced at a time.
- There is no priority level associated with Mondo vector interrupts; they are serviced on a first come, first served basis.

This interrupt packet delivery now happens inside UltraSPARC IIi, rather than being visible on the UPA interconnect. Since it is an internal dedicated uniprocessor path, the flow control issues are simpler, and no interrupt retry is needed. UltraSPARC IIi just causes one interrupt packet delivery at a time, after each acknowledgment by software (clearing of the MVR_BUSY bit in the mondo receive trap handler).

11.1.1 Mondo Dispatch Overview

UltraSPARC IIi's PIE logic block is responsible for fielding interrupts from external PCI sources, other external sources, and internal UltraSPARC IIi sources, loading the mondo data receive registers, and signalling a mondo receive trap to the UltraSPARC IIi pipeline.

External interrupt sources include 8 PCI slots on two separate PCI busses, the onboard IO devices, a graphics interrupt, and the expansion UPA slot.

These interrupts are concentrated in an external ASIC and presented to the Mondo Unit one at a time. This saves pins on UltraSPARC IIi.

Internal interrupt sources include ECC (errors) and PBM (PCI bus errors).

Each of the 8 PCI slots have 4 interrupts. However, with the current RIC chip, only 26 PCI interrupt requests can be connected.

The documentation assumes these interrupts are mapped to certain slots and INTA-D wires. System designers are free to distribute the PCI interrupt wires differently, but system software will need a new mapping of PCI slots, and related CSRs.

The CSRs and logic are implemented so that 32 PCI interrupts can be handled, if required, using a new RIC IC.

11.2 Mondo Unit Functional Description

11.2.1 Mondo Vectors.

The Sun4u architectural specification states that interrupts are delivered to the process potentially using three double words used to carry "pertinent" information. Note that UltraSPARC IIi does not deliver interrupt data, only the Interrupt Number. Reads of Mondo Data Receive registers 1 and 2 always return 0.

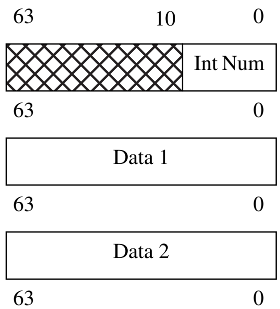


Figure 11-1 Mondo Vector Format

The first data register contains the interrupt number (11 bits).

The interrupt number is specific to each interrupt source.

The CPU can process only one interrupt at a time. The Mondo Dispatch Unit is responsible for remembering all interrupts that have arrived, and serializing them to the CPU pipeline as traps. In addition, it tracks the state of pending DMA writes in the APB and UltraSPARC IIi, and guarantees that all DMA writes completed on the Secondary PCI buses (temporally) before a PCI interrupt request, complete to memory before notifying the CPU.

11.2.1.1 DMA synchronization

After receiving a any external interrupt request, the PIE checks whether the two SB_EMPTY lines are asserted, indicating no pending DMA writes inside external APB ASICs.

If SB_EMPTY, the PIE then checks there are no pending DMA writes to the MCU.

If either empty indication were false, the PIE asserts SB_DRAIN, blocking arrival of future DMA writes (some may arrive during the transmission time). The PIE then waits for both SB_EMPTY assertions, and then further waits for the internal EMPTY assertion. At this point the trap may be delivered, and all other pending interrupts marked as “synchronized”, so that this process is again unnecessary when these arrive at the CPU.

The PIE deasserts SB_DRAIN once it sees that DMA writes are successfully cleared from both APB and the MCU/PBM.

SB_DRAIN does not have to block any other external PCI activity, as long as the SB_EMPTY and MCU/PBM DMA activity signals only reflect the status of pending DMA writes.

There is no deadlock, since the MCU can only forward DMA writes to slave devices, i.e. memory and UPA64S.

There is a read-only CSR available that causes this DRAIN-EMPTY protocol to be activated by a noncacheable load. The load does not complete until the DRAIN-EMPTY synchronization protocol completes. This allows software to synchronize against outstanding DMA writes when there is a standard PCI bus bridge beyond the APB. (First issue a PIO read to the far bus bridge, then after completion, synchronize against APB and UltraSPARC IIi using the CSR read).

11.2.1.2 Interrupt Number Register

Generally, each interrupt source has an Interrupt Number Register (INR) associated with it. The INR is either fully or partially software programmable and contains the Interrupt Number and a valid bit which enables or disables the interrupt.

31	30 26	25 11	10 0
V	Target Processor	Reserved	Interrupt Number

Figure 11-2 Full INR Contents

As shown the INR has 3 fields:

1. Valid bit (1 bit) - enables the interrupt when set to 1. Note that when an interrupt is present and the valid bit is 0, the interrupt is prevented from being delivered. However, once the valid bit is set to 1, the interrupt is delivered.
2. Target Processor (5 bits) - Read-only as 0 for UltraSPARC IIi.
3. Interrupt Number (11 bits)

For most interrupts, the Interrupt Number field is further broken down into two separate fields: the Interrupt Group Number (IGN) and the Interrupt Number Offset. The Interrupt Number Offset (INO) is a fixed value depending on the interrupt.

Compatibility Note – The IGN on UltraSPARC IIi is not programmable, and fixed to 0x1F.

31	30–26	25–11	10–6	5–0
V	Target Processor	Reserved	Int. Group. Number	Int. Num. Offset

Figure 11-3 Partial INR Contents

External Interrupts

External Interrupts refer to those interrupts that are generated external to UltraSPARC IIi. All external sources for interrupts (PCI, OBIO, Graphics, and UPA64S) go through the Interrupt Concentrator, a RIC ASIC.

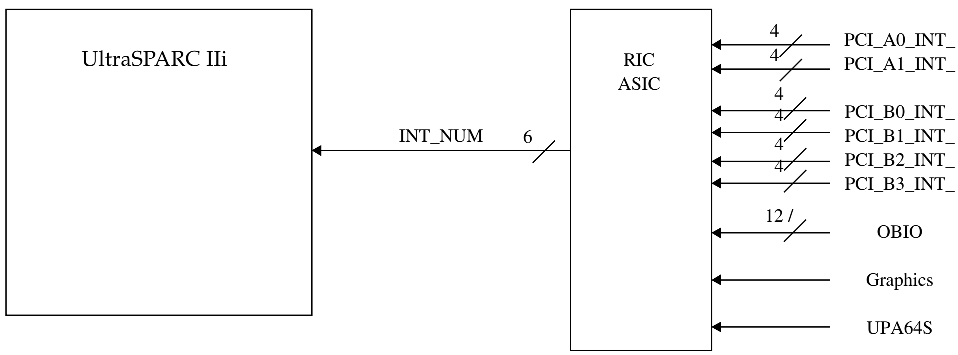


Figure 11-4 Interrupt Concentrator

The Interrupt Concentrator simply samples all interrupts lines in round-robin fashion, and presents one of them at a time to UltraSPARC IIi. To save package pins, the 38 interrupt lines are simply encoded into a 6 bit value that passes to UltraSPARC IIi.

- PCI - UltraSPARC IIi supports 8 total PCI slots on two separate busses. Each PCI slot has 4 interrupt lines. RIC only supports 26 of these.
- On-board IO Devices (OBIO) - There are 12 interrupts from OBIO devices.
- Graphics/UPA - 2 UPA slot interrupts are supported. These are the only two interrupts that are of pulse type (see below). These are also the only interrupts with the complete, fully software programmable, INR register. All other interrupts have IGN and INO fields.

11.2.1.3 Priority

Each interrupt has a priority associated with it. There are eight priority levels. priority 8 is the highest and priority 1 is the lowest.

Priority is taken into account during interrupt arbitration. When multiple interrupts are present, the highest priority interrupt is delivered first. If multiple interrupts with the same priority are present, they are delivered in a round-robin fashion. When all interrupts at the highest priority level are delivered, the next highest priority level is processed.

Table 11-1 Interrupt Receiver State Register

Level	Number of Interrupts	Source
8	6	Audio Record, Power Fail, Floppy, UE ECC, CE ECC, PBM error
7	6	Kbd/mouse/serial, Serial Int, Audio Playback PCI_A0_INTA#, PCI_A1_INTA#
6	6	PCI_B0_INTA#, PCI_B1_INTA# PCI_B2_INTA#, PCI_B3_INTA# PCI_A2_INTA#,PCI_A3_INTA#
5	7	OB Graphics, UPA64S Int PCI_A0_INTB#, PCI_A1_INTB# PCI_A0_INTC#, PCI_A1_INTC# PCI_A2_INTB#
4	7	Keyboard Int, Mouse Int PCI_B0_INTB#, PCI_B1_INTB# PCI_B2_INTB#, PCI_B3_INTB# PCI_A3_INTB#
3	6	SCSI Int, Ethernet Int PCI_B0_INTC#, PCI_B1_INTC# PCI_B2_INTC#, PCI_B3_INTC#
2	6	Parallel Port, Spare Int PCI_A0_INTD#, PCI_A1_INTD# PCI_A2_INTC#, PCI_A3_INTC#
1	6	PCI_B0_INTD#, PCI_B1_INTD# PCI_B2_INTD#, PCI_B3_INTD# PCI_A2_INTD#, PCI_A3_INTD#

11.3 Details

Three registers are loaded with data on each interrupt.

For UltraSPARC IIi, the upper 53 bits of the first interrupt word as well as the last two 64 bit words are 0. The least significant 11 bits of the first word contain an interrupt number (INR) which indicates the type of interrupting event. Software uses the INR to index into a table which will typically supply the IRL, PC of the interrupt service routine, and the arguments for the routine.

Two types of interrupt lines enter the concentrator: pulse and level. The distinction between these is not visible to software but is explained for clarity.

Processing hardware treats these types of interrupts slightly differently. In the case of the level interrupt, the concentrator takes the set of asserted level interrupt lines, scans them and sends the code corresponding to that interrupt once per scan time. Hardware within the UltraSPARC IIi detects the first assertion of a code, and causes a state transition which queues an interrupt packet for the UltraSPARC IIi core.

A three state FSM transmits only one interrupt (provided it remains in the PENDING state) regardless of how many interrupt codes it receives from a source. A software write causes a transition to the IDLE state and "rearms" the FSM to accept another interrupt.

Pulse interrupts are scanned and delivered to UltraSPARC IIi in a similar fashion; however, only one code is given *per pulse*. The distinction is subtle, but very important.

In the case of the existing interrupts, multiple interrupt sources can contribute to the physical line signalling the interrupt, but there is no restriction which guarantees that software knows that the interrupt line has properly deasserted.

In the case of pulse interrupts, this is required. There must be the equivalent of the pending register in the device sourcing the interrupt. Writing to this register guarantees that the interrupt line has been deasserted and therefore pulsed. As a consequence, the state machine in the UltraSPARC IIi that corresponds to a pulse interrupt has only two states.

Refer to *Interrupt States* on page 115 for a discussion of the state transitions.

11.4 Interrupt Initialization

All fields in all mapping registers listed above reset to 0. When the valid bit is cleared, no interrupts are generated from that interrupt group.

Prior to receiving the first interrupt, software must program all mapping registers to set INR.

Hardware guarantees that any transaction not in progress when the valid bit is disabled does not proceed. Once the valid bit is enabled again, interrupts proceed.

Note the valid bit *only* gates delivery of interrupts to the processor. It does not affect other state transitions within the interrupt logic. An interrupt can be delivered immediately upon first setting the valid bit if an interrupt condition exists.

11.5 Interrupt Servicing

Upon receipt of an interrupt, and assuming that PSTATE.IE=1, the UltraSPARC IIi core will take a type 0x60 trap. The INR is used to index into a table which provides three pieces of information: the IRL, the PC for the interrupt service routine, and the arguments that need to be supplied. A SOFINT trap is issued to call the interrupt service enqueue routine with this information.

When the interrupt service routine has performed all device level servicing, it calls an operating system service to dequeue it. This OS service *must* write the clear interrupt register for the appropriate interrupt source in order to re-enable interrupts from that source. Information in the appropriate clear interrupt register should be saved at the time of enqueue.

Note – The UltraSPARC IIi core uses PSTATE.IE to enable the generation of trap for IRL[4:0]. Software should not disable PSTATE.IE for a long period of time when servicing IRL[4:0].

11.6 Interrupt Sources

Interrupts in UltraSPARC IIi systems come from I/O devices, system error conditions, and software. Examples of sources of I/O device interrupts are PCI slots and the graphics interface. All I/O device interrupts are connected to the Interrupt Concentrator (the RIC IC). The Interrupt Concentrator scans through its inputs and encodes the interrupt into 6-bits for UltraSPARC IIi. UltraSPARC IIi maintains state information on all of the interrupt sources and sends an interrupt packet to the proper processor.

A unique interrupt number can be assigned to each interrupt signal line connected to the Interrupt Concentrator. The interrupt number allows the software to identify the interrupt source without polling devices. Excepting the serial ports and the keyboard and mouse, system devices do not share interrupts.

There are no outgoing interrupts from the processor.

11.6.1 PCI Interrupts

The 24 (6 slot) interrupts of prior PCI-based UltraSPARC systems are supported. eight interrupts for two more slots are also supported, although RIC does not support all the INT_NUM[4:0] encodings that are specified.

11.6.2 On-board Device Interrupts

Additional interrupts are available for use by non-PCI devices or integrated I/O devices with more interrupt requests.

11.6.3 Graphic Interrupt

During the vertical blanking period, the UPA64S device can generate an interrupt that is fed to the interrupt concentrator. Masking and clearing the UPA64S interrupt is done through the UPA64S ASIC register.

11.6.4 Error Interrupts

Internal errors detected by the PCI logic in UltraSPARC IIi are generally reported through interrupts. Error related information is recorded in UltraSPARC IIi internal registers. Refer to Chapter 16, *Error Handling* for details.

Since the Advanced PCI Bridge (APB) can delay the completion of writes, it may cause a late error report that it cannot complete the write on the secondary PCI busses. APB logs status associated with this error, and signals an error (SERR) to UltraSPARC IIi, which causes an interrupt.

11.6.5 Software Interrupts

The processor can send an interrupt to itself by setting bits in the UltraSPARC IIi SOFTINT Register.

11.7 Interrupt Concentrator

The Interrupt Concentrator logic is implemented in a Reset/interrupt/Clock Controller (RIC) chip, part number STP2210QFP, to encode interrupts from various sources into a 6-bit code that UltraSPARC IIi IO uses to identify the interrupt source. The code assignment is transparent to the software. See *Table 11-4*.

Note – A value of all ones in INT_NUM indicates the idle condition.

The Interrupt Concentrator scans the interrupt inputs in fixed order. If there is no active interrupt, the IDLE code is sent to UltraSPARC IIi. When it detects an active interrupt, the Interrupt Concentrator changes the code from IDLE to one of the active codes. It can deliver one interrupt code to UltraSPARC IIi every PCI clock cycle with an initial latency of three clock cycles.

If multiple interrupts are active at the same time, the interrupts behind the current one observe the latency due to the Interrupt Concentrator. The worst case latency introduced by the Interrupt Concentrator is 50 PCI clock periods. This figure only describes the latency from the assertion of an interrupt line to the receipt of the interrupt code in the UltraSPARC IIi.

The Interrupt Concentrator does not keep track of any state for level interrupts. For pulse interrupts, it tracks the assertion of the interrupt, and transmits only one code for each assertion. Filter logic within the chip inhibits sending additional codes to UltraSPARC IIi until the interrupt signal is deasserted. *Table 11-2* lists the edge-sensitive interrupts.

Table 11-2 INT Code Assignments for Edge-sensitive Interrupts

INT Code	Interrupt Source
0x23	Graphics Interrupt
0x26	Spare edge sensitive interrupt

Level interrupt codes are sent to the UltraSPARC IIi whenever there is a currently active interrupt. The UltraSPARC IIi must ignore incoming interrupt code when an interrupt has been detected.

11.8 UltraSPARC IIi Interrupt Handling

11.8.1 Interrupt States

Interrupts generated by I/O devices are of level or pulse type and are converted into UPA interrupt packets. UltraSPARC IIi must track of the state of each level interrupt to avoid reacting to an interrupt that the processor already received.

The three FSM states are IDLE, XMIT, and PEND. Pulse interrupts only use IDLE and XMIT.

Table 11-3 Interrupt State Transition Table

State Transition	Description
IDLE -> XMIT	An active interrupt is detected from Interrupt Concentrator.
XMIT -> PEND	The interrupt has been delivered to the processor. This transition is present only for the three state version.
XMIT -> IDLE	The interrupt has been delivered to the processor. This transition is present only for the two state version.
PEND -> IDLE	The interrupt has been cleared by software.

Note – The PEND state is to indicate that the interrupt was already sent to the UltraSPARC IIi core and is not yet cleared. For the state machine to transition to this state, the valid bit in the mapping register must be set. Interrupts for which the valid bit is not set can transition to the XMIT state, but may not dispatch to the UltraSPARC IIi core.

The interrupt state information can be obtained from Interrupt State Registers in UltraSPARC IIi. Two bits in each register define the state of a interrupt. Please refer to Section 19.3.3, *Interrupt Registers* on page 300 for a description of the registers.

11.8.2 Interrupt Prioritizing

If there are multiple interrupts in the XMIT state, their dispatch is based on a fixed priority. Between interrupts of the same priority, round-robin priority arbitration is applied.

11.8.3 Interrupt Dispatching

UltraSPARC IIi maintains an interrupt number lookup table as shown in *Table 11-4*. The Interrupt Vector Data Registers in UltraSPARC IIi are used to store the INR created from this lookup.

After an Interrupt Vector Data Register is loaded with data, the UltraSPARC IIi core must not receive another interrupt until it empties the register. Loading interrupt data into an Interrupt Vector Data Register sets the Interrupt Vector Receive Register "Busy" bit. This bit indicates to the UltraSPARC IIi IO that it must neither send another interrupt to the UltraSPARC IIi core, nor load an Interrupt Vector Data Register until this bit is cleared. The "Busy" bit can also be cleared by software.

After the UltraSPARC IIi core receives the interrupt, an interrupt trap is generated if IE bit of PSTATE Register is set to 1. The trap type for the interrupt trap is 0x60.

Table 11-4 Summary of Interrupts

RIC pin	Interrupt	Int/Ext	Source	INT_NUM (from RIC)	Type	Offset	Priority
SB0_INTREQ7	PCI A Slot 0, INTA#	Ext	PCI	0x07	Level	0x00	7
SB0_INTREQ5	PCI A Slot 0, INTB#	Ext	PCI	0x05	Level	0x01	5
SB2_INTREQ5	PCI A Slot 0, INTC#	Ext	PCI	0x15	Level	0x02	5
SB0_INTREQ2	PCI A Slot 0, INTD#	Ext	PCI	0x02	Level	0x03	2
SB1_INTREQ7	PCI A Slot 1, INTA#	Ext	PCI	0x0F	Level	0x04	7
SB1_INTREQ5	PCI A Slot 1, INTB#	Ext	PCI	0x0D	Level	0x05	5
SB3_INTREQ5	PCI A Slot 1, INTC#	Ext	PCI	0x1D	Level	0x06	5
SB1_INTREQ2	PCI A Slot 1, INTD#	Ext	PCI	0x0A	Level	0x07	2
SB2_INTREQ7	PCI A Slot 2, INTA#	Ext	PCI	0x17	Level	0x08	6
(no RIC support)	PCI A Slot 2, INTB#	Ext	PCI	0x38	Level	0x09	5
(no RIC support)	PCI A Slot 2, INTC#	Ext	PCI	0x10	Level	0x0A	2
SB2_INTREQ2	PCI A Slot 2, INTD#	Ext	PCI	0x12	Level	0x0B	1
(no RIC support)	PCI A Slot 3, INTA#	Ext	PCI	0x18	Level	0x0C	6
(no RIC support)	PCI A Slot 3, INTB#	Ext	PCI	0x39	Level	0x0D	4
(no RIC support)	PCI A Slot 3, INTC#	Ext	PCI	0x00	Level	0x0E	2
SB3_INTREQ2	PCI A Slot 3, INTD#	Ext	PCI	0x1A	Level	0x0F	1
SB0_INTREQ6	PCI B Slot 0, INTA#	Ext	PCI	0x06	Level	0x10	6
SB0_INTREQ4	PCI B Slot 0, INTB#	Ext	PCI	0x04	Level	0x11	4
SB0_INTREQ3	PCI B Slot 0, INTC#	Ext	PCI	0x03	Level	0x12	3
SB0_INTREQ1	PCI B Slot 0, INTD#	Ext	PCI	0x01	Level	0x13	1
SB1_INTREQ6	PCI B Slot 1, INTA#	Ext	PCI	0x0E	Level	0x14	6
SB1_INTREQ4	PCI B Slot 1, INTB#	Ext	PCI	0x0C	Level	0x15	4

Table 11-4 Summary of Interrupts *(Continued)*

SB1_INTREQ3	PCI B Slot 1, INTC#	Ext	PCI	0x0B	Level	0x16	3
SB1_INTREQ1	PCI B Slot 1, INTD#	Ext	PCI	0x09	Level	0x17	1
SB2_INTREQ6	PCI B Slot 2, INTA#	Ext	PCI	0x16	Level	0x18	6
SB2_INTREQ4	PCI B Slot 2, INTB#	Ext	PCI	0x14	Level	0x19	4
SB2_INTREQ3	PCI B Slot 2, INTC#	Ext	PCI	0x13	Level	0x1A	3
SB2_INTREQ1	PCI B Slot 2, INTD#	Ext	PCI	0x11	Level	0x1B	1
SB3_INTREQ6	PCI B Slot 3, INTA#	Ext	PCI	0x1E	Level	0x1C	6
SB3_INTREQ4	PCI B Slot 3, INTB#	Ext	PCI	0x1C	Level	0x1D	4
SB3_INTREQ3	PCI B Slot 3, INTC#	Ext	PCI	0x1B	Level	0x1E	3
SB3_INTREQ1	PCI B Slot 3, INTD#	Ext	PCI	0x19	Level	0x1F	1
SCSI_INT	SCSI	Ext	OBIO	0x20	Level	0x20	3
ETHERNET_INT	Ethernet	Ext	OBIO	0x21	Level	0x21	3
PARALLEL_INT	Parallel Port	Ext	OBIO	0x22	Level	0x22	2
AUDIO_INT	Audio Record	Ext	OBIO	0x24	Level	0x23	8
SB3_INTREQ7	Audio Playback	Ext	OBIO	0x1F	Level	0x24	7
POWER_FAIL_INT	Power Fail	Ext	OBIO	0x25	Level	0x25	8
KEYBOARD_INT	Kbd/Mouse/Serial	Ext	OBIO	0x28	Level	0x26	7
FLOPPY_INT	Floppy	Ext	OBIO	0x29	Level	0x27	8
SPARE_INT	Spare Hardware	Ext	OBIO	0x2A	Level	0x28	2
SKEY_INT	Keyboard	Ext	OBIO	0x2B	Level	0x29	4
SMOU_INT	Mouse	Ext	OBIO	0x2C	Level	0x2A	4
SSER_INT	Serial	Ext	OBIO	0x2D	Level	0x2B	7
	reserved					0x2C-2D	
	Uncorrectable ECC	Int	ECC		Level	0x2E	8
	Correctable ECC	Int	ECC		Level	0x2F	8
	PCI Bus Error	Int	PBM		Level	0x30	8
	reserved	Int				0x31-32	
GRAPHIC1_INT	Graphics	Ext	UPA64S	0x23	Pulse	From INR	5
GRAPHIC2_INT	Graphics	Ext	UPA64S	0x26	Pulse	From INR	5
	No interrupt	Ext	None	0x3F	N/A	N/A	N/A

11.9 Interrupt Global Registers

To expedite interrupt processing, a separate set of global registers is implemented in UltraSPARC IIi. As described in Section 11.10.5, *Interrupt Vector Receive* on page 120, the processor takes an implementation-dependent *interrupt_vector* trap after receiving an interrupt packet. Software uses a number of scratch registers while determining the appropriate handler and constructing the interrupt state.

UltraSPARC IIi provides a separate set of eight Interrupt Global Registers (IG) that replace the eight programmer-visible global registers during interrupt processing. When an *interrupt_vector* trap is taken, the hardware selects the interrupt global registers by setting the PSTATE.IG field. The PSTATE extension is described in Section 14.5.9, *PSTATE Extensions: Trap Globals* on page 193. The previous value of PSTATE is restored from the trap stack by a DONE or RETRY instruction on exit from the interrupt handler.

11.10 Interrupt ASI Registers

Note – MEMBAR `#Sync` is generally needed after stores to interrupt ASI registers.

Caution – Using ASI 0x76/77/7E/7F with VA[40:39]==00 and a VA[15:0] matching any of the PA[15:0] listed for the CSR addresses in noncacheable space, other than 0x00, 0x18, 0x20, 0x38, 0x40, 0x50, 0x60, or 0x70, can cause a load to return data, and a store to modify, the corresponding CSR. The list of addresses is in the *DMA Error Registers* on page 316.

11.10.1 Outgoing Interrupt Vector Data<2:0>

Name: Outgoing Interrupt Vector Data Registers (Privileged)

ASI_SDB_INTR_W (data 0): ASI== 0x77, VA<63:0>==0x40

ASI_SDB_INTR_W (data 1): ASI== 0x77, VA<63:0>==0x50

ASI_SDB_INTR_W (data 2): ASI== 0x77, VA<63:0>==0x60

Table 11-5 Outgoing Interrupt Vector Data Register Format

Bits	Field	Use	RW
<63:0>	Data	Data	W

Data: Interrupt data

Compatibility Note – UltraSPARC IIi does not send interrupts to any devices. A write to these registers has no effect.

Non-privileged access to this register causes a *privileged_action* trap.

11.10.2 Interrupt Vector Dispatch

Name: ASI_SDB_INTR_W (interrupt dispatch) (Privileged, write-only)

ASI: 0x77, VA<63:19>==0, VA<18:14>== target MID, VA<13:0>==0x70

UltraSPARC IIi does not send interrupts to any devices. A write to this register has no effect.

A read from this ASI causes n *data_access_exception* trap.

Non-privileged access to this register causes a *privileged_action* trap.

11.10.3 Interrupt Vector Dispatch Status Register

Name: ASI_INTR_DISPATCH_STATUS (Privileged, read-only)

ASI: 0x48, VA<63:0>==0

Table 11-6 Interrupt Dispatch Status Register Format

Bits	Field	Use	RW
<63:2>	Reserved	—	R
<1>	NACK	Always 0.	R
<0>	BUSY	Always 0.	R

NACK: Cleared at the start of every interrupt dispatch attempt; set when a dispatch has failed.

BUSY: Set if there is an outstanding dispatch.

Compatibility Note – UltraSPARC IIi does not send interrupts to any devices. A read of this register always returns zeros.

Writes to this ASI cause a *data_access_exception* trap.

Non-privileged access to this register causes a *privileged_action* trap.

11.10.4 Incoming Interrupt Vector Data<2:0>

Name: Incoming Interrupt Vector Data Registers (Privileged)

ASI_SDB_INTR_R (data 0): ASI== 0x7F, VA<63:0>==0x40

ASI_SDB_INTR_R (data 1): ASI== 0x7F, VA<63:0>==0x50

ASI_SDB_INTR_R (data 2): ASI== 0x7F, VA<63:0>==0x60

Table 11-7 Incoming Interrupt Vector Data Register Format

Bits	Field	Use	RW
<63:0>	Data	Data	R

Data: Interrupt data

Compatibility Note – UltraSPARC IIi only supports the interrupt data that were present in prior UltraSPARC-based systems; that is, bits 10:0 (INR) of ASI_SDB_INTR(0). All other bits are read as 0.

Non-privileged access to this register causes a *privileged_action* trap

11.10.5 Interrupt Vector Receive

Name: ASI_INTR_RECEIVE (Privileged)

ASI: 0x49, VA<63:0>==0

Table 11-8 Interrupt Vector Receive Register Format

Bits	Field	Use	RW
<63:6>	Reserved	—	R
<5>	BUSY	Set when an interrupt vector is received	RW
<4:0>	MID<4:0>	Always 0	R

BUSY: This bit is set when an interrupt vector is received.

MID<4:0>: Module ID of interrupter. Always 0 on UltraSPARC IIi.

Note – The BUSY bit must be cleared by software writing zero.

The status of an incoming interrupt can be read from ASI_INTR_RECEIVE. The BUSY bit is cleared by writing a zero to this register.

Non-privileged access to this register causes a *privileged_action* trap.

11.11 Software Interrupt (SOFTINT) Register

In order to schedule interrupt vectors for later processing, each processor can send signals to itself by setting bits in the SOFTINT Register.

Table 11-9 SOFTINT Register Format

Bits	**Field**	**Use**	**RW**
<15:1>	SOFTINT<15:1>	When set, bits<15:1> cause interrupts at levels IRL<15:1> respectively.	RW
<0>	TICK_INT	Timer interrupt	RW

SOFTINT: When set, bits<15:1> cause interrupts at levels IRL<15:1> respectively.

TICK_INT: When the TICK_CMPR's INT_DIS field is cleared (that is, the TICK interrupt is enabled) and the 63-bit TICK_Compare Register's TICK_CMPR field matches the TICK Register's counter field, the TICK_INT field is set and a software interrupt is generated. See also Section 14.1.8, *TICK Register* on page 179 and Section 14.5.1, *Per-Processor TICK Compare Field of TICK Register* on page 191.

The SOFTINT register (ASR 16_{16}) is used for communication from (TL > 0) Nucleus code to (T=0) kernel code. Non privileged accesses to this register cause a *privileged_opcode* trap. Interrupt packets and other service requests can be scheduled in queues or mailboxes in memory by the nucleus, which then sets SOFTINT<*n*> to cause an interrupt at level <*n*>. Setting SOFTINT<*n*> is done via a write to the SET_SOFTINT register (ASR 14_{16}) with bit <*n*> corresponding to the interrupt level set. Note that the value written to the SET_SOFTINT register is effectively ORd into the SOFTINT register. This action allows the interrupt handler to set one or more bits in the SOFTINT register with a single instruction. Read accesses to the SET_SOFTINT register cause an *illegal_instruction* trap. Non-privileged accesses to this register cause a *privileged_opcode* trap. When the nucleus returns, if (PSTATE.IE=1) and (PIL < *n*), the processor receives the highest priority interrupt IRL<*n*> of the asserted bits in SOFTINT<15:0>.

The processor then takes a trap for the interrupt request, the nucleus sets the return state to the interrupt handler at that PIL, and returns to TL0. In this manner, the nucleus can schedule services at various priorities and process them according to their priority.

When all interrupts scheduled for service at level *n* have been serviced, the kernel writes to the CLEAR_SOFTINT register (ASR 15_{16}) with bit *n* set, to clear that interrupt. Note that the complement of the value written to the CLEAR_SOFTINT register is effectively ANDd with the SOFTINT register. This allows the interrupt handler to clear one or more bits in the SOFTINT register with a single instruction. Read accesses to the CLEAR_SOFTINT register cause an *illegal_instruction* trap. Non privileged write accesses to this register cause a *privileged_opcode* trap.

The timer interrupt TICK_INT is equivalent to SOFTINT<14> and has the same effect.

Note – To avoid a race condition between the kernel clearing an interrupt and the nucleus setting it, the kernel should reexamine the queue for any valid entries after clearing the interrupt bit.

Table 11-10 SOFTINT ASRs

ASR Value	ASR Name/Syntax	Access	Description
14_{16}	SET_SOFTINT	W	Set bit(s) in Soft Interrupt register
15_{16}	CLEAR_SOFTINT	W	Clear bit(s) in Soft Interrupt register
16_{16}	SOFTINT_REG	RW	Per-processor Soft Interrupt register

CHAPTER **12**

Instruction Set Summary

The UltraSPARC IIi CPU implements both the standard SPARC-V9 instruction set and a number of implementation-dependent extended instructions. Standard SPARC-V9 instructions are documented in *The SPARC Architecture Manual, Version 9*. UltraSPARC IIi extended instructions are documented in Chapter 13, *VIS™ and Additional Instructions*."

Table 12-1 lists the complete UltraSPARC IIi instruction set. A check (✓) in the "Ext" column indicates that the instruction is an UltraSPARC IIi extension; the absence of a check indicates a SPARC-V9 core instruction. The "Ref" column lists the section number that contains the instruction documentation. SPARC-V9 core instructions are documented in *The SPARC Architecture Manual, Version 9*; UltraSPARC IIi extensions are documented in this manual.

Note – The first printing of *The SPARC Architecture Manual, Version 9* contains two sections numbered A.31; the subsequent sections in Appendix A are misnumbered. For convenience, *Table 12-1* on page 123 of this manual follows this incorrect numbering scheme. When *The SPARC Architecture Manual, Version 9* is corrected, *Table 12-1* will be changed to match the correct numbering.

Table 12-1 Complete UltraSPARC IIi Instruction Set

Opcode	Description	Ext	Ref
ADD (ADDcc)	Add (and modify condition codes)		V9, App.A[1]
ADDC (ADDCcc)	Add with carry (and modify condition codes)		V9, App.A
ALIGNADDRESS	Calculate address for misaligned data access	✓	Section 13.4.5
ALIGNADDRESSL	Calculate address for misaligned data access (little-endian)	✓	Section 13.4.5
AND (ANDcc)	And (and modify condition codes)		V9, App.A
ANDN (ANDNcc)	And not (and modify condition codes)		V9, App.A

Table 12-1 Complete UltraSPARC IIi Instruction Set *(Continued)*

Opcode	Description	Ext	Ref
ARRAY{8,16,32}	3-D address to blocked byte address conversion	✓	Section 13.4.1 0
Bicc	Branch on integer condition codes		V9, App.A
BLD	64-byte block load	✓	Section 13.4.1 0
BPcc	Branch on integer condition codes with prediction		V9, App.A
BPr	Branch on contents of integer register with prediction		V9, App.A
BST	64-byte block store	✓	Section 13.5.3
CALL	Call and link		V9, App.A
CASA	Compare and swap word in alternate space		V9, App.A
CASXA	Compare and swap doubleword in alternate space		V9, App.A
DONE	Return from trap		V9, App.A
EDGE{8,16,32}{L}	Edge boundary processing {little-endian}	✓	Section 13.4.8
FABS(s,d,q)	Floating-point absolute value		V9, App.A
FADD(s,d,q)	Floating-point add		V9, App.A
FALIGNDATA	Perform data alignment for misaligned data	✓	Section 13.4.5
FANDNOT1{s}	Negated src1 AND src2 (single precision)	✓	Section 13.4.6
FANDNOT2{s}	src1 AND negated src2 (single precision)	✓	Section 13.4.6
FAND{s}	Logical AND (single precision)		Section 13.4.6
FBPfcc	Branch on floating-point condition codes with prediction		V9, App.A
FBfcc	Branch on floating-point condition codes		V9, App.A
FCMP(s,d,q)	Floating-point compare		V9, App.A
FCMPE(s,d,q)	Floating-point compare (exception if unordered)		V9, App.A
FCMPEQ{16,32}	Four 16-bit/two 32-bit compare; set integer dest if src1 = src2	✓	Section 13.4.7
FCMPGT{16,32}	Four 16-bit/two 32-bit compare; set integer dest if src1 > src2	✓	Section 13.4.7
FCMPLE{16,32}	Four 16-bit/two 32-bit compare; set integer dest if src1 <= src2	✓	Section 13.4.7
FCMPNE{16,32}	Four 16-bit/two 32-bit compare; set integer dest if src1 != src2	✓	Section 13.4.7
FDIV(s,d,q)	Floating-point divide		V9, App.A
FdMULq	Floating-point multiply double to quad		V9, App.A
FEXPAND	Four 8-bit to 16-bit expand	✓	Section 13.4.3
FiTO(s,d,q)	Convert integer to floating-point		V9, App.A
FLUSH	Flush instruction memory		V9, App.A

Table 12-1 Complete UltraSPARC IIi Instruction Set *(Continued)*

Opcode	Description	Ext	Ref
FLUSHW	Flush register windows		V9, App.A
FMOV(s,d,q)	Floating-point move		V9, App.A
FMOV(s,d,q)cc	Move floating-point register if condition is satisfied		V9, App.A
FMOV(s,d,q)r	Move floating-point register if integer register contents satisfy condition		V9, App.A
FMUL(s,d,q)	Floating-point multiply		V9, App.A
FMUL8SUx16	Signed upper 8- × 16-bit partitioned product of corresponding components	✓	Section 13.4.4
FMUL8ULx16	Unsigned lower 8- × 16-bit partitioned product of corresponding components	✓	Section 13.4.4
FMUL8x16	8- × 16-bit partitioned product of corresponding components	✓	Section 13.4.4
FMUL8x16AL	8- × 16-bit lower α partitioned product of 4 components	✓	Section 13.4.4
FMUL8x16AU	8- × 16-bit upper α partitioned product of 4 components	✓	Section 13.4.4
FMULD8SUx16	Signed upper 8- × 16-bit multiply → 32-bit partitioned product of components	✓	Section 13.4.4
FMULD8ULx16	Unsigned lower 8- × 16-bit multiply → 32-bit partitioned product of components	✓	Section 13.4.4
FNAND{s}	Logical NAND (single precision)	✓	Section 13.4.6
FNEG(s,d,q)	Floating-point negate	✓	Section 13.4.6
FNOR{s}	Logical NOR (single precision)	✓	Section 13.4.6
FNOT1{s}	Negate (1′s complement) src1 (single precision)	✓	Section 13.4.6
FNOT2{s}	Negate (1′s complement) src2 (single precision)	✓	Section 13.4.6
FONE{s}	One fill (single precision)	✓	Section 13.4.6
FORNOT1{s}	Negated src1 OR src2 (single precision)	✓	Section 13.4.6
FORNOT2{s}	src1 OR negated src2 (single precision)	✓	Section 13.4.6
FOR{s}	Logical OR (single precision)	✓	Section 13.4.6
FPACKFIX	Two 32-bit to 16-bit fixed pack	✓	Section 13.4.3
FPACK{16,32}	Four 16-bit/two 32-bit pixel pack	✓	Section 13.4.3
FPADD{16,32}{s}	Four 16-bit/two 32-bit partitioned add (single precision)	✓	Section 13.4.2
FPMERGE	Two 32-bit pixel to 64-bit pixel merge	✓	Section 13.4.3
FPSUB{16,32}{s}	Four 16-bit/two 32-bit partitioned subtract (single precision)	✓	Section 13.4.2
FsMULd	Floating-point multiply single to double		V9, App.A
FSQRT(s,d,q)	Floating-point square root		V9, App.A

Table 12-1 Complete UltraSPARC IIi Instruction Set *(Continued)*

Opcode	Description	Ext	Ref
FSRC1{s}	Copy src1 (single precision)	✓	Section 13.4.6
FSRC2{s}	Copy src2 (single precision)	✓	Section 13.4.6
F(s,d,q)TO(s,d,q)	Convert between floating-point formats		V9, App.A
F(s,d,q)TOi	Convert floating point to integer		V9, App.A
F(s,d,q)TOx	Convert floating point to 64-bit integer		V9, App.A
FSUB(s,d,q)	Floating-point subtract		V9, App.A
FXNOR{s}	Logical XNOR (single precision)	✓	Section 13.4.6
FXOR{s}	Logical XOR (single precision)	✓	Section 13.4.6
FxTO(s,d,q)	Convert 64-bit integer to floating-point		V9, App.A
FZERO{s}	Zero fill (single precision)	✓	Section 13.4.6
ILLTRAP	Illegal instruction		V9, App.A
IMPDEP1	Implementation-dependent instruction		V9, App.A
IMPDEP2	Implementation-dependent instruction		V9, App.A
JMPL	Jump and link		V9, App.A
LDD	Load doubleword		V9, App.A
LDDA	Load doubleword from alternate space		V9, App.A
LDDA	128-bit atomic load	✓	Section 13.6.1
LDDF	Load double floating-point		V9, App.A
LDDFA	Load double floating-point from alternate space		V9, App.A
LDDFA	Zero-extended 8-/16-bit load to a double precision FP register	✓	Section 13.5.2
LDF	Load floating-point		V9, App.A
LDFA	Load floating-point from alternate space		V9, App.A
LDFSR	Load floating-point state register lower		V9, App.A
LDQF	Load quad floating-point		V9, App.A
LDQFA	Load quad floating-point from alternate space		V9, App.A
LDSB	Load signed byte		V9, App.A
LDSBA	Load signed byte from alternate space		V9, App.A
LDSH	Load signed halfword		V9, App.A
LDSHA	Load signed halfword from alternate space		V9, App.A
LDSTUB	Load-store unsigned byte		V9, App.A
LDSTUBA	Load-store unsigned byte in alternate space		V9, App.A

Table 12-1 Complete UltraSPARC IIi Instruction Set *(Continued)*

Opcode	Description	Ext	Ref
LDSW	Load signed word		V9, App.A
LDSWA	Load signed word from alternate space		V9, App.A
LDUB	Load unsigned byte		V9, App.A
LDUBA	Load unsigned byte from alternate space		V9, App.A
LDUH	Load unsigned halfword		V9, App.A
LDUHA	Load unsigned halfword from alternate space		V9, App.A
LDUW	Load unsigned word		V9, App.A
LDUWA	Load unsigned word from alternate space		V9, App.A
LDX	Load extended		V9, App.A
LDXA	Load extended from alternate space		V9, App.A
LDXFSR	Load extended floating-point state register		V9, App.A
MEMBAR	Memory barrier		V9, App.A
MOVcc	Move integer register if condition is satisfied		V9, App.A
MOVr	Move integer register on contents of integer register		V9, App.A
MULScc	Multiply step (and modify condition codes)		V9, App.A
MULX	Multiply 64-bit integers		V9, App.A
NOP	No operation		V9, App.A
OR (ORcc)	Inclusive-or (and modify condition codes)		V9, App.A
ORN (ORNcc)	Inclusive-or not (and modify condition codes)		V9, App.A
PDIST	Distance between 8 8-bit components	✓	Section 13.4.9
POPC	Population count		V9, App.A
PREFETCH[2]	Prefetch data		V9, App.A
PREFETCHA[2]	Prefetch data from alternate space		V9, App.A
PST	Eight 8-bit/4 16-bit/2 32-bit partial stores	✓	Section 13.5.1
RDASI	Read ASI register		V9, App.A
RDASR	Read ancillary state register		V9, App.A
RDCCR	Read condition codes register		V9, App.A
RDFPRS	Read floating-point registers state register		V9, App.A
RDPC	Read program counter		V9, App.A
RDPR	Read privileged register		V9, App.A
RDTICK	Read TICK register		V9, App.A

Table 12-1 Complete UltraSPARC IIi Instruction Set *(Continued)*

Opcode	Description	Ext	Ref
RDY	Read Y register		V9, App.A
RESTORE	Restore caller's window		V9, App.A
RESTORED	Window has been restored		V9, App.A
RETRY	Return from trap and retry		V9, App.A
RETURN	Return		V9, App.A
SAVE	Save caller's window		V9, App.A
SAVED	Window has been saved		V9, App.A
SDIV (SDIVcc)	32-bit signed integer divide (and modify condition codes)		V9, App.A
SDIVX	64-bit signed integer divide		V9, App.A
SETHI	Set high 22 bits of low word of integer register		V9, App.A
SHUTDOWN	Power-down support	✓	Section 13.6.2
SIR	Software-initiated reset		V9, App.A
SLL	Shift left logical		V9, App.A
SLLX	Shift left logical, extended		V9, App.A
SMUL (SMULcc)	Signed integer multiply (and modify condition codes)		V9, App.A
SRA	Shift right arithmetic		V9, App.A
SRAX	Shift right arithmetic, extended		V9, App.A
SRL	Shift right logical		V9, App.A
SRLX	Shift right logical, extended		V9, App.A
STB	Store byte		V9, App.A
STBA	Store byte into alternate space		V9, App.A
STBAR	Store barrier		V9, App.A
STD	Store doubleword		V9, App.A
STDA	Store doubleword into alternate space		V9, App.A
STDF	Store double floating-point		V9, App.A
STDFA	Store double floating-point into alternate space		V9, App.A
STDFA	8-/16-bit store from a double precision FP register	✓	Section 13.5.2
STF	Store floating-point		V9, App.A
STFA	Store floating-point into alternate space		V9, App.A
STFSR	Store floating-point state register		V9, App.A
STH	Store halfword		V9, App.A

Table 12-1 Complete UltraSPARC IIi Instruction Set *(Continued)*

Opcode	Description	Ext	Ref
STHA	Store halfword into alternate space		V9, App.A
STQF	Store quad floating-point		V9, App.A
STQFA	Store quad floating-point into alternate space		V9, App.A
STW	Store word		V9, App.A
STWA	Store word into alternate space		V9, App.A
STX	Store extended		V9, App.A
STXA	Store extended into alternate space		V9, App.A
STXFSR	Store extended floating-point state register		V9, App.A
SUB (SUBcc)	Subtract (and modify condition codes)		V9, App.A
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)		V9, App.A
SWAP	Swap integer register with memory		V9, App.A
SWAPA	Swap integer register with memory in alternate space		V9, App.A
TADDcc (TADDccTV)	Tagged add and modify condition codes (trap on overflow)		V9, App.A
TSUBcc (TSUBccTV)	Tagged subtract and modify condition codes (trap on overflow)		V9, App.A
Tcc	Trap on integer condition codes		V9, App.A
UDIV (UDIVcc)	Unsigned integer divide (and modify condition codes)		V9, App.A
UDIVX	64-bit unsigned integer divide		V9, App.A
UMUL (UMULcc)	Unsigned integer multiply (and modify condition codes)		V9, App.A
WRASI	Write ASI register		V9, App.A
WRASR	Write ancillary state register		V9, App.A
WRCCR	Write condition codes register		V9, App.A
WRFPRS	Write floating-point registers state register		V9, App.A
WRPR	Write privileged register		V9, App.A
WRY	Write Y register		V9, App.A
XNOR (XNORcc)	Exclusive-nor (and modify condition codes)		V9, App.A
XOR (XORcc)	Exclusive-or (and modify condition codes)		V9, App.A

1. Reference is to Appendix A of *The The SPARC Architecture Manual, Version 9*.

2. UltraSPARC-I does not implement the PREFETCH and PREFETCHA instructions.

CHAPTER **13**

VIS™ and Additional Instructions

13.1 Introduction

The UltraSPARC IIi CPU extends the standard SPARC-V9 instruction set with new classes of instructions that enhance graphics functionality (see Section 13.4, *Graphics Instructions*), and improve the efficiency of memory accesses (see Section 13.5, *Memory Access Instructions*). These are collectively known as the VIS Instruction Set, or VIS.

13.2 Graphics Data Formats

Graphics instructions are optimized for short integer arithmetic, where the overhead of converting to and from floating-point is significant. Image components may be 8 or 16 bits; intermediate results are 16 or 32 bits.

13.2.1 8-Bit Format

Pixels consist of four unsigned 8-bit integers contained in a 32-bit word. Typically, they represent intensity values for an image (e.g. α, B, G, R). UltraSPARC IIi supports

- *Band interleaved* images, with the various color components of a point in the image stored together, and
- *Band sequential* images, with all of the values for one color component stored together.

13.2.2 Fixed Data Formats

The fixed 16-bit data format consists of four 16-bit signed fixed-point values contained in a 64-bit word. The fixed 32-bit format consists of two 32-bit signed fixed point-values contained in a 64-bit word. Fixed data values provide an intermediate format with enough precision and dynamic range for filtering and simple image computations on pixel values. Conversion from pixel data to fixed data occurs through pixel multiplication. Conversion from fixed data to pixel data is done with the pack instructions, which clip and truncate to an 8-bit unsigned value. Conversion from 32-bit fixed to 16-bit fixed is also supported with the FPACKFIX instruction. Rounding can be performed by adding 1 to the round bit position. Complex calculations needing more dynamic range or precision should be performed using floating-point data.

These formats are shown in *Figure 13-1*.

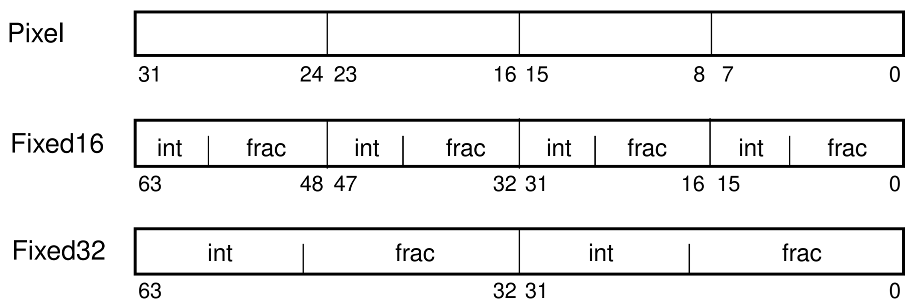


Figure 13-1 Graphics Fixed Data Formats

Note – Sun frame buffer pixel component ordering is: α, B, G, R.

13.3 Graphics Status Register (GSR)

The GSR is accessed with implementation-dependent RDASR and WRASR instructions using ASR 13_{16}

Table 13-1 Graphics Status Register Opcodes

opcode	op3	reg field	operation
RDASR	10 1000	*rs1* = 19	Read GSR
WRASR	11 0000	*rd* = 19	Write GSR

10	rd	op3	rs1	i=0	—
31 30	29 25	24 19	18 14	13	12 0

Figure 13-2 RDASR Format

10	rd	op3	rs1	i=0	—	rs2
10	rd	op3	rs1	i=1	simm13	
31 30	29 25	24 19	18 14	13	12 5	4 0

Figure 13-3 WRASR Format

Table 13-2 GSR Instruction Syntax

Suggested Assembly Language Syntax	
`rd`	`%gsr,` *reg*$_{rd}$
`wr`	*reg*$_{rs1}$, *reg_or_imm*, `%gsr`

Accesses to this register cause an *fp_disabled* trap if either PSTATE.PEF or FPRS.FEF is zero.

—	scale_factor	alignaddr_offset
63 7	6 3	2 0

Figure 13-4 GSR Format (ASR 10_{16})

scale_factor: Shift count in the range 0..15, used by PACK instructions for pixel formatting.

alignaddr_offset: Least significant three bits of the address computed by the last ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction. See Section 13.4.5, *Alignment Instructions* on page 148.

Traps

fp_disabled

13.4 Graphics Instructions

All instruction operands are in floating-point registers, unless otherwise specified. This arrangement provides the maximum number of registers (32 double-precision) and the maximum instruction parallelism (for example, UltraSPARC IIi is four scalar for floating-point/graphics code only). Pixel values are stored in single-precision floating point registers and fixed values are stored in double-precision floating-point registers, unless otherwise specified.

13.4.1 Opcode Format

The graphics instruction set maps to the opcode space reserved for the Implementation-Dependent Instruction 1 (IMPDEP1) instructions.

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-5 Graphics Instruction Format (3)

13.4.2 Partitioned Add/Subtract Instructions

Table 13-3 Partitioned Add/Subtract Instruction Opcodes

opcode	opf	operation
FPADD16	0 0101 0000	Four 16-bit add
FPADD16 S	0 0101 0001	Two 16-bit add
FPADD32	0 0101 0010	Two 32-bit add
FPADD32S	0 0101 0011	One 32-bit add
FPSUB16	0 0101 0100	Four 16-bit subtract

Table 13-3 Partitioned Add/Subtract Instruction Opcodes *(Continued)*

opcode	opf	operation
FPSUB16S	0 0101 0101	Two 16-bit subtract
FPSUB32	0 0101 0110	Two 32-bit subtract
FPSUB32S	0 0101 0111	One 32-bit subtract

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-6 Partitioned Add/Subtract Instruction Format (3)

Table 13-4 Partitioned Add/Subtract Instruction Syntax

Suggested Assembly Language Syntax	
fpadd16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpadd16s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpadd32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpadd32s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub16s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpsub32s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

The standard versions of these instructions perform four 16-bit or two 32-bit partitioned adds or subtracts between the corresponding fixed point values contained in the source operands (*rs1*, *rs2*). For subtraction, *rs2* is subtracted from *rs1*. The result is placed in the destination register (*rd*).

The single precision version of these instructions (FPADD16S, FPSUB16S, FPADD32S, FPSUB32S) perform two (16-bit) or one (32-bit) partitioned adds or subtracts.

Note – For good performance, do not use the result of a single FPADD as part of a 64-bit graphics instruction source operand in the next instruction group. Similarly, do not use the result of a standard FPADD as a 32-bit graphics instruction source operand in the next instruction group.

Traps

fp_disabled

13.4.3 Pixel Formatting Instructions

Table 13-5 Pixel Formatting Instruction Opcode Format

opcode	opf	operation
FPACK16	0 0011 1011	Four 16-bit packs
FPACK32	0 0011 1010	Two 32-bit packs
FPACKFIX	0 0011 1101	Four 16-bit packs
FEXPAND	0 0100 1101	Four 16-bit expands
FPMERGE	0 0100 1011	Two 32-bit merges

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-7 Pixel Formatting Instruction Format (3)

Table 13-6 Pixel Formatting Instruction Syntax

Suggested Assembly Language Syntax	
fpack16	$freg_{rs2}$, $freg_{rd}$
fpack32	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fpackfix	$freg_{rs2}$, $freg_{rd}$
fexpand	$freg_{rs2}$, $freg_{rd}$
fpmerge	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

The PACK instructions convert to a lower precision fixed or pixel format. Input values are clipped to the dynamic range of the output format. Packing applies a scale factor from GSR.*scale_factor* to allow flexible positioning of the binary point.

Note – For good performance, do not use the result of an FPACK as part of a 64-bit graphics instruction source operand in the next three instruction groups. Do not use the result of FEXPAND or FPMERGE as a 32-bit graphics instruction source operand in the next three instruction groups.

Traps

fp_disabled

13.4.3.1 FPACK16

FPACK16 takes four 16-bit fixed values in *rs2*, scales, truncates and clips them into four 8-bit unsigned integers and stores the results in the 32-bit *rd* register.

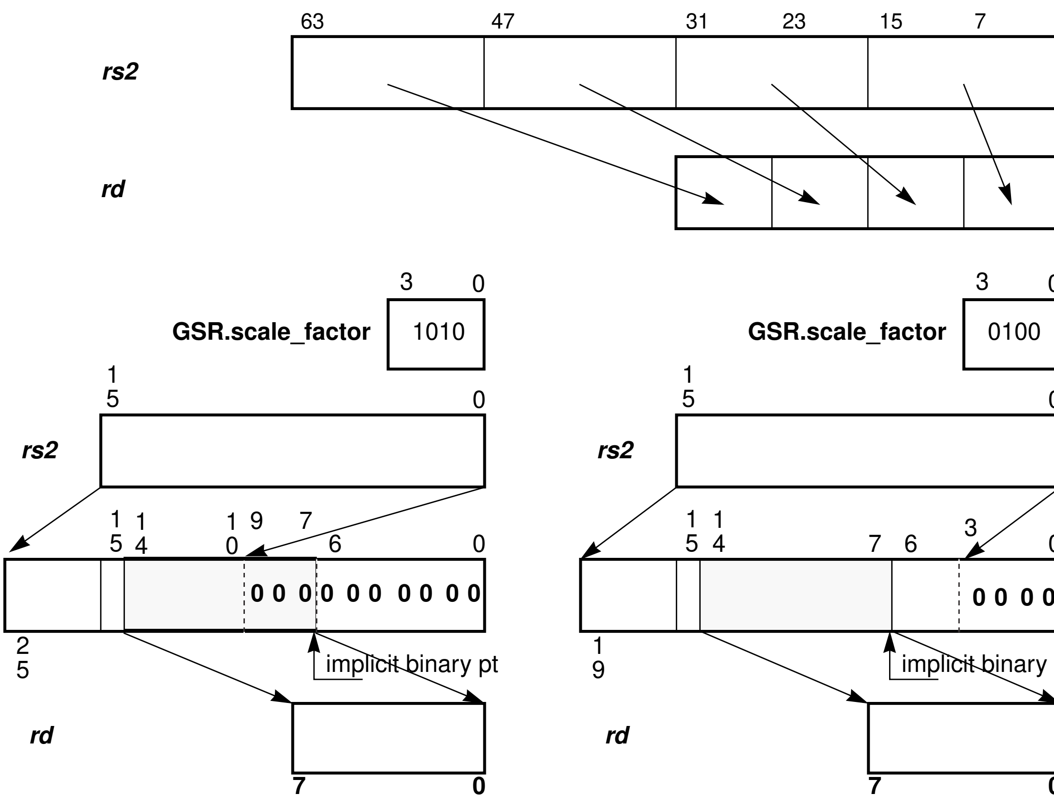


Figure 13-8 FPACK16 Operation

This operation, illustrated in *Figure 13-8*, is carried out as follows:

1. Left shift the value in *rs2* by the number of bits in the GSR.*scale_factor*, while maintaining clipping information.

2. Truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (i.e. between bits 7 and 6 for each 16-bit word). Truncation is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, the MSB is set), zero is delivered as the clipped value. If the value is greater than 255, then 255 is delivered. Otherwise the scaled value is the final result.

3. Store the result in the corresponding byte in the 32-bit *rd* register.

13.4.3.2 FPACK32

FPACK32 takes two 32-bit fixed values in *rs2*, scales, truncates and clips them into two 8-bit unsigned integers. The two 8-bit integers are merged at the corresponding least significant byte positions of each 32-bit word in *rs1* left shifted by 8 bits. The 64-bit result is stored in the *rd* register. This allows two pixels to be assembled by successive FPACK32 instructions using three or four pairs of 32-bit fixed values.

This operation, illustrated in *Figure 13-9*, is carried out as follows:

1. Left shift each 32-bit value in *rs2* by the number of bits in the GSR.*scale_factor*, while maintaining clipping information.

2. For each 32-bit value, truncate and clip to an 8-bit unsigned integer starting at the bit immediately to the left of the implicit binary point (i.e. between bits 23 and 22 of each 32-bit word). Truncation is performed to convert the scaled value into a signed integer (that is, round toward negative infinity). If the resulting value is negative (that is, the MSB is set), zero is delivered as the clipped value. If the value is greater than 255, then 255 is delivered. Otherwise the scaled value is the final result.

3. Left shift each 32-bit values in *rs1* by 8 bits.

4. Merge the two clipped 8-bit unsigned values into the corresponding least significant byte positions in the left-shifted *rs2* value.

5. Store the result in the *rd* register.

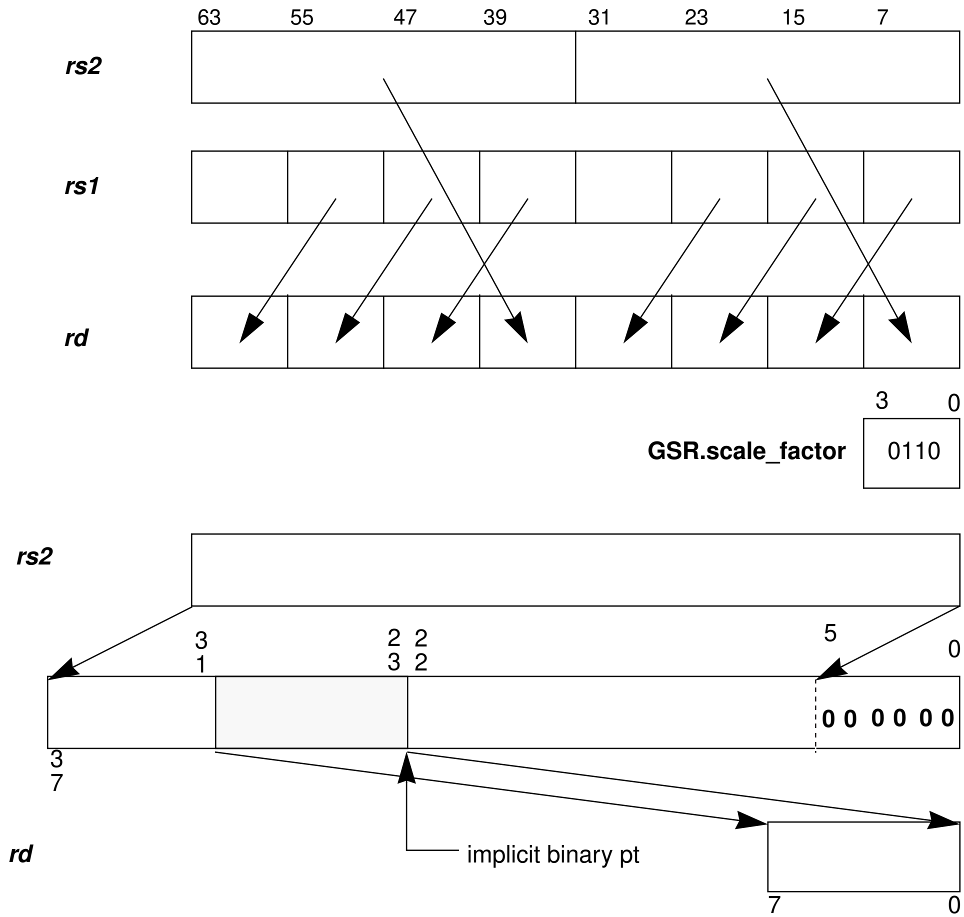


Figure 13-9 FPACK32 Operation

13.4.3.3 FPACKFIX

FPACKFIX takes two 32-bit fixed values in *rs2*, scales, truncates and clips them into two 16-bit signed integers, then stores the result in the 32-bit *rd* register.

This operation, illustrated in *Figure 13-10*, is carried out as follows:

1. Left shift each 32-bit value in *rs2* by the number of bits in the GSR.*scale_factor*, while maintaining clipping information.
2. For each 32-bit value, truncate and clip to a 16-bit signed integer starting at the bit immediately to the left of the implicit binary point (i.e. between bits 16 and 15 of each 32-bit word). Truncation is performed to convert the scaled value into a

signed integer (i.e. rounds toward negative infinity). If the resulting value is less than -32768, -32768 is delivered as the clipped value. If the value is greater than 32767, 32767 is delivered. Otherwise the scaled value is the final result.

3. Store the result in the 32-bit *rd* register.

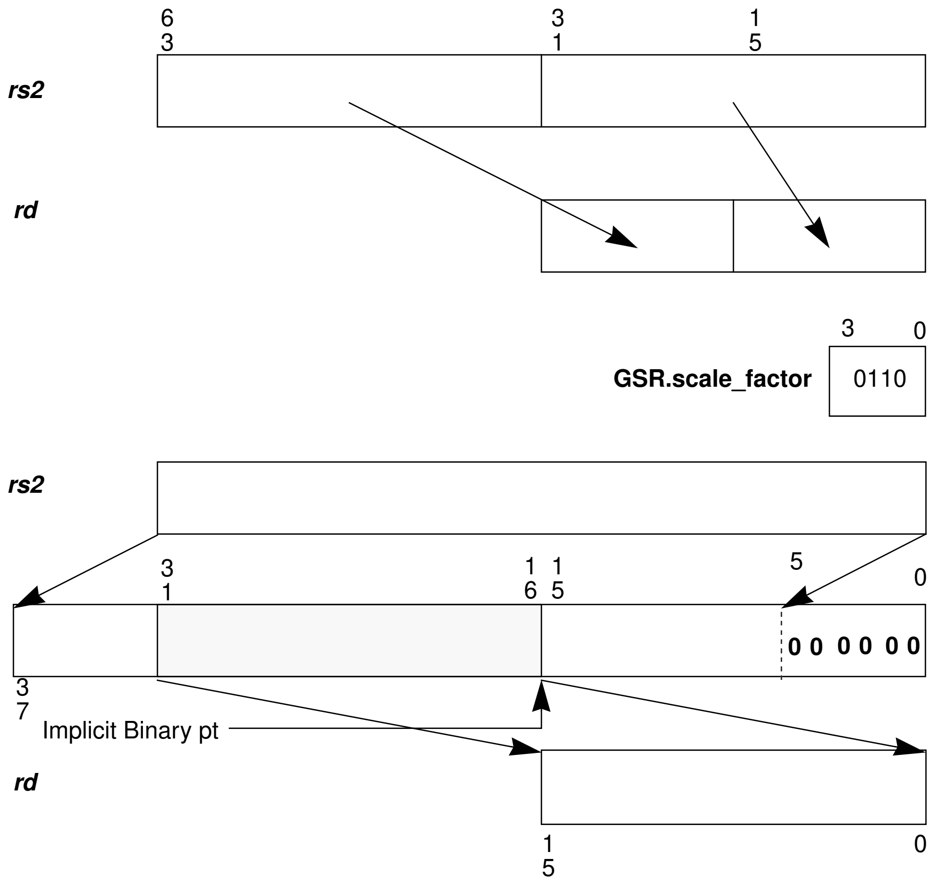


Figure 13-10 FPACKFIX Operation

13.4.3.4 FEXPAND

FEXPAND takes four 8-bit unsigned integers in *rs2*, converts each integer to a 16-bit fixed value, and stores the four 16-bit results in the *rd* register.

This operation, illustrated in *Figure 13-11*, is carried out as follows:

1. Left shift each 8-bit value by 4 and zero-extend the results to a 16-bit fixed value.

2. Stores the results in the *rd* register.

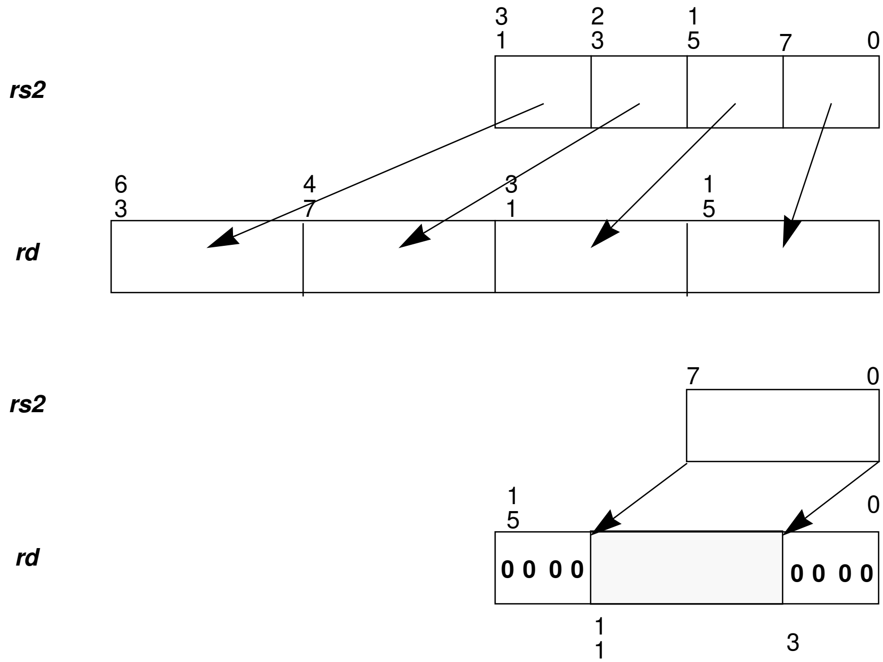


Figure 13-11 FEXPAND Operation

13.4.3.5 FPMERGE

FPMERGE interleaves four corresponding 8-bit unsigned values in *rs1* and *rs2*, to produce a 64-bit value in the *rd* register. This instruction converts from packed to planar representation when it is applied twice in succession; for example:

R1G1B1A1, R3G3B3A3 → R1R3G1G3B1B3 → R1R2R3R4B1B2B3B4

FPMERGE also converts from planar to packed when it is applied twice in succession; for example:

R1R2R3R4, B1B2B3B4 → R1B1R2B2R3B3R4B4 → R1G1B1A1R2G2B2A2

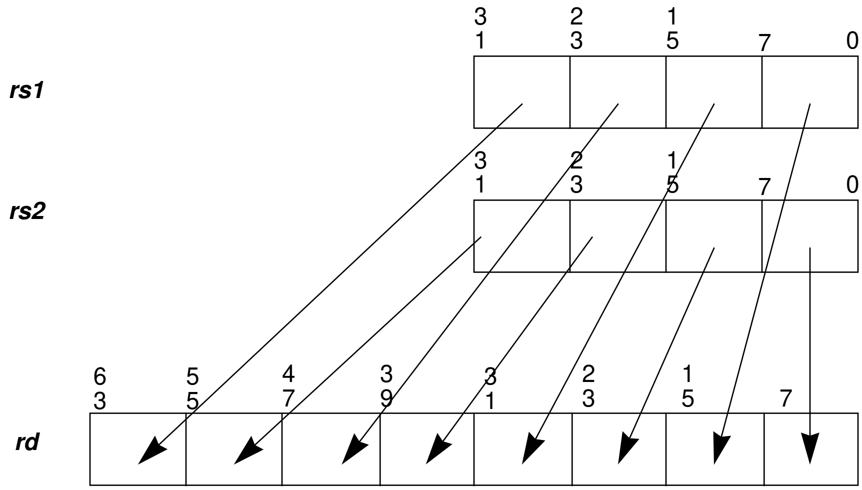


Figure 13-12 FPMERGE Operation

13.4.4 Partitioned Multiply Instructions

Table 13-7 Partitioned Multiply Instruction Opcodes

opcode	opf	operation
FMUL8x16	0 0011 0001	8- × 16-bit partitioned product
FMUL8x16AU	0 0011 0011	8- × 16-bit upper α partitioned product
FMUL8x16AL	0 0011 0101	8- × 16-bit lower α partitioned product
FMUL8SUx16	0 0011 0110	upper 8- × 16-bit partitioned product
FMUL8ULx16	0 0011 0111	lower unsigned 8- × 16-bit partitioned product
FMULD8SUx16	0 0011 1000	upper 8- × 16-bit partitioned product
FMULD8ULx16	0 0011 1001	lower unsigned 8- × 16-bit partitioned product

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-13 Partitioned Multiply Instruction Format (3)

Table 13-8 Partitioned Multiply Instruction Syntax

Suggested Assembly Language Syntax	
fmul8x16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8x16au	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8x16al	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8sux16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmul8ulx16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmuld8sux16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fmuld8ulx16	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

The following sections describe the variations of partitioned multiply.

Note – For good performance, do not use the result of a partitioned multiply as a 32-bit graphics instruction source operand in the next three instruction groups.

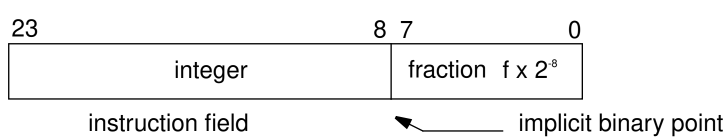
Traps

fp_disabled

Note – When software emulates an 8-bit unsigned by 16-bit signed multiply, the unsigned value must be zero-extended and the 16-bit value must be sign-extended before the multiplication.

13.4.4.1 FMUL8x16

FMUL8x16 multiplies each unsigned 8-bit value (i.e., a pixel) in *rs1* by the corresponding (signed) 16-bit fixed-point integers in *rs2*; it rounds the 24-bit product (assuming a binary point between bits 7 and 8) and stores the upper 16 bits of the result into the corresponding 16-bit field in the *rd* register. *Figure 13-14* illustrates the operation.



Note – This instruction treats the pixel values as fixed-point with the binary point to the left of the most significant bit. Typically, this operation is used with filter coefficients as the fixed-point *rs2* value and image data as the *rs1* pixel value. Appropriate scaling of the coefficient allows various fixed-point scaling to be realized.

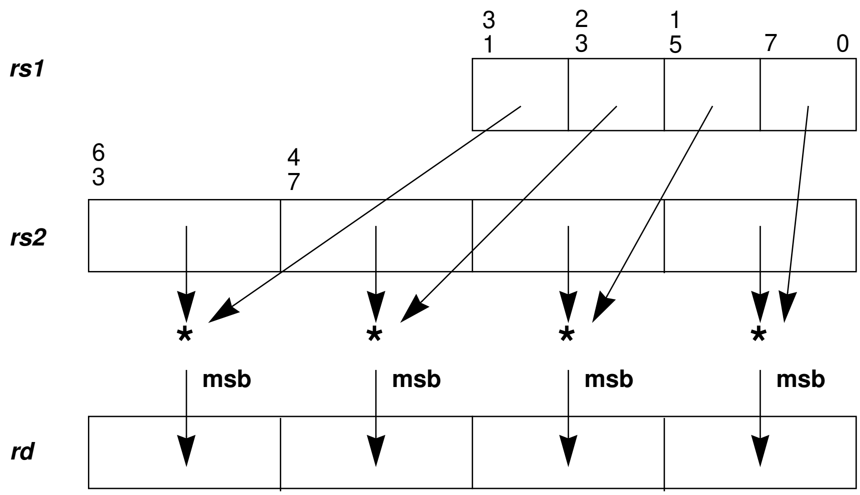


Figure 13-14 FMUL8x16 Operation

13.4.4.2 FMUL8x16AU

FMUL8x16AU is similar to FMUL8x16, except that one 16-bit fixed-point value is used for all four multiplies. This value is the most significant 16 bits of the 32-bit *rs2* register, which is typically an α value. The operation is illustrated in *Figure 13-15* on page 145.

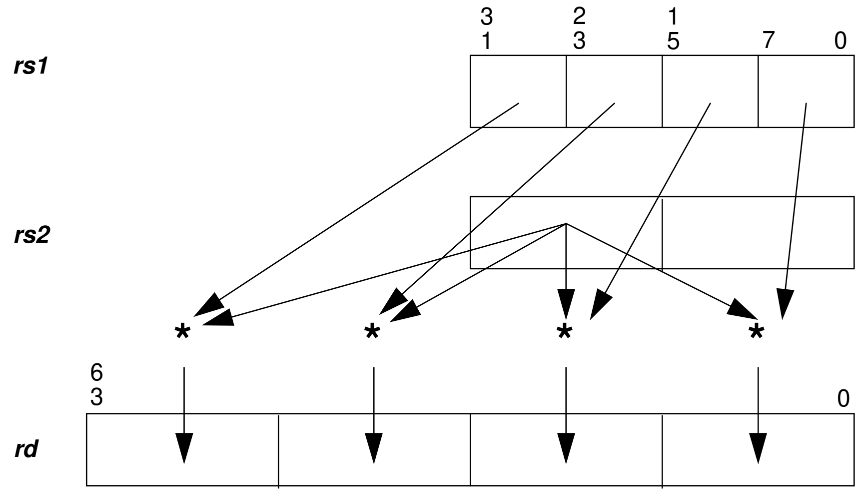


Figure 13-15 FMUL8x16AU Operation

13.4.4.3 FMUL8x16AL

FMUL8x16AL is similar to FMUL8x16AU, except that the least significant 16 bits of the 32-bit *rs2* register are used for the α value.

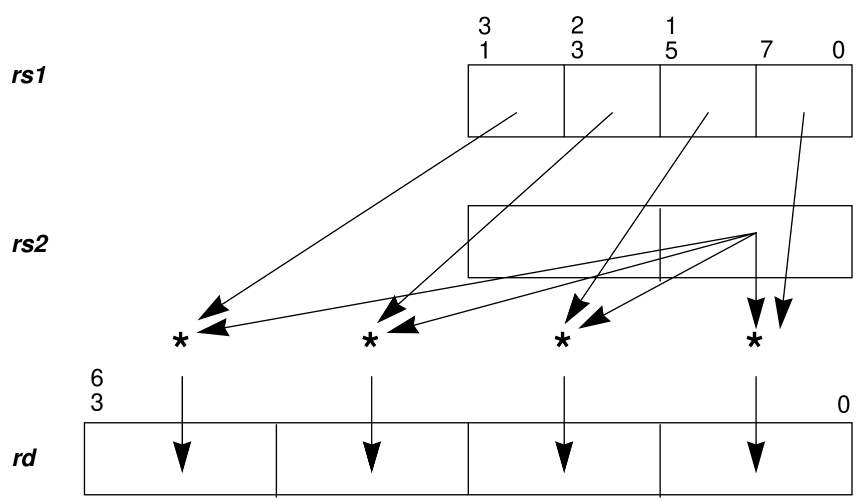


Figure 13-16 FMUL8x16AL Operation

13.4.4.4 FMUL8SUx16

FMUL8SUx16 multiplies the upper 8 bits of each 16-bit signed value in *rs1* by the corresponding signed 16-bit fixed-point signed integer in *rs2*. It rounds the 24-bit product (to the nearest integer assuming a boundary point between 7 and 8) and stores the upper 16 bits of the result into the corresponding 16-bit field of the *rd* register. If the product lies exactly mid- way between two integers, the result is rounded towards positive infinity. *Figure 13-17* illustrates the operation.

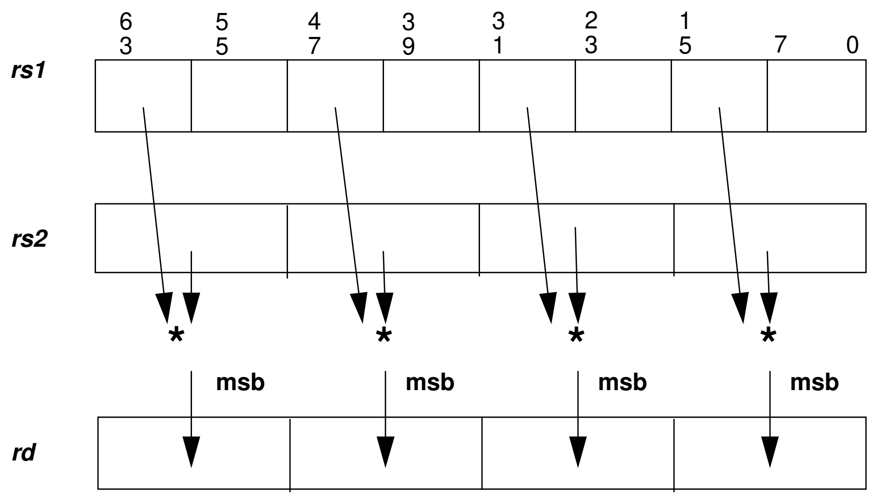


Figure 13-17 FMUL8SUx16 Operation

13.4.4.5 FMUL8ULx16

FMUL8ULx16 multiplies the unsigned lower 8 bits of each 16-bit value in *rs1* by the corresponding fixed point signed integer in *rs2*. Each 24-bit product is sign-extended to 32 bits. The upper 16-bits of the sign extended value are rounded to the nearest integer and stored in the corresponding 16 bits of the *rd* register. In the case that the result is exactly half way between two integers, the result is rounded towards positive infinity. The operation is illustrated in *Figure 13-18*.

Code Example 13-1 16-bit x 16-bit → 16-bit Multiply

```
fmul8sux16 %f0, %f2, %f4
fmul8ulx16 %f0, %f2, %f6
fpadd16    %f4, %f6, %f8
```

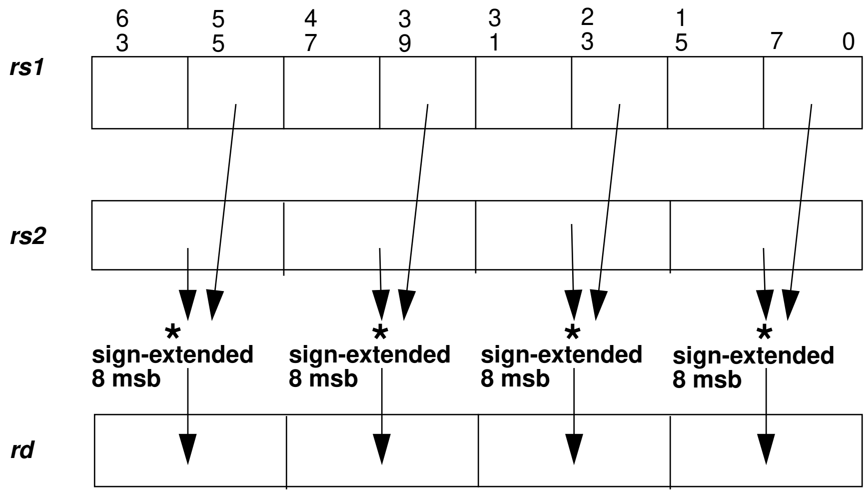


Figure 13-18 FMUL8ULx16 Operation

13.4.4.6 FMULD8SUx16

FMULD8SUx16 multiplies the upper 8 bits of each 16-bit signed value in *rs1* by the corresponding signed 16-bit fixed point signed integer in *rs2*. The 24-bit product is shifted left by 8-bits to make up a 32-bit result. The result is stored in the corresponding 32-bit of the destination *rd* register. The operation is illustrated in *Figure 13-19*.

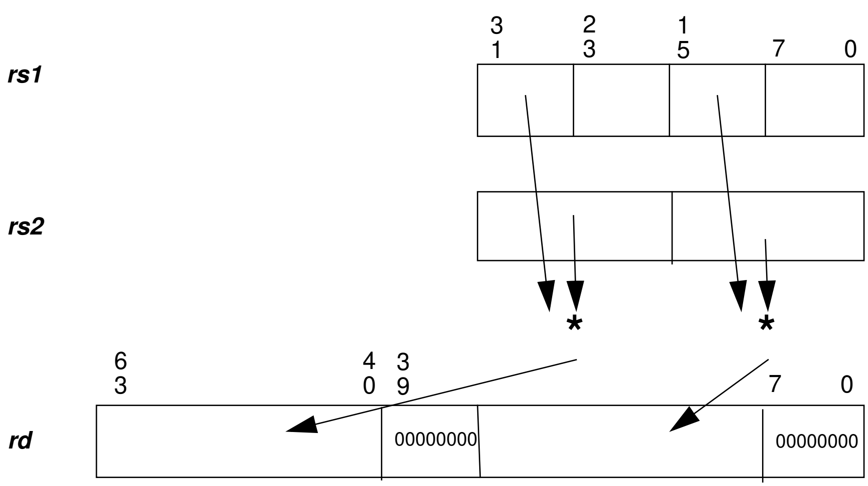


Figure 13-19 FMULD8SUx16 Operation

13.4.4.7 FMULD8ULx16

FMULD8ULx16 multiplies the unsigned lower 8 bits of each 16-bit value in *rs1* by the corresponding fixed point signed integer in *rs2*. Each 24-bit product is sign-extended to 32 bits and stored in the *rd* register. *Figure 13-20* illustrates the operation.

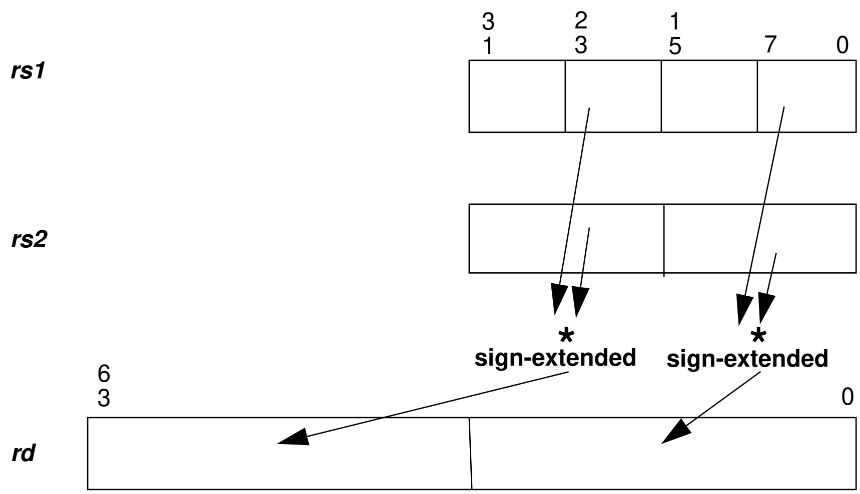


Figure 13-20 FMULD8ULx16 Operation

Code Example 13-2 16-bit x 16-bit → 32-bit Multiply

```
fmuld8sux16%f0, %f2, %f4
fmuld8ulx16%f0, %f2, %f6
fpadd32    %f4, %f6, %f8
```

13.4.5 Alignment Instructions

Table 13-9 Alignment Instruction Opcodes

opcode	opf	operation
ALIGNADDRESS	0 0001 1000	Calculate address for misaligned data access
ALIGNADDRESS_LITTLE	0 0001 1010	Calculate address for misaligned data access, little-endian
FALIGNDATA	0 0100 1000	Perform data alignment for misaligned data

10	rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-21 Alignment Instruction Format (3)

Table 13-10 Alignment Instruction Syntax

Suggested Assembly Language Syntax	
alignaddr	reg_{rs1}, reg_{rs2}, reg_{rd}
alignaddrl	reg_{rs1}, reg_{rs2}, reg_{rd}
faligndata	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

ALIGNADDRESS adds two integer registers, *rs1* and *rs2*, and stores the result, with the least significant 3 bits forced to zero, in the integer *rd* register. The least significant 3 bits of the result are stored in the GSR.*alignaddr_offset* field.

ALIGNADDRESS_LITTLE is the same as ALIGNADDRESS, except that the 2's complement of the least significant 3 bits of the result is stored in GSR.*alignaddr_offset*.

Note – ALIGNADDRL is used to generate the opposite-endian byte ordering for a subsequent FALIGNDATA operation.

FALIGNDATA concatenates two 64-bit floating-point registers, *rs1* and *rs2*, to form a 16-byte value; it stores the result in the 64-bit floating-point *rd* register. The concatenated value contains *rs1* is its upper half and *rs2* in its lower half. Bytes in this value are numbered from most significant to least significant, with the most significant byte being byte 0. Eight bytes are extracted from this value, where the most significant byte of the extracted value is the byte whose number is specified by the GSR.*alignaddr_offset* field.

A byte-aligned 64-bit load can be performed as shown in *Code Example 13-3*.

Code Example 13-3 Byte-Aligned 64-bit Load

```
alignaddr  Address, Offset, Address
ldd        [Address], %f0
ldd        [Address + 8], %f4
faligndata %f0, %f4, %f8
```

Traps

fp_disabled

Note – For good performance, do not use the result of FALIGN as a 32-bit graphics instruction source operand in the next instruction group.

13.4.6 Logical Operate Instructions

Table 13-11 Logical Operate Instructions

opcode	opf	operation
FZERO	0 0110 0000	Zero fill
FZEROS	0 0110 0001	Zero fill, single precision
FONE	0 0111 1110	One fill
FONES	0 0111 1111	One fill, single precision
FSRC1	0 0111 0100	Copy *src1*
FSRC1S	0 0111 0101	Copy *src1*, single precision
FSRC2	0 0111 1000	Copy *src2*
FSRC2S	0 0111 1001	Copy *src2*, single precision
FNOT1	0 0110 1010	Negate (1’s complement) *src1*
FNOT1S	0 0110 1011	Negate (1’s complement) *src1*, single precision
FNOT2	0 0110 0110	Negate (1’s complement) *src2*
FNOT2S	0 0110 0111	Negate (1’s complement) *src2*, single precision
FOR	0 0111 1100	Logical OR
FORS	0 0111 1101	Logical OR, single precision
FNOR	0 0110 0010	Logical NOR
FNORS	0 0110 0011	Logical NOR, single precision
FAND	0 0111 0000	Logical AND
FANDS	0 0111 0001	Logical AND, single precision
FNAND	0 0110 1110	Logical NAND
FNANDS	0 0110 1111	Logical NAND, single precision
FXOR	0 0110 1100	Logical XOR
FXORS	0 0110 1101	Logical XOR, single precision

Table 13-11 Logical Operate Instructions *(Continued)*

opcode	opf	operation
FXNOR	0 0111 0010	Logical XNOR
FXNORS	0 0111 0011	Logical XNOR, single precision
FORNOT1	0 0111 1010	Negated *src1* OR *src2*
FORNOT1S	0 0111 1011	Negated *src1* OR *src2*, single precision
FORNOT2	0 0111 0110	*Src1* OR negated *src2*
FORNOT2S	0 0111 0111	*Src1* OR negated *src2*, single precision
FANDNOT1	0 0110 1000	Negated *src1* AND *src2*
FANDNOT1S	0 0110 1001	Negated *src1* AND *src2*, single precision
FANDNOT2	0 0110 0100	*Src1* AND negated *src2*
FANDNOT2S	0 0110 0101	*Src1* AND negated *src2*, single precision

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-22 Logical Operate Instruction Format (3)

Table 13-12 Logical Operate Instruction Syntax

Suggested Assembly Language Syntax	
fzero	freg_{rd}
fzeros	freg_{rd}
fone	freg_{rd}
fones	freg_{rd}
fsrc1	$freg_{rs1}$, $freg_{rd}$
fsrc1s	$freg_{rs1}$, $freg_{rd}$
fsrc2	$freg_{rs2}$, $freg_{rd}$
fsrc2s	$freg_{rs2}$, $freg_{rd}$
fnot1	$freg_{rs1}$, $freg_{rd}$
fnot1s	$freg_{rs1}$, $freg_{rd}$
fnot2	$freg_{rs2}$, $freg_{rd}$
fnot2s	$freg_{rs2}$, $freg_{rd}$
for	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Table 13-12 Logical Operate Instruction Syntax *(Continued)*

Suggested Assembly Language Syntax	
fnor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fand	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fands	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnand	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fnands	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxnor	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fxnors	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fornot1	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fornot1s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fornot2	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fornot2s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fandnot1	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fandnot1s	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fandnot2	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$
fandnot2	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

The standard 64-bit version of these instructions perform one of sixteen 64-bit logical operations between *rs1* and *rs2*. The result is stored in *rd*. The 32-bit (single-precision) version of these instructions performs 32-bit logical operations.

Note – For good performance, do not use the result of a single logical as part of a 64-bit graphics instruction source operand in the next instruction group. Similarly, do not use the result of a standard logical as a 32-bit graphics instruction source operand in the next instruction group.

Traps

fp_disabled

13.4.7 Pixel Compare Instructions

Table 13-13 Pixel Compare Instruction Opcodes

opcode	opf	operation
FCMPGT16	0 0010 1000	Four 16-bit compare; set *rd* if *src1* > *src2*
FCMPGT32	0 0010 1100	Two 32-bit compare; set *rd* if *src1* > *src2*
FCMPLE16	0 0010 0000	Four 16-bit compare; set *rd* if *src1* ≤ *src2*
FCMPLE32	0 0010 0100	Two 32-bit compare; set *rd* if *src1* ≤ *src2*
FCMPNE16	0 0010 0010	Four 16-bit compare; set *rd* if *src1* ≠ *src2*
FCMPNE32	0 0010 0110	Two 32-bit compare; set *rd* if *src1* ≠ *src2*
FCMPEQ16	0 0010 1010	Four 16-bit compare; set *rd* if *src1* = *src2*
FCMPEQ32	0 0010 1110	Two 32-bit compare; set *rd* if *src1* = *src2*

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-23 Pixel Compare Instruction Format (3)

Table 13-14 Pixel Compare Instruction Syntax

Suggested Assembly Language Syntax	
fcmpgt16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmpgt32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmple16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmple32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmpne16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmpne32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmpeq16	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}
fcmpeq32	$freg_{rs1}$, $freg_{rs2}$, reg_{rd}

Description

Four 16-bit or two 32-bit fixed-point values in *rs1* and *rs2* are compared. The 4-bit or 2-bit results are stored in the corresponding least significant bits of the integer *rd* register. Bit zero of *rd* corresponds to the least significant 16-bit or 32-bit graphics compare result.

For FCMPGT, each bit in the result is set if the corresponding value in *rs1* is greater than the value in *rs2*. Less-than comparisons are made by swapping the operands.

For FCMPLE, each bit in the result is set if the corresponding value in *rs1* is less than or equal to the value in *rs2*. Greater-than-or-equal comparisons are made by swapping the operands.

For FCMPEQ, each bit in the result is set if the corresponding value in *rs1* is equal to the value in *rs2*.

For FCMPNE, each bit in the result is set if the corresponding value in *rs1* is not equal to the value in *rs2*.

Traps

fp_disabled

13.4.8 Edge Handling Instructions

Table 13-15 Edge Handling Instruction Opcodes

opcode	opf	operation
`EDGE8`	0 0000 0000	`Eight 8-bit edge boundary processing`
`EDGE8L`	0 0000 0010	`Eight 8-bit edge boundary processing, little-endian`
`EDGE16`	0 0000 0100	`Four 16-bit edge boundary processing`
`EDGE16L`	0 0000 0110	`Four 16-bit edge boundary processing, little-endian`
`EDGE32`	0 0000 1000	`Four 32-bit edge boundary processing`
`EDGE32L`	0 0000 1010	`Two 32-bit edge boundary processing, little-endian`

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-24 Edge Handling Instruction Format (3)

Table 13-16 Edge Handling Instruction Syntax

Suggested Assembly Language Syntax	
`edge8`	reg_{rs1}, reg_{rs2}, reg_{rd}
`edge8l`	reg_{rs1}, reg_{rs2}, reg_{rd}
`edge16`	reg_{rs1}, reg_{rs2}, reg_{rd}
`edge16l`	reg_{rs1}, reg_{rs2}, reg_{rd}
`edge32`	reg_{rs1}, reg_{rs2}, reg_{rd}
`edge32l`	reg_{rs1}, reg_{rs2}, reg_{rd}

Description

These instructions are used to handle the boundary conditions for parallel pixel scan line loops, where *src1* is the address of the next pixel to render and *src2* is the address of the last pixel in the scan line.

EDGE8L, EDGE16L, and EDGE32L are little-endian versions of EDGE8, EDGE16 and EDGE32. They produce an edge mask that is bit reversed from their big-endian counterparts, but are otherwise the same. This makes the mask consistent with the mask generated by the graphics compare operations (see Section 13.4.7, *Pixel Compare Instructions* on page 153) on little-endian data.

A 2- (EDGE32), 4- (EDGE16), or 8-bit (EDGE8) pixel mask is stored in the least significant bits of *rd*. The mask is computed from left and right edge masks as follows:

1. The left edge mask is computed from the 3 least significant bits (LSBs) of *rs1* and the right edge mask is computed from the 3 LSBs of *rs2*, according to *Table 13-17* (*Table 13-18* for little-endian byte ordering).
2. If 32-bit address masking is disabled (PSTATE.AM = 0, 64-bit addressing) and the upper 61 bits of *rs1* are equal to the corresponding bits in *rs2*, *rd* is set equal to the right edge mask ANDed with the left edge mask.
3. If 32-bit address masking is enabled (PSTATE.AM = 1, 32-bit addressing) is set and the bits <31:3> of *rs1* are equal to the corresponding bits in *rs2*, *rd* is set to the right edge mask ANDd with the left edge mask.
4. Otherwise, *rd* is set to the left edge mask.

The integer condition codes are set the same as a SUBCC instruction with the same operands. End of scan line comparison tests may be performed using edge with an appropriate conditional branch instruction.

Traps

None

Table 13-17 Edge Mask Specification

Edge Size	A2..A0	Left Edge	Right Edge
8	000	1111 1111	1000 0000
8	001	0111 1111	1100 0000
8	010	0011 1111	1110 0000
8	011	0001 1111	1111 0000
8	100	0000 1111	1111 1000
8	101	0000 0111	1111 1100
8	110	0000 0011	1111 1110
8	111	0000 0001	1111 1111
16	00x	1111	1000
16	01x	0111	1100
16	10x	0011	1110
16	11x	0001	1111
32	0xx	11	10
32	1xx	01	11

Table 13-18 Edge Mask Specification (Little-Endian)

Edge Size	A2..A0	Left Edge	Right Edge
8	000	1111 1111	0000 0001
8	001	1111 1110	0000 0011
8	010	1111 1100	0000 0111
8	011	1111 1000	0000 1111
8	100	1111 0000	0001 1111
8	101	1110 0000	0011 1111
8	110	1100 0000	0111 1111
8	111	1000 0000	1111 1111
16	00x	1111	0001
16	01x	1110	0011
16	10x	1100	0111

Table 13-18 Edge Mask Specification (Little-Endian) *(Continued)*

Edge Size	A2..A0	Left Edge	Right Edge
16	11x	1000	1111
32	0xx	11	01
32	1xx	10	11

13.4.9 Pixel Component Distance (PDIST)

Table 13-19 Pixel Component Distance Opcode

opcode	opf	operation
PDIST	0 0011 1110	distance between 8 8-bit components

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-25 Pixel Component Distance Format (3)

Table 13-20 Pixel Component Distance Syntax

Suggested Assembly Language Syntax	
pdist	$freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description

Eight unsigned 8-bit values are contained in the 64-bit *rs1* and *rs2* registers. The corresponding 8-bit values in *rs1* and *rs2* are subtracted (i.e., *rs1* – *rs2*). The sum of the absolute value of each difference is added to the integer in the 64-bit *rd* register. The result is stored in *rd*. Typically, this instruction is used for motion estimation in video compression algorithms.

Note – For good performance, the *rd* operand of PDIST should not reference the result of a non PDIST instruction in the previous two instruction groups.

Traps

fp_disabled

13.4.10 Three-Dimensional Array Addressing Instructions

Table 13-21 Three-Dimensional Array Addressing Instruction Opcodes

opcode	opf	operation
`ARRAY8`	0 0001 0000	`Convert 8-bit 3-D address to blocked byte address`
`ARRAY16`	0 0001 0010	`Convert 16-bit 3-D address to blocked byte address`
`ARRAY32`	0 0001 0100	`Convert 32-bit 3-D address to blocked byte address`

10	rd	11 0110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-26 Three-Dimensional Array Addressing Instruction Format (3)

Table 13-22 Three-Dimensional Array Addressing Instruction Syntax

Suggested Assembly Language Syntax	
array8	reg_{rs1}, reg_{rs2}, reg_{rd}
array16	reg_{rs1}, reg_{rs2}, reg_{rd}
array32	reg_{rs1}, reg_{rs2}, reg_{rd}

Description

These instructions convert three dimensional (3D) fixed-point addresses contained in *rs1* to a blocked-byte address; they store the result in *rd*. Fixed-point addresses typically are used for address interpolation for planar reformatting operations. Blocking is performed at the 64-byte level to maximize external cache block reuse, and at the 64k-byte level to maximize TLB entry reuse, regardless of the orientation of the address interpolation. These instructions specify an element size of 8 (ARRAY8), 16 (ARRAY16) or 32 bits (ARRAY32). The *rs2* operand specifies the power-of-two size of the X and Y dimensions of a 3D image array. The legal values for *rs2* and their meanings are shown in *Table 13-23*. Illegal values produce undefined results in the *rd* register.

Table 13-23 Allowable values for *rs2*

rs2 Value	Number of Elements
0	64
1	128
2	256
3	512
4	1,024
5	2,048

Figure 13-27 shows the format of *rs1*.

Z integer	Z fraction	Y integer	Y fraction	X integer	X fraction
63 55	54 44	43 33	32 22	21 11	10 0

Figure 13-27 Three Dimensional Array Fixed-Point Address Format

The integer parts of X, Y, and Z are converted to the blocked-address formats of *Figure 13-28*, *Figure 13-29*, and *Figure 13-30*, as appropriate.

Upper			Middle			Lower		
Z	Y	X	Z	Y	X	Z	Y	X

20 + 2 isrc2, 17 + 2 isrc2, 17 + isrc2, 17, 13, 9, 5, 4, 2, 0

Figure 13-28 Three Dimensional Array Blocked-Address Format (Array8)

Upper			Middle			Lower			0
Z	Y	X	Z	Y	X	Z	Y	X	

21 + 2 isrc2, 18 + 2 isrc2, 18 + isrc2, 18, 14, 10, 6, 5, 3, 1, 0

Figure 13-29 Three Dimensional Array Blocked-Address Format (Array16)

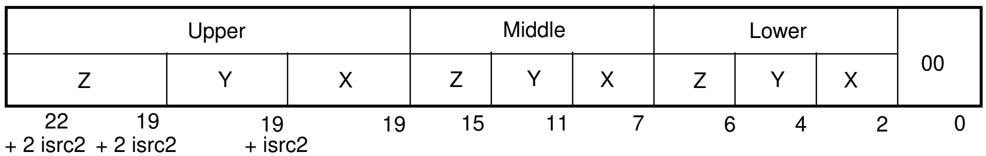


Figure 13-30 Three Dimensional Array Blocked-Address Format (Array32)

The bits above Z upper are set to zero. The number of zeros in the least significant bits is determined by the element size. An element size of eight bits has no zeros, an element size of 16-bits has one zero, and an element size of 32-bits has two zeros. Bits in X and Y above the size specified by *rs2* are ignored.

Note – To maximize reuse of E-cache and TLB data, software should block array references for large images to the 64 kB level. This means processing elements within a 32x64x64 block.

The following code fragment shows assembly of components along an interpolated line at the rate of one component per clock on UltraSPARC IIi:

Code Example 13-4 Assembly of Components along an Interpolated Line

```
        add         Addr, DeltaAddr, Addr
        array8      Addr, %g0, bAddr
        ldda        [bAddr] ASI_FL8_PRIMARY, data
        faligndata  data, accum, accum
```

Traps

None

13.5 Memory Access Instructions

13.5.1 Partial Store Instructions

Table 13-24 Partial Store Opcodes

Opcode	imm_asi	ASI Value	Operation
STDFA	ASI_PST8_P	$C0_{16}$	Eight 8-bit conditional stores to primary address space
STDFA	ASI_PST8_S	$C1_{16}$	Eight 8-bit conditional stores to secondary address space
STDFA	ASI_PST8_PL	$C8_{16}$	Eight 8-bit conditional stores to primary address space, little-endian
STDFA	ASI_PST8_SL	$C9_{16}$	Eight 8-bit conditional stores to secondary address space, little-endian
STDFA	ASI_PST16_P	$C2_{16}$	Four 16-bit conditional stores to primary address space
STDFA	ASI_PST16_S	$C3_{16}$	Four 16-bit conditional stores to secondary address space
STDFA	ASI_PST16_PL	CA_{16}	Four 16-bit conditional stores to primary address space, little-endian
STDFA	ASI_PST16_SL	CB_{16}	Four 16-bit conditional stores to secondary address space, little-endian
STDFA	ASI_PST32_P	$C4_{16}$	Two 32-bit conditional stores to primary address space
STDFA	ASI_PST32_S	$C5_{16}$	Two 32-bit conditional stores to secondary address space
STDFA	ASI_PST832_PL	CC_{16}	Two 32-bit conditional stores to primary address space, little-endian
STDFA	ASI_PST32_SL	CD_{16}	Two 32-bit conditional stores to secondary address space, little-endian

11	rd	11 0111	rs1	i=0	imm_asi	rs2
31 30	29 25	24 19	18 14	13	12 5	4 0

Figure 13-31 Partial Store Format (3)

Table 13-25 Partial Store Syntax

Suggested Assembly Language Syntax	
`stda`	*$freg_{rd}$, [reg_{rs1}] reg_{rs2}, imm_asi*

Description

The partial store instructions are selected by using one of the partial store ASIs with the STDA instruction.

Two 32-bit, four 16-bit or eight 8-bit values from the 64-bit *rd* register are conditionally stored at the address specified by *rs1* using the mask specified by *rs2*. The value in *rs2* has the same format as the result generated by the pixel compare instructions (see Section 13.4.7, *Pixel Compare Instructions* on page 153). The most significant bit of the mask (not the entire register) corresponds to the most significant part of the *rs1* register. The data is stored in little-endian form in memory if the ASI name has a "_LITTLE" suffix; otherwise, it is big-endian.

Note – If the byte ordering is little-endian, the byte enables generated by this instruction are swapped with respect to big-endian.

Traps

fp_disabled

mem_address_not_aligned

data_access_exception

PA_watchpoint

VA_watchpoint

illegal_instruction (when i = 1, no immediate mode is supported. This is not checked if there is a data_access_exception for a non-STDFA opcode).

13.5.2 Short Floating-Point Load and Store Instructions

Table 13-26 Short Floating-Point Load and Store Instruction

Opcode	imm_asi	ASI Value	Operation
LDDFA STDFA	ASI_FL8_P	$D0_{16}$	8-bit load/store from/to primary address space
LDDFA STDFA	ASI_FL8_S	$D1_{16}$	8-bit load/store from/to secondary address space
LDDFA STDFA	ASI_FL8_PL	$D8_{16}$	8-bit load/store from/to primary address space, little-endian

Table 13-26 Short Floating-Point Load and Store Instruction

Opcode	imm_asi	ASI Value	Operation
LDDFA STDFA	ASI_FL8_SL	$D9_{16}$	8-bit load/store from/to secondary address space, little-endian
LDDFA STDFA	ASI_FL16_P	$D2_{16}$	16-bit load/store from/to primary address space
LDDFA STDFA	ASI_FL16_S	$D3_{16}$	16-bit load/store from/to secondary address space
LDDFA STDFA	ASI_FL16_PL	DA_{16}	16-bit load/store from/to primary address space, little-endian
LDDFA STDFA	ASI_FL16_SL	DB_{16}	16-bit load/store from/to secondary address space, little-endian

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	11 0011	rs1	i=0	imm_asi	rs2
11	rd	11 0011	rs1	i=1	simm_13	

Figure 13-32 Format (3) LDDFA

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	11 0111	rs1	i=0	imm_asi	rs2
11	rd	11 0111	rs1	i=1	simm_13	

Figure 13-33 Format (3) STDFA

Table 13-27 Short Floating-Point Load and Store Instruction Syntax

Suggested Assembly Language Syntax	
ldda	[*reg_addr*] *imm_asi*, $freg_{rd}$
ldda	[*reg_plus_imm*] %asi, $freg_{rd}$
stda	$freg_{rd}$, [*reg_addr*] *imm_asi*
stda	$freg_{rd}$, [*reg_plus_imm*] %asi

Description

Short floating-point load and store instructions are selected by using one of the short ASIs with the LDDA and STDA instructions.

These ASIs allow 8- and 16-bit loads or stores to be performed to the floating-point registers. Eight-bit loads can be performed to arbitrary byte addresses. For sixteen bit loads, the least significant bit of the address must be zero, or a *mem_not_aligned* trap is taken. Short loads are zero-extended to the full floating point register. Short stores access the low order 8 or 16 bits of the register.

Little-endian ASIs transfer data in little-endian format in memory; otherwise, memory is assumed to big-endian. Short loads and stores typically are used with the FALIGNDATA instruction (see Section 13.4.5, *Alignment Instructions* on page 148) to assemble or store 64 bits of non-contiguous components.

Traps

fp_disabledPA_watchpoint

VA_watchpoint

mem_address_not_aligned (Checked for opcode implied alignment if the opcode is not LDFA or STDFA)

13.5.3 Block Load and Store Instructions

Table 13-28 Block Load and Store Instruction Opcodes

Opcode	imm_asi	ASI Value	Operation
LDDFA STDFA	ASI_BLK_AIUP	70_{16}	64-byte block load/store from/ to primary address space, user privilege
LDDFA STDFA	ASI_BLK_AIUS	71_{16}	64-byte block load/store from/ to secondary address space, user privilege
LDDFA STDFA	ASI_BLK_AIUPL	78_{16}	64-byte block load/store from/ to primary address space, user privilege, little-endian
LDDFA STDFA	ASI_BLK_AIUSL	79_{16}	64-byte block load/store from/ to secondary address space, user privilege, little-endian
LDDFA STDFA	ASI_BLK_P	$F0_{16}$	64-byte block load/store from/to primary address space
LDDFA STDFA	ASI_BLK_S	$F1_{16}$	64-byte block load/store from/ to secondary address space

Table 13-28 Block Load and Store Instruction Opcodes

Opcode	imm_asi	ASI Value	Operation
LDDFA STDFA	ASI_BLK_PL	$F8_{16}$	64-byte block load/store from/to primary address space, little-endian
LDDFA STDFA	ASI_BLK_SL	$F9_{16}$	64-byte block load/store from/to secondary address space, little-endian
STDFA	ASI_BLK_COMMIT_P	$E0_{16}$	64-byte block commit store to primary address space
STDFA	ASI_BLK_COMMIT_S	$E1_{16}$	64-byte block commit store to secondary address space

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	11 0011	rs1	i=0	imm_asi	rs2
11	rd	11 0011	rs1	i=1	simm_13	

Figure 13-34 Format (3) LDDFA:

31 30	29 25	24 19	18 14	13	12 5	4 0
11	rd	11 0111	rs1	i=0	imm_asi	rs2
11	rd	11 0111	rs1	i=1	simm_13	

Figure 13-35 Format (3) STDFA:

Table 13-29 Block Load and Store Instruction Syntax

Suggested Assembly Language Syntax	
`ldda`	[*reg_addr*] *imm_asi*, $freg_{rd}$
`ldda`	[*reg_plus_imm*] `%asi`, $freg_{rd}$
`stda`	$freg_{rd}$, [*reg_addr*] *imm_asi*
`stda`	$freg_{rd}$, [*reg_plus_imm*] `%asi`

Description

Block load and store instructions are selected by using one of the block transfer ASIs with the LDDA and STDA instructions. These ASIs allow block loads or stores to be performed to the same address spaces as normal loads and stores. Little-endian ASIs access data in little-endian format, otherwise the access is assumed to be big-endian. The byte swapping is performed separately for each of the eight double-precision registers used by the instruction. Endianness does not matter if these instructions are being used for block copy.

Block stores with commit force the data to be written to memory and invalidate copies in all caches, if present. As a result, block commit stores maintain coherency with the I-cache unlike other stores. They do not, however, flush instructions that have already been fetched into the pipeline. Execute a FLUSH, DONE, or RETRY instruction to flush the pipeline before executing the modified code.

LDDA with a block transfer ASI loads 64 bytes of data from a 64-byte aligned memory area into eight double-precision floating-point registers specified by $freg_{rd}$. The lowest addressed eight bytes in memory are loaded into the lowest numbered double-precision *rd* register. An *illegal_instruction* trap is taken if the floating-point registers are not aligned on an eight-double-precision register boundary. The least significant 6 bits of the address must be zero or a *mem_address_not_aligned* trap is taken.

STDA with a block transfer ASI stores data from eight double-precision floating-point registers specified by *rs1* to a 64 byte aligned memory area. The lowest addressed eight bytes in memory are stored from the lowest numbered double precision *freg*. An *illegal_instruction* trap is taken if the floating-point registers are not aligned on an eight register boundary. The least significant 6 bits of the address must be zero, or a *mem_address_not_aligned* trap is taken.

Traps

fp_disabled

illegal_instruction (nonaligned rd. Not checked if opcode is not LDFA or STDFA)

data_access_exception

mem_address_not_aligned (Checked for opcode implied alignment if the opcode is not LDFA or STDFA)

PA_watchpoint

VA_watchpoint

Note – These instructions are used for transferring large blocks of data (more than 256 bytes); for example, BCOPY and BFILL. On UltraSPARC IIi they do not allocate in the D-cache or E-cache on a miss. UltraSPARC IIi updates the E-cache on a hit.

BLD does not provide register dependency interlocks like ordinary load instructions.

Before referencing BLD data, a second BLD (to a different set of registers) or a MEMBAR #Sync must be performed. If a second BLD is used to synchronize against returning data, the UltraSPARC CPU continues execution before all data has been returned. The programmer is then responsible for scheduling instructions so registers are only used when they become valid.

To determine when data is valid, the programmer must count instruction groups containing FP operation instructions (not FP loads or stores). The lowest number register being loaded by the first BLD may be referenced in the first instruction group following the second BLD, using an FP operation instruction only.

The second lowest number register may be referenced in the second instruction group containing an FP operation instruction, and so on. The best case grouping of FP operations should be assumed, that is, issuing any M-Class FP operation in the same group as any possible A-Class FP operation. (The UltraSPARC IIi CPU can issue two FP operation instructions simultaneously, assuming they are in different classes).

If this BLD/BLD synchronization mechanism is used, the initial reference to the BLD data must be an FP operation instruction (not a FP store), and only instruction groups with FP operation instructions are counted when determining BLD data availability.

If these rules are violated, data from before or after the BLD may be returned.

If a MEMBAR #Sync is used to synchronize on BLD data, there are no restrictions on data usage, although this will cause lower performance. No other MEMBARs can be used to provide data synchronization for BLD.

FP operation instructions can be issued in a single group with FP stores. If BLD/BLD synchronization is used, FP operations and FP stores can be interlaced. This allows an FP operation to reference the returning data before using the data in any FP store (normal store, or block store).

Typically, the FP operation instruction is a FMOVD or FALIGNDATA.

The UltraSPARC CPU also continues execution, without register interlocks, before all of the store data for BSTs is transferred from the register file.

If store data registers are overwritten before the next block store or MEMBAR #Sync instruction, the following rule must be observed: The first register can be overwritten in the same instruction group as the BST, the second register can be overwritten in the instruction group following the block store, and so on. If this rule is violated, the store may use the old or the new overwritten data.

When determining correctness for a code sample, be aware that UltraSPARC implementations may interlock more than required above, for instance there may be partial register interlocks. (for instance on the lowest-numbered register).

Code that does not meet the above constraints may appear to work on a particular platform because of implemenation-dependent timing that causes additional dependencies or delays to be created.

However, for portability across all UltraSPARC implementations, all of the above rules must be followed.

There must be a MEMBAR `#Sync` or a trap following a BST before executing a DONE, RETRY, or WRPR to PSTATE instruction. If this is rule is violated, instructions after the DONE, RETRY, or WRPR to PSTATE may not see the effects of the updated PSTATE.

BLD does not follow memory model ordering with respect to stores. In particular, read-after-write and write-after-read hazards to overlapping addresses are not detected. The side effects bit associated with the access is ignored (see Section 15.2, *Translation Table Entry (TTE)* on page 197). If ordering with respect to earlier stores is important (for example, a block load that overlaps previous stores), then there must be an intervening MEMBAR `#StoreLoad` or stronger MEMBAR. If ordering with respect to later stores is important (e.g. a block load that overlaps a subsequent store), then there must be an intervening MEMBAR `#LoadStore` or reference to the block load data. This restriction does not apply when a trap is taken, so the trap handler need not consider pending block loads. If the BLD overlaps a previous or later store and there is no intervening MEMBAR, trap, or data reference, the BLD may return data from before or after the store.

Compatibility Note – Prior UltraSPARCs may have provided the first two registers at the same time. If code depends upon this unsupported behavior it must be modified for UltraSPARC IIi.

BST does not follow memory model ordering with respect to loads, stores or flushes. In particular, read-after-write, write-after-write, flush after write and write-after-read hazards to overlapping addresses are not detected. The side effects bit associated with the access is ignored. If ordering with respect to earlier or later loads or stores is important then there must be an intervening reference to the load data (for earlier loads), or appropriate MEMBAR instruction. This restriction does not apply when a trap is taken, so the trap handler does not have to worry about pending block stores. If the BST overlaps a previous load and there is no intervening load data reference or MEMBAR `#LoadStore` instruction, the load may return data from before or after the store and the contents of the block are undefined. If the BST overlaps a later load and there is no intervening trap or MEMBAR `#StoreLoad` instruction, the contents of the block are undefined. If the BST overlaps a later store or flush and there is no intervening trap or MEMBAR `#StoreStore` instruction, the contents of the block are undefined.

Block load and store operations do not obey the ordering restrictions of the currently selected processor memory model (TSO, PSO, or RMO); block operations always execute under an RMO memory ordering model. Explicit MEMBAR instructions are required to order block operations among themselves or with respect to normal loads and stores. In addition, block operations do not conform to dependence order on the issuing processor; that is, no read-after-write or writer-after-read checking occurs between block loads and stores. Explicit MEMBARs are required to enforce dependence ordering between block operations that reference the same address.

Typically, BLD and BST are used in loops where software can ensure that there is no overlap between the data being loaded and the data being stored. The loop is preceded and followed by the appropriate MEMBARs to ensure that there are no hazards with loads and stores outside the loops. *Code Example 13-5* on page 170 illustrates the inner loop of a byte-aligned block copy operation.

Note that the loop must be unrolled twice to achieve maximum performance. All FP registers are double-precision. Eight versions of this loop are needed to handle all the cases of double word misalignment between the source and destination.

Code Example 13-5 Byte-Aligned Block Copy Inner Loop

```
loop:
  faligndata      %f0, %f2, %f34
  faligndata      %f2, %f4, %f36
  faligndata      %f4, %f6, %f38
  faligndata      %f6, %f8, %f40
  faligndata      %f8, %f10, %f42
  faligndata      %f10, %f12, %f44
  faligndata      %f12, %f14, %f46
  addcc           l0, -1, l0
  bg,pt           l1
  fmovd           %f14, %f48
  (end of loop handling)
l1:ldda           [regaddr] ASI_BLK_P, %f0
  stda            %f32, [regaddr] ASI_BLK_P
  faligndata      %f48, %f16, %f32
  faligndata      %f16, %f18, %f34
  faligndata      %f18, %f20, %f36
  faligndata      %f20, %f22, %f38
  faligndata      %f22, %f24, %f40
  faligndata      %f24, %f26, %f42
  faligndata      %f26, %f28, %f44
  faligndata      %f28, %f30, %f46
  addcc           l0, -1, l0
  be,pnt          done
  fmovd           %f30, %f48
  ldda            [regaddr] ASI_BLK_P, %f16
  stda            %f32, [regaddr] ASI_BLK_P
  ba              loop
  faligndata      %f48, %f0, %f32
done:  (end of loop processing)
```

13.6 Additional Instructions

13.6.1 Atomic Quad Load

Table 13-30 Atomic Quad Load Opcodes

Opcode	imm_asi	ASI Value	Operation
`LDDA`	ASI_NUCLEUS_QUAD_LDD	24_{16}	`128-bit atomic load`
`LDDA`	ASI_NUCLEUS_QUAD_LDD_L	$2C_{16}$	`128-bit atomic load, little endian`

11	rd	01 0011	rs1	i=0	imm_asi	rs2

11	rd	01 0011	rs1	i=1	simm_13
31 30	29 25	24 19	18 14	13	12 5 4 0

Figure 13-36 Format (3) LDDA

Table 13-31 Atomic Quad Load Syntax

Suggested Assembly Language Syntax	
`ldda`	`[`*`reg_addr`*`]` *`imm_asi`*`,` *`reg`*$_{rd}$
`ldda`	`[`*`reg_plus_imm`*`] %asi,` *`reg`*$_{rd}$

Description

These ASIs are used with the LDDA instruction to atomically read a 128-bit data item. They are intended to be used by the TLB miss handler to access TSB entries without requiring locks. The data is placed in an even/odd pair of 64-bit integer registers. The lowest address 64-bits is placed in the even register; the highest address 64-bits is placed in the odd register. The reference is made from the nucleus context. In addition to the usual traps for LDDA using a privileged ASI, a *data_access_exception* trap is taken for a noncacheable access, or use with any instruction other than LDDA. A *mem_address_not_aligned* trap is taken if the access is not aligned on a 128-bit boundary.

Traps

fp_disabled

PA_watchpoint

VA_watchpoint

mem_address_not_aligned (Checked for opcode implied alignment if the opcode is not LDFA or STDFA)

data_access_exception

13.6.2 SHUTDOWN

Table 13-32 SHUTDOWN Opcode

opcode	opf	operation
SHUTDOWN	0 1000 0000	Shutdown to enter power down mode

10	—	11 0110	—	opf	—
31 30	29 25	24 19	18 14	13 5	4 0

Figure 13-37 SHUTDOWN Instruction Format (3)

Table 13-33 SHUTDOWN Syntax

Suggested Assembly Language Syntax
shutdown

Description

The EPA Energy Star specification requires a system standby power consumption of less than 30 W (excluding the system monitor).

To enter SHUTDOWN mode, UltraSPARC IIi software saves everything to disk and the power supply is turned off. A timer turns the power back on after 30 minutes. UltraSPARC IIi does not support the full feature set of some earlier PCI-based UltraSPARC systems, principally to avoid the circuit complexity of maintaining memory refresh while the processor is shut down.

Invoking the SHUTDOWN instruction causes all processor, cache and memory state to be lost. A power-on reset (POR) must be used restart the processor. A status bit indicates the reason for the POR. This instruction stops all internal clocks, achieving the lowest possible power consumption while the power supply is on.

To leave the system and external cache interface in a clean state, the SHUTDOWN instruction waits for all outstanding transactions to be completed before sending a shutdown signal to the internal clock generator. The internal clock generator asserts the internal reset for 19 clocks to force the chip into a safe state, and then stops the internal clock and the PLL. The internal clock is left in the high state. All external signals should be left in the normal reset state.

An external power-down signal (EPD) is activated by the clock generator at the same time as the internal reset. This signal is used to put the E-cache RAMs in standby mode.

This is a privileged instruction; an attempt to execute it while in non-privileged mode causes a *privileged_opcode* trap.

Compatibility Note – When the processor is reset, UPA64S, PCI, and APB are also reset.

Traps

privileged_opcode

CHAPTER 14

Implementation Dependencies

14.1 SPARC-V9 General Information

14.1.1 Level-2 Compliance (Impdep #1)

The UltraSPARC IIi CPU is designed to meet Level-2 SPARC-V9 compliance. It

- Correctly interprets all non-privileged operations, and
- Correctly interprets all privileged elements of the architecture.

Note – System emulation routines (for example, quad-precision floating-point operations) shipped with UltraSPARC IIi also must be Level-2 compliant.

14.1.2 Unimplemented Opcodes, ASIs, and ILLTRAP

SPARC-V9 unimplemented, *reserved*, ILLTRAP opcodes, and instructions with invalid values in *reserved* fields (other than *reserved* FPops or fields in graphics instructions that reference floating-point registers and the *reserved* field in the Tcc instruction) encountered during execution cause an *illegal_instruction* trap. The *reserved* field in the Tcc instruction is not checked because SPARC-V8 did not reserve this field. Reserved FPops and invalid values in *reserved* fields in graphics instructions that reference floating-point registers cause an *fp_exception_other* (with *FSR.ftt=unimplemented_FPop)* trap. Unimplemented and *reserved* ASI values cause a *data_access_exception* trap.

14.1.3 Trap Levels (Impdep #37, 38, 39, 40, 114, 115)

UltraSPARC IIi supports five trap levels; that is, MAXTL=5. Normal execution is at TL0. Traps at MAXTL–1 cause the CPU to enter RED_state. If a trap is generated while the CPU is operating at TL = MAXTL, the CPU will enter error_state and generate a Watchdog Reset (WDR). CWP updates for window traps that cause entry to error_state are the same as when error_state is not entered.

A processor normally executes at trap level 0 (execute_state, TL0). The trap handling mechanism in SPARC-V9 differs from SPARC-V8 when a trap or error condition is encountered at TL0. In SPARC-V8, the CPU enters trap state and system (privileged) software must save enough processor state to guarantee that any error condition detected while in the trap handler will not put the CPU into error_state (that is, cause a reset). Then the trap routine is entered to process the erroneous condition. Upon completion of trap processing, the state of the CPU is restored before returning to the offending code or terminating the process. This time-consuming operation is necessary because SPARC-V8 does not support nested traps.

In SPARC-V9, a trap makes the CPU enter the next higher trap level, which is a very fast and efficient process because there is one set of trap state registers for each trap level. After saving the most important machine states (PC, next PC, PSTATE) on the trap stack at this level, the trap (or error) condition is processed.

For a complete description of traps and RED_state handling, see Section 17.4, *Machine State after Reset and in RED_state* on page 261.

Note – The RED_state trap vector address (RSTVaddr) is 256 MB below the top of the virtual address space; this is, at virtual address FFFF FFFF F000 0000_{16}, which is passed through to physical address 1FF F000 0000_{16} in RED_state. UltraSPARC IIi has a second RSTV available — see *RED_state Trap Vector* on page 260.

14.1.4 Alternate RSTV support

UltraSPARC IIi has a pin to select a second RSTV to allow use of PC compatible SuperIO chips on a PCI bus. See Section 17.2.7.3, *Reset_Control Register (0x1FE.0000.F020)* on page 257 and Section 17.3.2, *RED_state Trap Vector* on page 260.

14.1.5 Trap Handling (Impdep #16, 32, 33, 35, 36, 44)

UltraSPARC IIi supports precise trap handling for all operations except for deferred or disrupting traps from hardware failures encountered during memory accesses. These failures are discussed in Section 16.2, *Deferred Errors* on page 232 and

Section 16.3, *Disrupting Errors* on page 234. UltraSPARC IIi implements precise traps, interrupts, and exceptions for all instructions, including long latency floating-point operations. Five traps levels are supported, which allows graceful recovery from faults. The trap levels are shown in *Figure 14-1*. UltraSPARC IIi can efficiently execute kernel code even in the event of multiple nested traps, promoting processor efficiency while dramatically reducing the system overhead needed for trap handling. Three sets of alternate globals are selected for different kinds of traps:

- MMU globals for memory faults
- Interrupt globals, and
- Alternate globals for all other exceptions.

This further increases OS performance, providing fast trap execution by avoiding the need to save and restore registers while processing exceptions.

Level 0: Normal Program Execution

Level 1: System Calls, Interrupt Handlers, Emulation

Level 2: Exceptions in Common OS Routines

Level 3: Page Fault Handlers

Level 4: RED_state Handler

Figure 14-1 Nested Trap Levels

All traps supported in UltraSPARC IIi are listed in *Table 6-12* on page 54.

14.1.6 SIGM Support (Impdep #116)

UltraSPARC IIi initiates a Software-Initiated Reset (SIR) by executing a SIGM instruction while in privileged mode. When in non-privileged mode, SIGM behaves as a NOP. See also Section 17.2.3, *Watchdog Reset (WDR) and error_state* on page 253.

14.1.7 44-bit Virtual Address Space

UltraSPARC IIi supports a 44-bit subset of the full 64-bit virtual address space. Although the full 64 bits are generated and stored in integer registers, legal addresses are restricted to two equal halves at the extreme lower and upper portions of the full virtual address space. Virtual addresses between 0000 0800 0000 0000_{16} and FFFF F7FF FFFF $FFFF_{16}$ inclusive lie within a "VA Hole," are termed "out-of-range," and are illegal. Prior UltraSPARC implementations introduced the additional restriction on software to not use pages within 4 GB of the VA hole as instruction pages to avoid problems with prefetching into the VA hole. UltraSPARC IIi assumes that this convention is followed for similar reasons. Note that there are no trap mechanisms to detect a violation of this convention. Address translation and MMU related descriptions can be found in Section 4.2, *Virtual Address Translation* on page 23.

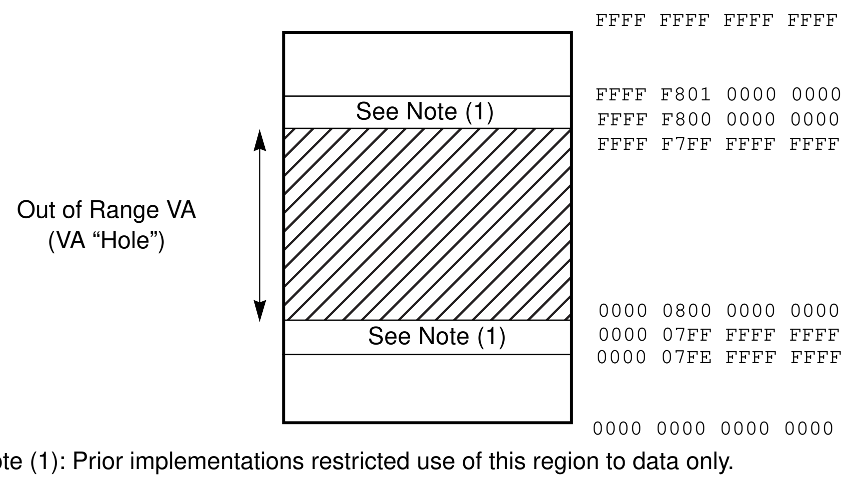


Figure 14-2 UltraSPARC IIi's 44-bit Virtual Address Space, with Hole (Same as *Figure 4-2* on page 25)

Note – Throughout this document, when virtual address fields are specified as 64-bit quantities, they are assumed to be sign-extended based on VA<43>.

A number of state registers are affected by the reduced virtual address space. TBA, TPC, TNPC, VA and PA watchpoint, and DMMU SFAR registers are 44-bits, sign-extended to 64-bits on read accesses. No checks are done when these registers are written by software. It is the responsibility of privileged software to properly update these registers.

An out of range address during an instruction access causes an *instruction_access_exception* trap if PSTATE.AM is not set.

If the target address of a JMPL or RETURN instruction is an out-of-range address and PSTATE.AM is not set, a trap is generated with the PC = the address of the JMPL or RETURN instruction and the trap type in the I-MMU SFSR register. This *instruction_access_exception* trap is lower priority than other traps on the JMPL or RETURN (*illegal_instruction* due to nonzero reserved fields in the JMPL or RETURN, *mem_address_not_aligned* trap, or *window_fill* trap), because it really applies to the target. The trap handler can determine the out-of-range address by decoding the JMPL instruction from the code.

All other control transfer instructions trap on the PC of the target instruction along with different status in the I-MMU SFSR register. Because the PC is sign-extended to 64 bits, the trap handler must adjust the PC value to compute the faulting address by XORing ones into the upper 20 bits. See also Section 15.9.4, *I-/D-MMU Synchronous Fault Status Registers (SFSR)* on page 216 and Section 15.9.5, *I-/D-MMU Synchronous Fault Address Registers (SFAR)* on page 218.

When a trap occurs on the delay slot of a taken branch or call whose target is out-of-range, or the last instruction below the VA hole, UltraSPARC IIi records the fact that nPC points to an out of range instruction. If the trap handler executes a DONE or RETRY without saving nPC, the *instruction_access_exception* trap is taken when the instruction at nPC is executed. If nPC is saved and subsequently restored by the trap handler, the fact that nPC points to an out of range instruction is lost. To guarantee that all out of range instruction accesses cause traps, software should not map addresses within 2^{31} bytes of either side of the VA hole as executable.

An out of range address during a data access results in a *data_access_exception* trap if PSTATE.AM is not set. Because the D-MMU SFAR contains only 44 bits, the trap handler must decode the load or store instruction if the full 64-bit virtual address is needed. See also Section 15.9.4, *I-/D-MMU Synchronous Fault Status Registers (SFSR)* on page 216 and Section 15.9.5, *I-/D-MMU Synchronous Fault Address Registers (SFAR)* on page 218.

14.1.8 TICK Register

UltraSPARC IIi implements a 63-bit TICK counter. For the state of this register at reset, see *Table 17-1* on page 256.

Table 14-1 TICK Register Format

Bits	Field	Use	RW
<63>	NPT	Non-privileged Trap enable	RW
<62:0>	counter	Elapsed CPU clock cycle counter	RW

NPT: Non-privileged Trap enable. If set, an attempt by non-privileged software to read the TICK register causes a *privileged_action* trap. If clear, nonprivileged software can read this register with the RDTICK instruction. This register can only be written by privileged software. A write attempt by nonprivileged software causes a *privileged_action* trap.

counter: 63-bit elapsed CPU clock cycle counter.

Note – TICK.NPT is set and TICK.counter is cleared after both a Power-On-Reset (POR) and an Externally Initiated Reset (XIR).

14.1.9 Population Count Instruction (POPC)

The population count instruction is emulated in software rather that being executed in hardware.

14.1.10 Secure Software

To establish an enhanced security environment, it may be necessary to initialize certain processor states between contexts. Examples of such states are the contents of integer and floating-point register files, condition codes, and state registers. See also Section 14.2.2, *Clean Window Handling (Impdep #102)*.

14.1.11 Address Masking (Impdep #125)

When PSTATE.AM=1, the CALL, JMPL, and RDPC instructions and all traps transmit zero in the high-order 32-bits of the PC to their specified destination registers.

14.2 SPARC-V9 Integer Operations

14.2.1 Integer Register File and Window Control Registers (Impdep #2)

UltraSPARC IIi implements an eight window 64-bit integer register file; that is, NWINDOWS = 8. UltraSPARC IIi truncates values stored in the CWP, CANSAVE, CANRESTORE, CLEANWIN, and OTHERWIN registers to three bits. This includes implicit updates to these registers by SAVE(D) and RESTORE(D) instructions. The upper two bits of these registers read as zero.

14.2.2 Clean Window Handling (Impdep #102)

SPARC-V9 introduced the concept of "clean window" to enhance security and integrity during program execution. A clean window is defined to be a register window that contains either all zeroes or addresses and data that belong to the current context. The CLEANWIN register records the number of available clean windows.

When a SAVE instruction requests a window, and there are no more clean windows, a *clean_window* trap is generated. System software must then initialize all registers in the next available window, or windows, to zero before returning to the requesting context.

14.2.3 Integer Multiply and Divide

Integer multiplications (MULScc, SMUL{cc}, MULX) and divisions (SDIV{cc}, UDIV{cc}, UDIVX) are executed directly in hardware.

Multiplications are done 2 bits at a time with early exit when the final result is generated. Divisions use a 1-bit non-restoring division algorithm.

Note – For best performance, the smaller of the two operands of a multiply should be the rs1 operand.

14.2.4 Version Register (Impdep #2, 13, 101, 104)

Consult the product data sheet for the content of the Version Register for an implementation. For the state of this register after resets, see *Table 17-5* on page 261.

Table 14-2 Version Register Format

Bits	Field	Use	RW
<63:48>	manuf	Manufacturer identification	R
<47:32>	impl	Implementation identification	R
<31:24>	mask	Mask set version	R
<23:16>	Reserved	—	R
<15:8>	maxtl	Maximum trap level supported	R
<7:5>	Reserved	—	R
<4:0>	maxwin	Maximum number of windows of integer register file.	R

manuf: 16-bit manufacturer code, 0017_{16} (TI JEDEC number), that identifies the manufacturer of an UltraSPARC IIi CPU.

impl:1 6-bit implementation code, 0010_{16}, that uniquely identifies an UltraSPARC IIi-class CPU. *Table 14-3* shows the VER.impl values for each UltraSPARC IIi model.

Table 14-3 VER.impl Values by UltraSPARC IIi Model

	UltraSPARC-I	UltraSPARC-II
VER.impl	0010_{16}	0011_{16}

mask: 8-bit mask set revision number that identifies the mask set revision of this UltraSPARC IIi. This is subdivided into a 4 bit major mask number <31:28> and a 4-bit minor mask number <27:24>. The major number starts at zero and is incremented for each all-layer mask revision. The minor number starts at zero for each major revision, and is incremented for each less-than-all-layer mask revision.

maxtl: Maximum number of supported trap levels beyond level 0; the same as the largest possible value for the TL register; for UltraSPARC IIi, maxtl=5

maxwin: Maximum index number available for use as a valid CWP value. The value is NWINDOWS–1; for UltraSPARC IIi maxwin=7.

14.3 SPARC-V9 Floating-Point Operations

14.3.1 Subnormal Operands & Results; Non-standard Operation

UltraSPARC IIi handles some cases of subnormal operands or results directly in hardware and traps on the rest. In the trapping cases, an *fp_exception_other* (with *FSR.ftt=2*, *unfinished_FPop*) trap is signalled and these operations are handled in system software. The unfinished trapping cases are listed in Table 14-4, and Table 14-5.

Because trapping on subnormal operands and results can be costly, UltraSPARC IIi supports the non-standard result option of the SPARC-V9 architecture. If FSR.NS = 1, subnormal operands or results encountered in trapping cases are flushed to zero and the *unfinished_FPop* floating-point trap type are not taken.

14.3.1.1 Subnormal Operands

If FSR.NS=1, the subnormal operands of these operations are replaced by zeroes with the same sign. An inexact exception is signalled in this case, which causes an *fp_exception_ieee_754* trap if enabled by FSR.TEM. If FSR.NS=0, subnormal operands generate traps according to *Table 14-4* on page 183. E_R is the biased exponent of the result before rounding.

Table 14-4 Subnormal Operand Trapping Cases (NS=0)

Operations	One Subnormal Operand	Two Subnormal Operands
F(sd)TO(ix) F(sd)TO(ds) FSQRT(sd)	Unfinished trap always	—
FADD/SUB(sd) FSMULD	Unfinished trap always	Unfinished trap always
FMUL(sd) FDIV(sd)	Unfinished trap if no overflow and: $-25 < E_R$ (SP); $-54 < E_R$ (DP)	Unfinished trap always

14.3.1.2 Subnormal Results

If FSR.NS=1, the subnormal results are replaced by zero with the same sign. Underflow and inexact exceptions are signalled in this case. This will cause an *fp_exception_ieee_754* trap if enabled by FSR.TEM (only *ufc* will be set in FSR.*cexc* when underflow trap is enabled, otherwise only *nxc* will be set when inexact trap is enabled). If FSR.NS=0, then subnormal results generate traps according to Table 14-5. For FDTOS and FADD, E_R is the biased exponent of the result before rounding. For multiply, E_R is the biased sum of the exponents plus one. For divide, E_R is the biased difference of the exponents of the operands.

Table 14-5 Subnormal Result Trapping Cases (NS=0)

Operations	Trap
FDTOS FADD/SUB(sd) FMUL(sd)	Unfinished trap if: $-25 < E_R < 1$ (SP) $-54 < E_R < 1$ (DP)
FDIV(sd)	Unfinished trap if: $-25 < E_R \leq 1$ (SP) $-54 < E_R \leq 1$ (DP)

14.3.2 Overflow, Underflow, and Inexact Traps (Impdep #3, 55)

UltraSPARC IIi implements precise floating-point exception handling. Underflow is detected before rounding. Prediction of overflow, underflow and inexact traps for divide and square root is used to simplify the hardware.

For divide, pessimistic prediction occurs when underflow/overflow can not be determined from examining the source operand exponents. For divide and square root, pessimistic prediction of inexact occurs unless one of the operands is a zero, NAN or infinity. When pessimistic prediction occurs and the exception is enabled, an *fp_exception_other* (with *FSR.ftt*=2, *unfinished_FPop*) trap is generated. System software will properly handle these cases and resume execution. If the exception is not enabled, the actual result status is used to update the aexec bits of the fsr.

Note – Major performance degradation may be observed while running with the inexact exception enabled.

14.3.3 Quad-Precision Floating-Point Operations (Impdep #3)

All quad-precision floating-point instructions, listed in *Table 14-6*, cause an *fp_exception_other* (with FSR.*ftt=3, unimplemented_FPop)* trap. These operations are emulated in system software.

Table 14-6 Unimplemented Quad-Precision Floating-Point Instructions

Instruction	Description
F{s,d}TOq	Convert single-/double- to quad-precision floating-point
F{i,x}TOq	Convert 32-/64-bit integer to quad-precision floating-point
FqTO{s,d}	Convert quad- to single-/double-precision floating-point
FqTO{i,x}	Convert quad-precision floating-point to 32-/64-bit integer
FCMP{E}q	Quad-precision floating-point compares
FMOVq	Quad-precision floating-point move
FMOVqcc	Quad-precision floating-point move, if condition is satisfied
FMOVqr	Quad-precision floating-point move if register match condition
FABSq	Quad-precision floating-point absolute value
FADDq	Quad-precision floating-point addition
FDIVq	Quad-precision floating-point division
FdMULq	Double- to quad-precision floating-point multiply
FMULq	Quad-precision floating-point multiply
FNEGq	Quad-precision floating-point negation
FSQRTq	Quad-precision floating-point square root
FSUBq	Quad-precision floating-point subtraction

14.3.4 Floating Point Upper and Lower Dirty Bits in FPRS Register

The FPRS_dirty_upper (DU) and FPRS_dirty_lower (DL) bits in the Floating-Point Registers State (FPRS) Register are set when an instruction that modifies the corresponding upper and lower half of the floating-point register file is dispatched. Floating-point register file modifying instructions include floating-point operate, graphics, floating-point loads and block load instructions.

The FPRS.DU and FPRS.DL may be set pessimistically, even though the instruction that modified the floating-point register file is nullified.

14.3.5 Floating-Point Status Register (FSR) (Impdep #13, 19, 22, 23, 24)

UltraSPARC IIi supports precise-traps and implements all three exception fields (TEM, *cexc*, and *aexc*) conforming to IEEE Standard 754-1985. The state of the FSR after reset is documented in *Table 17-5* on page 261.

Table 14-7 Floating-Point Status Register Format

Bits	Field	Use	RW
<63:38>	Reserved	—	R
<37:36>	fcc3	Floating-point condition code (set 3)	RW
<35:34>	fcc2	Floating-point condition code (set 2)	RW
<33:32>	fcc1	Floating-point condition code (set 1)	RW
<31:30>	RD	Rounding direction	RW
<29:28>	u	Unused	R
<27:23>	TEM	IEEE-754 trap enable mask	RW
<22>	NS	Non-standard floating-point results	R
<21:20>	Reserved	—	R
<19:17>	ver	FPU version number	R
<16:14>	ftt	Floating-point trap type	RW
<13:>	qne	Floating-point deferred-trap queue (FQ) not empty	RW
<12>	u	Unused	R
<11:10>	fcc0	Floating-point condition code (set 0)	RW
<9:5>	aexc	Accumulated outstanding exceptions	RW
<4:0>	cexc	Current outstanding exceptions	RW

u: Unused field, read as 0.

Note – The LD{X}FSR instruction should write zeroes to the **u** fields; undefined values (read as 0) of these fields are stored by the ST{X}FSR instruction.

***fcc3, fcc2, fcc1, fcc0*:** Four sets of 2-bit floating-point condition codes, which are modified by the FCMP{E} (and LD{X}FSR) instructions. The FBfcc, FMOVcc, and MOVcc instructions use one of these condition code sets to determine conditional control transfers and conditional register moves.

Note – *fcc0* is the same as the *fcc* in SPARC-V8.

RD: IEEE Std. 754-1985 Rounding Direction.

Table 14-8 Floating-Point Rounding Modes

RD	Round Toward
0	Nearest (even if tie)
1	0
2	$+\infty$
3	$-\infty$

TEM: 5-bit trap enable mask for the IEEE-754 floating-point exceptions. If a floating-point operate instruction produces one or more exceptions, the corresponding *cexc/aexc* bits are set and an *fp_exception_ieee_754* (with *FSR.ftt=1, IEEE_754_exception)* exception is generated.

NS: When this field = 0, UltraSPARC IIi produces IEEE-754 compatible results. In particular, subnormal operands or results may cause a trap. When this field=1, UltraSPARC IIi may deliver a non-IEEE-754 compatible result. In particular, subnormal operands and results may be flushed to zero. See *Table 14-4* and *Table 14-5* on page 184.

***ver*:** his field identifies a particular implementation of the UltraSPARC IIi FPU architecture.

***ftt*:** The 3-bit floating point trap type field is set whenever an floating-point instruction causes the *fp_exception_ieee_754* or *fp_exception_other* traps.

Table 14-9 Floating-Point Trap Type Values

ftt	Floating-Point Trap Type	Trap Signalled
0	None	—
1	IEEE_754_exception	fp_exception_ieee_754
2	unfinished_FPop	fp_exception_other
3	unimplemented_FPop	fp_exception_other
4	sequence_error	fp_exception_other

Table 14-9 Floating-Point Trap Type Values *(Continued)*

ftt	Floating-Point Trap Type	Trap Signalled
5	hardware_error	—
6	invalid_fp_register	—
7	reserved	—

Note – UltraSPARC IIi neither detects nor generates the *hardware_error* or *invalid_fp_register* trap types directly in hardware.

Note – UltraSPARC IIi does not contain an FQ. An attempt to read the FQ with a RDPR instruction causes an *illegal_instruction* trap.

Note – SPARC-V8-compatible programs should set the least significant bit of the floating-point register number to zero for all double-precision instructions. Violation of this SPARC-V8 architectural constraint may result in unexpected program behavior.

---

qne: This bit is not used, because UltraSPARC IIi implements precise floating-point exceptions.

aexc: 5-bit accrued exception field accumulates IEEE 754 exceptions while floating-point exception traps are disabled (that is, FSR.TEM=0).

cexc: 5-bit current exception field indicates the most recently generated IEEE 754 exceptions.

14.4 SPARC-V9 Memory-Related Operations

14.4.1 Load/Store Alternate Address Space (Impdep #5, 29, 30)

Supported ASI accesses are listed in Section 6.3, *Alternate Address Spaces* on page 39.

14.4.2 Load/Store ASR (Impdep #6,7,8,9, 47, 48)

Supported ASRs are listed in Section 6.5, *Ancillary State Registers* on page 51.

14.4.3 MMU Implementation (Impdep #41)

UltraSPARC IIi memory management is based on software-managed instruction and data Translation Lookaside Buffers (TLBs) and in-memory Translation Storage Buffers (TSBs) backed by a Software Translation Table. See Chapter 4, *Overview of I and D-MMUs* for more details.

14.4.4 FLUSH and Self-Modifying Code (Impdep #122)

FLUSH is needed to synchronize code and data spaces after code space is modified during program execution. FLUSH is described in Section 8.3.2, *Memory Synchronization: MEMBAR and FLUSH* on page 70. On UltraSPARC IIi, the FLUSH effective address is translated by the D-MMU. As a result, FLUSH can cause a *data_access_exception* (the page is mapped with side effects or no fault only bits set, virtual address out of range, or privilege violation) or a *data_access_MMU_miss* trap. For a *data_access_exception*, the trap handler can decode the FLUSH instruction, and perform a Done to be consistent with the normal SPARC-V9 behavior of no traps on FLUSH. For a *data_access_MMU_miss*, the trap handler should do the normal TLB miss processing and perform a RETRY if the page can be mapped in the TLB, otherwise perform a DONE.

Note – SPARC-V9 specifies that the FLUSH instruction has no latency on the issuing processor. In other words, a store to instruction space prior to the FLUSH instruction is visible immediately after the completion of FLUSH. MEMBAR `#StoreStore` is required to ensure proper ordering in multi-processing system when the memory model is not TSO. When a MEMBAR `#StoreStore`, FLUSH sequence is performed, UltraSPARC IIi guarantees that earlier code modifications will be visible across the whole system.

14.4.5 PREFETCH{A} (Impdep #103, 117)

For UltraSPARC-I, PREFETCH{A} instructions with *fcn*=0..4 are treated as NOPs.

For UltraSPARC-II, PREFETCH{A} instructions with *fcn*=0..4 have the meanings given in *Table 14-10*.

Table 14-10 PREFETCH{A} Variants (UltraSPARC-II)

fcn	Prefetch Function	Action
0	Prefetch for several reads	Generate P_RDS_REQ if desired line is not present in E-cache
1	Prefetch for one read	
4	Prefetch page	
2	Prefetch for several writes	Generate P_RDO_REQ if desired line is not present in E-cache in either E or M state
3	Prefetch for one write	

PREFETCH{A} instructions with *fcn*=5..15 cause an *illegal_instruction* trap. PREFETCH{A} instructions with *fcn*=16..31 are treated as NOPs.

14.4.6 Non-faulting Load and MMU Disable (Impdep #117)

When the data MMU is disabled, accesses are assumed to be non-cacheable (TTE.PC=0) and with side-effect (TTE.E=1). Non-faulting loads encountered when the MMU is disabled cause a *data_access_exception* trap with SFSR.FT=2 (speculative load to page with side-effect attribute).

14.4.7 LDD/STD Handling (Impdep #107, 108)

LDD and STD instructions are directly executed in hardware.

Note – LDD/STD are deprecated in SPARC-V9. In UltraSPARC IIi it is more efficient to use LDX/STX for accessing 64-bit data. LDD/STD take longer to execute than two 32-/64-bit loads/stores.

14.4.8 FP mem_address_not_aligned (Impdep #109, 110, 111, 112)

LDDF{A}/STDF{A} cause an *LDDF/STDF_ mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned.

LDQF{A}/STQF{A} are not directly executed in hardware; they cause an *illegal_instruction* trap.

14.4.9 Supported Memory Models (Impdep #113, 121)

UltraSPARC IIi supports all three memory models (TSO, PSO, RMO). See Section 20.2, *Supported Memory Models* on page 322.

14.4.10 I/O Operations (Impdep #118, 123)

I/O spaces and their accesses are specified in Section 8.3.7, *I/O (PCI or UPA64S) and Accesses with Side-effects* on page 76.

14.5 Non-SPARC-V9 Extensions

14.5.1 Per-Processor TICK Compare Field of TICK Register

The SPARC-V9 TICK register is used for fine-grain measurements of time in processor cycles. The TICK Compare field (TICK_CMPR) of the TICK Register provides added functionality for thread scheduling on a per-processor basis. Non privileged accesses to this register will cause a *privileged_opcode* trap. See *Table 17-5* on page 261 for a list of resets states.

Table 14-11 TICK_compare Register Format

Bits	Field	Use	RW
<63>	INT_DIS	TICK_INT interrupt enable	RW
<62:0>	TICK_CMPR	Compare value for TICK interrupts	RW

INT_DIS: If set, TICK_INT interrupt generation is disabled.

TICK_CMPR: Writes to the TICK_Compare Register load a value for comparison to the TICK register bits <62:0>. When these values match and (INT_DIS=0) a TICK_INT is posted in the SOFTINT register. This has the effect of posting a level-14 interrupt to the processor when the processor has (PSTATE.PIL $< D_{16}$) and (PSTATE.IE=1). The level-14 interrupt handler must check both SOFTINT<14> and TICK_INT. This function is independent on each processor.

14.5.2 Cache Sub-system

UltraSPARC IIi contains one or more levels of cache. The cache sub-system architecture is described in Chapter 3, *Cache Organization*.

14.5.3 Memory Management Unit

UltraSPARC IIi implements a multi-level memory management scheme. The MMU architecture is described in Chapter 4, *Overview of I and D-MMUs*."

14.5.4 Error Handling

UltraSPARC IIi implements a set of programmer-visible error and exception registers. These registers and their usage are described in Chapter 16, *Error Handling*.

14.5.5 Block Memory Operations

UltraSPARC IIi supports 64-byte block memory operations utilizing a block of eight double-precision floating point registers as a temporary buffer. See Section 13.5.3, *Block Load and Store Instructions* on page 164.

14.5.6 Partial Stores

UltraSPARC IIi supports 8-/16-/32-bit partial stores to memory. See Section 13.5.1, *Partial Store Instructions* on page 161.

14.5.7 Short Floating-Point Loads and Stores

UltraSPARC IIi supports 8-/16-bit loads and stores to the floating-point registers. See Section 13.5.2, *Short Floating-Point Load and Store Instructions* on page 162.

14.5.8 Atomic Quad-load

UltraSPARC IIi supports 128-bit atomic load operations to a pair of integer registers. See Section 13.6.1, *Atomic Quad Load* on page 171.

14.5.9 PSTATE Extensions: Trap Globals

UltraSPARC IIi supports two additional sets of eight 64-bit global registers: interrupt globals and MMU globals. These additional registers are called the "trap globals." Two 1-bit fields, PSTATE.IG and PSTATE.MG, have been added to the PSTATE register to select which set of global registers to use. The PSTATE.IG and PSTATE.MG bits are also stored with the rest of the PSTATE register in the TSTATE register when a trap is taken. See Chapter 11, *Interrupt Handling* for a description of the trap global registers. See *Table 17-5* on page 261 for the states of these bits on reset.

Table 14-12 Extended PSTATE Register

Bits	Field	Use	RW
<11>	IG	Interrupt globals enable	RW
<10>	MG	MMU globals enable	RW
<9>	CLE	Current little endian enable	RW
<8>	TLE	Trap little endian enable	RW
<7:6>	MM	Memory Model	RW
<5>	RED	RED_state enable	RW
<4>	PEF	Floating point enable	RW
<3>	AM	32-bit address mask enable	RW
<2>	PRIV	Privileged mode	RW
<1>	IE	Interrupt enable	RW
<0>	AG	Alternate global enable	RW

Note – Exiting RED_state by writing 0 to PSTATE.RED in the delay slot of a JMPL instruction is not recommended. A noncacheable instruction prefetch may be made to the JMPL target, which may be in a cacheable memory area. This may result in a bus error on some systems, which causes an *instruction_access_error* trap. The trap can be masked by setting the NCEEN bit in the ESTATE_ERR_EN register to zero, but this will mask all non-correctable error checking. Exiting RED_state with DONE or RETRY avoids this problem.

UltraSPARC IIi provides Interrupt and MMU global register sets in addition to the two global register sets specified by SPARC-V9. The currently active set of global registers is specified by the AG, IG and MG bits according to *Table 14-13* on page 194.

Note – The IG and MG fields are saved on the trap stack along with the rest of the PSTATE register.

Table 14-13 PSTATE Global Register Selection Encoding

AG	IG	MG	Globals in Use
0	0	0	Normal
0	0	1	MMU
0	1	0	Interrupt
0	1	1	Reserved
1	0	0	Alternate
1	0	1	Reserved
1	1	0	Reserved
1	1	1	Reserved

When an *interrupt_vector* trap (trap type=60_{16}) is taken, UltraSPARC IIi selects the Interrupt Global registers by setting IG and clearing AG and MG. When a *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, *fast_data_access_protection*, *data_access_exception*, or *instruction_access_exception* trap is taken, UltraSPARC IIi selects the MMU Global Registers by setting MG and clearing AG and IG. When any other type of trap occurs, UltraSPARC IIi selects the Alternate Global Registers by setting AG and clearing IG and MG. Note that global register selection is the same for traps that enter RED_state.

Executing a DONE or RETRY instruction restores the previous {AG, IG, MG} state before the trap is taken. These three bits can also be set or cleared by writing to the PSTATE register with a WRPR instruction.

Note – The AG, IG, and MG bits are mutually exclusive. Attempting to set a reserved encoding using a WRPR to PSTATE generates an *illegal_instruction* trap. UltraSPARC IIi does not check for a reserved encoding in TSTATE. This causes undefined results when a DONE or RETRY is executed.

14.5.10 Interrupt Vector Handling

Processors and I/O devices can interrupt a selected processor by assembling and sending an interrupt packet consisting of three 64-bit interrupt data words. This allows hardware interrupts and cross calls to have the same hardware mechanism and to share a common software interface for processing. Interrupt vectors are described in Chapter 11, *Interrupt Handling*.

14.5.11 Power Down Support and the SHUTDOWN Instruction

UltraSPARC IIi supports power down mode to reduce power requirements during idle periods. A privileged instruction, SHUTDOWN, has been added to facilitate a software-controlled power down of the CPU and system. Power down support and the SHUTDOWN instruction are described in Section 13.6.2, *SHUTDOWN* on page 172.

14.5.12 UltraSPARC IIi Instruction Set Extensions (Impdep #106)

The UltraSPARC IIi CPU extends the standard SPARC-V9 instruction set with three new classes of instructions. These are designed to support power down mode (see Section 13.6.2, *SHUTDOWN* on page 172), enhance graphics functionality (see Section 13.4, *Graphics Instructions*), and improve the efficiency of memory accesses (see Section 13.5, *Memory Access Instructions*).

Unimplemented IMPDEP1 and IMPDEP2 opcodes encountered during execution cause an *illegal_instruction* trap.

14.5.13 Performance Instrumentation

UltraSPARC IIi performance instrumentation is described in Section B.4, *Performance Instrumentation Counter Events* on page 389.

14.5.14 Debug and Diagnostics Support

UltraSPARC IIi support for debug and diagnostics is described in Appendix A, *Debug and Diagnostics Support*.

CHAPTER **15**

MMU Internal Architecture

15.1 Introduction

This chapter provides detailed information about the UltraSPARC IIi Memory Management Unit. It describes the internal architecture of the MMU and how to program it.

15.2 Translation Table Entry (TTE)

The Translation Table Entry, illustrated in *Figure 15-1*, is the UltraSPARC IIi equivalent of a SPARC-V8 page table entry; it holds information for a single page mapping. The TTE is broken into two 64-bit words, representing the tag and data of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB. If there is a hit, the data is fetched by software.

G	—	Context	—	VA_tag<63:22>	Tag
63	62 61	60 48	47 42	41 0	

V	Size	NFO	IE	Soft2	Diag	PA<40:13>	Soft	L	CP	CV	E	P	W	G	Data
63	62 61	60	59	58 50	49 41	40 13	12 7	6	5	4	3	2	1	0	

Figure 15-1 Translation Table Entry (TTE) (from TSB)

G: Global. If the Global bit is set, the Context field of the TTE is ignored during hit detection. This allows any page to be shared among all (user or supervisor) contexts running in the same processor. The Global bit is duplicated in the TTE tag and data to optimize the software miss handler.

Context: The 13-bit context identifier associated with the TTE.

VA_tag<63:22>: Virtual Address Tag. The virtual page number. Bits 21 through 13 are not maintained in the tag, since these bits are used to index the smallest direct-mapped TSB of 64 entries.

Note – Software must sign-extend bits VA_tag<63:44> to form an in-range VA.

V: Valid: If the Valid bit is set, the remaining fields of the TTE are meaningful. Note that the explicit Valid bit is redundant with the software convention of encoding an invalid TTE with an unused context. The encoding of the context field is necessary to cause a failure in the TTE tag comparison, while the explicit Valid bit in the TTE data simplifies the TLB miss handler.

Size: The page size of this entry, encoded as shown in the following table

Table 15-1 Size Field Encoding (from TTE)

Size<1:0>	Page Size
00	8 kB
01	64 kB
10	512 kB
11	4 MB

NFO: No-Fault-Only. If this bit is set, loads with ASI_PRIMARY_NO_FAULT{_LITTLE}, ASI_SECONDARY_NO_FAULT{_LITTLE} are translated. Any other access will trap with a *data_access_exception* trap (FT=10_{16}). The NFO-bit in the I-MMU is read as zero and ignored when written. If this bit is set before loading the TTE into the TLB, the iTLB miss handler should generate an error.

IE: Invert Endianness. If this bit is set, accesses to the associated page are processed with inverse endianness from what is specified by the instruction (big-for-little and little-for-big). See Section 15.6, *ASI Value, Context, and Endianness Selection for Translation* on page 208 for details. In the I-MMU this bit is read as zero and ignored when written.

Note – This bit is intended to be set primarily for noncacheable accesses. The performance of cacheable accesses will be degraded as if the access had missed the D-cache.

Soft<5:0>, Soft2<8:0>: Software-defined fields, provided for use by the operating system. The Soft and Soft2 fields may be written with any value; they read as zero.

Diag: Used by diagnostics to access the redundant information held in the TLB structure. Diag<0>=Used bit, Diag<3:1>=RAM size bits, Diag<6:4>=CAM size bits. (Size bits are 3-bit encoded as 000=8K, 001=64K, 011=512K, 111=4M.) The size bits are read-only; the Used bit is read/write. All other Diag bits are *reserved*.

PA<40:13>: The physical page number. Page offset bits for larger page sizes (PA<15:13>, PA<18:13>, and PA<21:13> for 64 kB, 512 kB, and 4 MB pages, respectively) are stored in the TLB and returned for a Data Access read, but ignored during normal translation.

L: Lock. If this bit is set, the TTE entry will be "locked down" when it is loaded into the TLB; that is, if this entry is valid, it will not be replaced by the automatic replacement algorithm invoked by an ASI store to the Data In register. The lock bit has no meaning for an invalid entry. Arbitrary entries may be locked down in the TLB. Software must ensure that at least one entry is not locked when replacing a TLB entry, otherwise the last TLB entry will be replaced.

CP, CV: The cacheable-in-physically-indexed-cache and cacheable-in-virtually-indexed-cache bits determine the placement of data in UltraSPARC IIi caches, according to *Table 15-2*. The MMU does not operate on the cacheable bits, but merely passes them through to the cache subsystem. The CV-bit in the I-MMU is read as zero and ignored when written.

Table 15-2 Cacheable Field Encoding (from TSB)

Cacheable {CP, CV}	Meaning of TTE When Placed in:	
	iTLB (I-cache PA-Indexed)	dTLB (D-cache VA-Indexed)
0x	Non-cacheable	Non-cacheable
10	Cacheable E-cache, I-cache	Cacheable E-cache only
11	Cacheable E-cache, I-cache	Cacheable E-cache, D-cache

Note – Erratum 58 describes the restricted use of Diag<0>, the Used bit.

E: Side-effect. If this bit is set, speculative loads and FLUSHes will trap for addresses within the page, noncacheable memory accesses other than block loads and stores are strongly ordered against other E-bit accesses, and noncacheable stores are not merged. This bit should be set for pages that map I/O devices having side-effects. Note, however, that the E-bit does not prevent normal instruction prefetching. The E-bit in the I-MMU is read as zero and ignored when written.

Note – The E-bit does not force an uncacheable access. It is expected, but not required, that the CP and CV bits will be set to zero when the E-bit is set.

P: Privileged. If the P bit is set, only the supervisor can access the page mapped by the TTE. If the P bit is set and an access to the page is attempted when PSTATE.PRIV=0, the MMU will signal an *instruction_access_exception* or *data_access_exception* trap (FT=1_{16}).

W: Writable. If the W bit is set, the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted and the MMU will cause a *data_access_protection* trap if a write is attempted. The W-bit in the I-MMU is read as zero and ignored when written.

G: Global. This bit must be identical to the Global bit in the TTE tag. Similar to the case of the Valid bit, the Global bit in the TTE tag is necessary for the TSB hit comparison, while the Global bit in the TTE data facilitates the loading of a TLB entry.

Compatibility Note – Referenced and Modified bits are maintained by software. The Global, Privileged, and Writable fields replace the 3-bit ACC field of the SPARC-V8 Reference MMU Page Translation Entry.

15.3 Translation Storage Buffer (TSB)

The TSB is an array of TTEs managed entirely by software. It serves as a cache of the Software Translation Table, used to quickly reload the TLB in the event of a TLB miss. The discussion in this section assumes the use of the hardware support for TSB access described in Section 15.3.1, *Hardware Support for TSB Access* on page 201, although the operating system is not required to make use of this support hardware.

Inclusion of the TLB entries in the TSB is not required; that is, translation information may exist in the TLB that is not present in the TSB.

The TSB is arranged as a direct-mapped cache of TTEs. The UltraSPARC IIi MMU provides precomputed pointers into the TSB for the 8 kB and 64 kB page TTEs. In each case, *N* least significant bits of the respective virtual page number are used as the offset from the TSB base address, with *N* equal to log base 2 of the number of TTEs in the TSB.

A bit in the TSB register allows the TSB 64 kB pointer to be computed for the case of common or split 8 kB/64 kB TSB(s).

No hardware TSB indexing support is provided for the 512 kB and 4 MB page TTEs. Since the TSB is entirely software managed, however, the operating system may choose to place these larger page TTEs in the TSB by forming the appropriate pointers. In addition, simple modifications to the 8 kB and 64 kB index pointers provided by the hardware allow formation of an M-way set-associative TSB, multiple TSBs per page size, and multiple TSBs per process.

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 level-2 cache resource should provide a better overall solution than that provided by a fixed partitioning.

Figure 15-2 shows both the common and shared TSB organization. The constant *N* is determined by the Size field in the TSB register; it may range from 512 bytes to 64 kB.

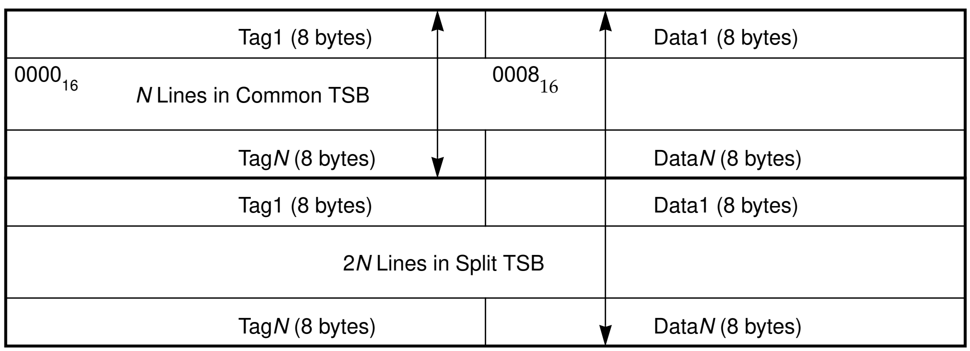


Figure 15-2 TSB Organization

15.3.1 Hardware Support for TSB Access

The MMU hardware provides services to allow the TLB miss handler to efficiently reload a missing TLB entry for an 8 kB or 64 kB page. These services include:

- Formation of TSB Pointers based on the missing virtual address.
- Formation of the TTE Tag Target used for the TSB tag comparison.
- Efficient atomic write of a TLB entry with a single store ASI operation.
- Alternate globals on MMU-signalled traps.

A typical TLB miss and refill sequence is as follows:

1. A TLB miss causes either an *instruction_access_MMU_miss* or a *data_access_MMU_miss* exception.

2. The appropriate TLB miss handler loads the TSB Pointers and the TTE Tag Target with loads from the MMU alternate space.

3. Using this information, the TLB miss handler checks to see if the desired TTE exists in the TSB. If so, the TTE Data is loaded into the TLB Data In register to initiate an atomic write of the TLB entry chosen by the replacement algorithm.

4. If the TTE does not exist in the TSB, the TLB miss handler jumps to a more sophisticated (and slower) TSB miss handler.

The virtual address used in the formation of the pointer addresses comes from the Tag Access register, which holds the virtual address and context of the load or store responsible for the MMU exception. See Section 15.9, *MMU Internal Registers and ASI Operations* on page 213. (Note that there are no separate physical registers in UltraSPARC IIi hardware for the Pointer registers, but rather they are implemented through a dynamic re-ordering of the data stored in the Tag Access and the TSB registers.)

Pointers are provided by hardware for the most common cases of 8 kB and 64 kB page miss processing. These pointers give the virtual addresses where the 8 kB and 64 kB TTEs would be stored if either is present in the TSB.

N is defined to be the TSB_Size field of the TSB register; it ranges from 0 to 7. Note that TSB_Size refers to the size of each TSB when the TSB is split.

For a shared TSB (TSB register split field=0):

```
8K_POINTER = TSB_Base<63:13+N> □ VA<21+N:13> □ 0000
64K_POINTER = TSB_Base<63:13+N> □ VA<24+N:16> □ 0000
```

For a split TSB (TSB register split field=1):

```
8K_POINTER = TSB_Base<63:14+N> □ 0 □ VA<21+N:13> □ 0000
64K_POINTER = TSB_Base<63:14+N> □ 1 □ VA<24+N:16> □ 0000
```

For a more detailed description of the pointer logic with pseudo-code and hardware implementation, see Section 15.11.3, *TSB Pointer Logic Hardware Description* on page 228.

The TSB Tag Target (described in Section 15.9, *MMU Internal Registers and ASI Operations* on page 213) is formed by aligning the missing access VA (from the Tag Access register) and the current context to positions found in the description of the TTE tag. This allows an XOR instruction for TSB hit detection.

These items must be locked in the TLB to avoid an error condition: TLB-miss handler, TSB and linked data, asynchronous trap handlers and data.

These items must be locked in the TSB (not necessarily the TLB) to avoid an error condition: TSB-miss handler and data, interrupt-vector handler and data.

15.3.2 Alternate Global Selection During TLB Misses

In the SPARC-V9 normal trap mode, the software is presented with an alternate set of global registers in the integer register file. UltraSPARC IIi provides an additional feature to facilitate fast handling of TLB misses. For the following traps, the trap handler is presented with a special set of MMU globals: *fast_{instruction,data}_access_MMU_miss*, *{instruction,data}_access_exception*, and *fast_data_access_protection*. The *privileged_action* and **mem_address_not_aligned* traps use the normal alternate global registers.

Compatibility Note – The UltraSPARC IIi MMU performs no hardware table walking. The MMU hardware never directly reads or writes to the TSB.

15.4 MMU-Related Faults and Traps

Table 15-3 lists the traps recorded by the MMU.

Table 15-3 MMU Traps

Trap Name	Trap Cause	Registers Updated (Stored State in MMU)			
		I-SFSR	I-Tag Access	D-SFSR, SFAR	D-Tag Access
fast_instruction_access_MMU_miss	iTLB miss		✓		
instruction_access_exception	Several (see below)	✓	✓[1]		
fast_data_access_MMU_miss	dTLB miss				✓
data_access_exception	Several (see below)			✓	✓
fast_data_access_protection	Protection violation			✓	✓
privileged_action	Use of privileged ASI			✓	
*_watchpoint	Watchpoint hit			✓	
*_mem_address_not_aligned	Misaligned mem op			✓	

[1]Contents undefined if *instruction_access_exception* is due to virtual address out of range.

Note – The *fast_instruction_access_MMU_miss*, *fast_data_access_MMU_miss*, and *fast_data_access_protection* traps are generated instead of *instruction_access_MMU_miss*, *data_access_MMU_miss*, and *data_access_protection* traps, respectively.

15.4.1 Instruction_access_MMU_miss Trap

This trap occurs when the I-MMU is unable to find a translation for an instruction access; that is, when the appropriate TTE is not in the iTLB.

15.4.2 Instruction_access_exception Trap

This trap occurs when the I-MMU is enabled and one of the following happens:

- The I-MMU detects a privilege violation for an instruction fetch; that is, an attempted access to a privileged page when PSTATE.PRIV=0.
- Virtual address out of range and PSTATE.AM is not set. See Section 14.1.7, *44-bit Virtual Address Space* on page 178. Note that the case of JMPL/RETURN and branch-CALL-sequential are handled differently. The contents of the I-Tag Access Register are undefined in this case, but are not needed by software.

15.4.3 Data_access_MMU_miss Trap

This trap occurs when the MMU is unable to find a translation for a data access; that is, when the appropriate TTE is not in the data TLB for a memory operation.

15.4.4 Data_access_exception Trap

This trap occurs when the D-MMU is enabled and one of the following events (the D-MMU does not prioritize these) occurs.

- The D-MMU detects a privilege violation for a data or FLUSH instruction access; that is, an attempted access to a privileged page when PSTATE.PRIV=0
- A speculative (non-faulting) load or FLUSH instruction issued to a page marked with the side-effect (E-bit)=1
- An atomic instruction (including 128-bit atomic load) issued to a memory address marked uncacheable in a physical cache; that is, with CP=0

- An invalid LDA/STA ASI value, invalid virtual address, read to write-only register, or write to read-only register, but not for an attempted user access to a restricted ASI (see the *privileged_action* trap described below)
- An access (including FLUSH) with an ASI other than ASI_{PRIMARY,SECONDARY}_NO_FAULT{_LITTLE} to a page marked with the NFO (no-fault-only) bit
- Virtual address out of range (including FLUSH) and PSTATE.AM is not set. See Section 4.2, *Virtual Address Translation* on page 23

The *data_access_exception* trap also occurs when the D-MMU is disabled and one the following occurs.

- Speculative (non-faulting) load or FLUSH instruction issued when LSU_Control_Register.DP=0
- An atomic instruction (including 128-bit atomic load) is issued using the ASI_PHYS_BYPASS_EC_WITH_EBIT{_LITTLE} ASIs. In this case SFSR.FT=04_{16}

15.4.5 Data_access_protection Trap

This trap occurs when the MMU detects a protection violation for a data access. A protection violation is defined to be an attempted store to a page without write permission.

15.4.6 Privileged_action Trap

This trap occurs when an access is attempted using a *restricted* ASI while in non-privileged mode (PSTATE.PRIV=0).

15.4.7 Watchpoint Trap

This trap occurs when watchpoints are enabled and the D-MMU detects a load or store to the virtual or physical address specified by the VA Data Watchpoint Register or the PA Data Watchpoint Register, respectively. See Section A.5, *Watchpoint Support* on page 368.

15.4.8 Mem_address_not_aligned Trap

This trap occurs when a load, store, atomic, or JMPL/RETURN instruction with a misaligned address is executed. The LSU signals this trap, but the D-MMU records the fault information in the SFSR and SFAR.

15.5 MMU Operation Summary

Table 15-6 on page 208 summarizes the behavior of the D-MMU; *Table 15-6* on page 208 summarizes the behavior of the I-MMU for normal (non-UltraSPARC IIi-internal) ASIs using tabulated abbreviations. In each case, and for all conditions, the behavior of the MMU is given by one of the abbreviations in *Table 15-4*. *Table 15-5* lists abbreviations for ASI types.

Table 15-4 Abbreviations for MMU Behavior

Abbreviation	Meaning
ok	Normal Translation
dmiss	*data_access_MMU_miss* trap
dexc	*data_access_exception* trap
dprot	*data_access_protection* trap
imiss	*instruction_access_MMU_miss* trap
iexc	*instruction_access_exception* trap

Table 15-5 Abbreviations for ASI Types

Abbreviation	Meaning
NUC	ASI_NUCLEUS*
PRIM	Any ASI with PRIMARY translation, except *NO_FAULT"
SEC	Any ASI with SECONDARY translation, except *NO_FAULT"
PRIM_NF	ASI_PRIMARY_NO_FAULT*
SEC_NF	ASI_SECONDARY_NO_FAULT*
U_PRIM	ASI_AS_IF_USER_PRIMARY*
U_SEC	ASI_AS_IF_USER_SECONDARY*
BYPASS	ASI_PHYS_* and also other ASIs that require the MMU to perform a bypass operation (such as D-cache access)

Note – The "*_LITTLE" versions of the ASIs behave the same as the big-endian versions with regard to the MMU table of operations.

Other abbreviations include “W” for the writable bit, “E” for the side-effect bit, and “P” for the privileged bit.

The tables do not cover the following cases:

- Invalid ASIs, ASIs that have no meaning for the opcodes listed, or non-existent ASIs; for example, ASI_PRIMARY_NO_FAULT for a store or atomic; also, access to UltraSPARC IIi internal registers other than LDXA, LDFA, STDFA or STXA, except for I-cache diagnostic accesses other than LDDA, STDFA or STXA; see Section 6.3.2, *UltraSPARC IIi (Non-SPARC-V9) ASI Extensions* on page 41; the MMU signals a *data_access_exception* trap (FT=08_{16}) for this case
- Attempted access using a restricted ASI in non-privileged mode; the MMU signals a *privileged_action* exception for this case
- An atomic instruction (including 128-bit atomic load) issued to a memory address marked uncacheable in a physical cache (that is, with CP=0), including cases in which the D-MMU is disabled; the MMU signals a *data_access_exception* trap (FT=04_{16}) for this case
- A data access (including FLUSH) with an ASI other than ASI_{PRIMARY,SECONDARY}_NO_FAULT{_LITTLE} to a page marked with the NFO (no-fault-only) bit; the MMU signals a *data_access_exception* trap (FT=10_{16}) for this case
- Virtual address out of range (including FLUSH) and PSTATE.AM is not set; the MMU signals a *data_access_exception* trap (FT=20_{16}) for this case

Table 15-6 D-MMU Operations for Normal ASIs

Condition				Behavior				
Opcode	**PRIV Mode**	**ASI**	**W**	**TLB Miss**	**E=0 P=0**	**E=0 P=1**	**E=1 P=0**	**E=1 P=1**
Load	0	PRIM, SEC	—	dmiss	ok	dexc	ok	dexc
		PRIM_NF, SEC_NF	—	dmiss	ok	dexc	dexc	dexc
	1	PRIM, SEC, NUC	—	dmiss	ok		ok	
		PRIM_NF, SEC_NF	—	dmiss	ok		dexc	
		U_PRIM, U_SEC	—	dmiss	ok	dexc	ok	dexc
FLUSH	0		—	dmiss	ok	dexc	dexc	dexc
	1		—	dmiss	ok	ok	dexc	dexc
Store or Atomic	0	PRIM, SEC	0	dmiss	dprot	dexc	dprot	dexc
			1	dmiss	ok	dexc	ok	dexc
	1	PRIM, SEC, NUC	0	dmiss	dprot		dprot	
			1	dmiss	ok		ok	
		U_PRIM, U_SEC	0	dmiss	dprot	dexc	dprot	dexc
			1	dmiss	ok	dexc	ok	dexc
—	0	BYPASS	—	privileged_action				
—	1	BYPASS	—	Bypass. No traps when D-MMU enabled, PRIV=1.				

Table 15-7 I-MMU Operations for Normal ASIs

Condition	Behavior		
PRIV Mode	**TLB Miss**	**P=0**	**P=1**
0	imiss	ok	iexc
1	imiss	ok	

See Section 6.3, *Alternate Address Spaces* on page 39 for a summary of the UltraSPARC IIi ASI map.

15.6 ASI Value, Context, and Endianness Selection for Translation

The MMU uses a two-step process to select the context for a translation:

1. The ASI is determined (conceptually by the Integer Unit) from the instruction, trap level, and the processor endian mode

2. The context register is determined directly from the ASI.

The ASI value and endianness (little or big) are determined for the I-MMU and D-MMU respectively according to *Table 15-8* and *Table 15-9* on page 210.

Note – The secondary context is never used to fetch instructions. The I-MMU uses the value stored in the D-MMU Primary Context register when using the Primary Context identifier; there is no I-MMU Primary Context register.

Note – The endianness of a data access is specified by three conditions: the ASI specified in the opcode or ASI register, the PSTATE current little endian bit, and the D-MMU invert endianness bit. The D-MMU invert endianness bit does not affect the ASI value recorded in the SFSR, but does invert the endianness that is otherwise specified for the access.

Note – The D-MMU Invert Endianness (IE) bit inverts the endianness for all accesses to translating ASIs, including LD/ST/Atomic alternates that have specified an ASI. That is, `LDXA [%g1]ASI_PRIMARY_LITTLE` will be big-endian if the IE bit is on. Accesses to non-translating ASIs are not affected by the D-MMUs IE bit. See Section 6.3, *Alternate Address Spaces* on page 39 for information about non-translating ASIs

---

Table 15-8 ASI Mapping for Instruction Accesses

Condition for Instruction Access	Resulting Action	
PSTATE.TL	**Endianness**	**ASI Value (in SFSR)**
0	Big	ASI_PRIMARY
> 0	Big	ASI_NUCLEUS

Table 15-9 ASI Mapping for Data Accesses

Condition for Data Access				Access Processed with:	
Opcode	**PSTATE. TL**	**PSTATE. CLE**	**D-MMU. IE**	**Endianness**	**ASI Value (Recorded in SFSR)**
LD/ST/Atomic/FLUSH	0	0	0	Big	ASI_PRIMARY
			1	Little	
		1	0	Little	ASI_PRIMARY_LITTLE
			1	Big	
	> 0	0	0	Big	ASI_NUCLEUS
			1	Little	
		1	0	Little	ASI_NUCLEUS_LITTLE
			1	Big	
LD/ST/Atomic Alternate with specified ASI *not* ending in “_LITTLE”	Don’t care	Don’t care	0	Big[1]	Specified ASI value from immediate field in opcode or ASI register
			1	Little[1]	
LD/ST/Atomic Alternate with specified ASI ending in ‘_LITTLE”	Don’t care	Don’t care	0	Little	Specified ASI value from immediate field in opcode or ASI register
			1	Big	

1 Accesses to non-translating ASIs are always made in “big endian” mode, regardless of the setting of D-MMU.IE. See Section 6.3, *Alternate Address Spaces* on page 39 for information about non-translating ASIs.

The context register used by the data and instruction MMUs is determined from the following table. A comprehensive list of ASI values can be found in the ASI map in Section 6.3, *Alternate Address Spaces* on page 39. The context register selection is not affected by the endianness of the access.

Table 15-10 I-MMU and D-MMU Context Register Usage

ASI Value	Context Register
ASI_*NUCLEUS*[1]	Nucleus (0000_{16} hard-wired)
ASI_*PRIMARY*[2]	Primary
ASI_*SECONDARY*[3]	Secondary
All other ASI values	(Not applicable, no translation)

1. Any ASI name containing the string “NUCLEUS”.
2. Any ASI name containing the string “PRIMARY”.
3. Any ASI name containing the string “SECONDARY”.

15.7 MMU Behavior During Reset, MMU Disable, and RED_state

During global reset of the UltraSPARC IIi CPU, the following actions occur:

- No change occurs in any block of the D-MMU.
- No change occurs in the data path or TLB blocks of the I-MMU.
- The I-MMU resets its internal state machine to normal (non-suspended) operation.
- The I-MMU and D-MMU Enable bits in the LSU Control Register (see Section A.6, *LSU_Control_Register* on page 370) are set to zero.

On entering RED_state, the I-MMU and D-MMU Enable bits in the LSU_Control_Register are set to zero.

Either MMU is defined to be disabled when its respective MMU Enable bit equals 0; also, the I-MMU is disabled whenever the CPU is in RED_state. The D-MMU is enabled or disabled solely by the state of the D-MMU Enable bit.

When the D-MMU is disabled it truncates all accesses, behaving as if ASI_PHYS_BYPASS_EC_WITH_EBIT had been used, notably with side effect bit (E-bit)=1, P=0 and CP=0. Other attribute bit settings can be found in Section 15.10, *MMU Bypass Mode* on page 226. However, if a bypass ASI is used while the D-MMU is disabled, the bypass operation behaves as it does when the D-MMU is enabled; that is, the access is processed with the E and CP bits as specified by the bypass ASI.

When the I-MMU is disabled, it truncates all instruction accesses and passes the physically-cacheable bit (CP=0) to the cache system. The access will not generate an *instruction_access_exception* trap.

When disabled, both the I-MMU and D-MMU correctly perform all LDXA and STXA operations to internal registers, and traps are signalled just as if the MMU were enabled. For instance, if a *NO_FAULT load is issued when the D-MMU is disabled, the D-MMU signals a *data_access_exception* trap (FT=02_{16}), since accesses when the D-MMU is disabled have E=1.

Note – While the D-MMU is disabled, data in the D-cache can be accessed only using load and store alternates to the UltraSPARC IIi internal D-cache access ASI. Normal loads and stores bypass the D-cache. Data in the D-cache cannot be accessed using load or store alternates that use ASI_PHYS_*.

Note – No reset of the MMU is performed by a chip reset or by entering RED_state. Before the MMUs are enabled, the operating system software must explicitly write each entry with either a valid TLB entry or an entry with the valid bit set to zero. The operation of the I-MMU or D-MMU in enabled mode is undefined if the TLB valid bits have not been set explicitly beforehand.

15.8 Compliance with the SPARC-V9 Annex F

The UltraSPARC IIi MMU complies completely with the SPARC-V9 MMU Requirements described in Annex F of the *The SPARC Architecture Manual, Version 9*. *Table 15-11* shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the I- or D-MMU. Note that this behavior requires specialized TLB miss handler code to guarantee these conditions.

Table 15-11 MMU Compliance w/SPARC-V9 Annex F Protection Mode

Condition			Resultant Protection Mode
TTE in D-MMU	**TTE in I-MMU**	**Writable Attribute Bit**	
Yes	No	0	Read-only
No	Yes	Don't Care	Execute-only
Yes	No	1	Read/Write
Yes	Yes	0	Read-only/Execute
Yes	Yes	1	Read/Write/Execute

15.9 MMU Internal Registers and ASI Operations

15.9.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the CPU through UltraSPARC IIi-defined ASIs. Several of the registers have been assigned their own ASI because these registers are crucial to the speed of the TLB miss handler. Allowing the use of `%g0` for the address reduces the number of instructions to perform the access to the alternate space (by eliminating address formation).

See Section 15.10, *MMU Bypass Mode* on page 226 for details on the behavior of the MMU during all other UltraSPARC IIi ASI accesses. For instance, to facilitate an access to the D-cache, the MMU performs a bypass operation.

Caution – STXA to an MMU register requires either a MEMBAR `#Sync`, FLUSH, DONE, or RETRY before the point that the effect must be visible to load / store / atomic accesses. Either a FLUSH, DONE, or RETRY is needed before the point that the effect must be visible to instruction accesses: MEMBAR `#Sync` is not sufficient. In either case, one of these instructions must be executed before the next non-internal store or load of any type and on or before the delay slot of a DCTI of any type. This is necessary to avoid corrupting data.

If the low order three bits of the VA are non-zero in a LDXA/STXA to/from these registers, a *mem_address_not_aligned* trap occurs. Writes to read-only, reads to write-only, illegal ASI values, or illegal VA for a given ASI may cause a *data_access_exception* trap (FT=08_{16}). (The hardware detects VA violations in only an unspecified lower portion of the virtual address.)

Caution – UltraSPARC IIi does not check for out-of-range virtual addresses during an STXA to any internal register; it simply sign extends the virtual address based on VA<43>. Software must guarantee that the VA is within range.

Writes to the TSB register, Tag Access register, and PA and VA Watchpoint Address Registers are not checked for out-of-range VA. No matter what is written to the register, VA<63:43> will always be identical on a read.

Table 15-12 UltraSPARC IIi MMU Internal Registers and ASI Operations

I-MMU ASI	D-MMU ASI	VA<63:0>	Access	Register or Operation Name
50_{16}	58_{16}	0_{16}	Read-only	I-/D-TSB Tag Target Registers
—	58_{16}	8_{16}	Read/Write	Primary Context Register
—	58_{16}	10_{16}	Read/Write	Secondary Context Register
50_{16}	58_{16}	18_{16}	Read/Write	I-/D-Synchronous Fault Status Registers
—	58_{16}	20_{16}	Read-only	D Synchronous Fault Address Register
50_{16}	58_{16}	28_{16}	Read/Write	I-/D-TSB Registers
50_{16}	58_{16}	30_{16}	Read/Write	I-/D-TLB Tag Access Registers
—	58_{16}	38_{16}	Read/Write	Virtual Watchpoint Address
—	58_{16}	40_{16}	Read/Write	Physical Watchpoint Address
51_{16}	59_{16}	0_{16}	Read-only	I-/D-TSB 8K Pointer Registers
52_{16}	$5A_{16}$	0_{16}	Read-only	I-/D-TSB 64K Pointer Registers
—	$5B_{16}$	0_{16}	Read-only	D-TSB Direct Pointer Register
54_{16}	$5C_{16}$	0_{16}	Write-only	I-/D-TLB Data In Registers
55_{16}	$5D_{16}$	$0_{16}..1F8_{16}$	Read/Write	I-/D-TLB Data Access Registers
56_{16}	$5E_{16}$	$0_{16}..1F8_{16}$	Read-only	I-/D-TLB Tag Read Register
57_{16}	5F	See 15.9.10	Write-only	I-/D-MMU Demap Operation

15.9.2 I-/D-TSB Tag Target Registers

The I- and D-TSB Tag Target registers are simply respective bit-shifted versions of the data stored in the I- and D-Tag Access registers. Since the I- or D-Tag Access registers are updated on I- or D-TLB misses, respectively, the I- and D-Tag Target registers appear to software to be updated on an I or D TLB miss.

000	Context	—	VA<63:22>
63 61	60 48	47 42	41 0

Figure 15-3 MMU Tag Target Registers (Two Registers)

I/D Context<12:0>: The context associated with the missing virtual address.

I/D VA<63:22>: The most significant bits of the missing virtual address.

15.9.3 Context Registers

The context registers are shared by the I- and D-MMUs. The Primary Context Register is defined as shown in *Figure 15-4*

—	PContext
63 — 13	12 — 0

Figure 15-4 D-MMU Primary Context Register

PContext: Context identifier for the primary address space.

The Secondary Context register is defined in *Figure 15-6*

—	SContext
63 — 13	12 — 0

Figure 15-5 D-MMU Secondary Context Register

SContext: Context identifier for the secondary address space.

The Nucleus Context register is hardwired to zero:

0000000000000000000000000000000000000000000000000000000000000000
63 — 0

Figure 15-6 D-MMU Nucleus Context Register

Compatibility Note – The single context register of the SPARC-V8 Reference MMU has been replaced in UltraSPARC IIi by the three context registers shown in Figures 15-4, 15-5, and 15-6.

Note – A STXA to the context registers requires either a MEMBAR `#Sync`, FLUSH, DONE, or RETRY before the point that the effect must be visible to data accesses. Either a FLUSH, DONE, or RETRY is needed before the point that the effect must be visible to instruction accesses: MEMBAR `#Sync` is not sufficient. In either case, one of these instructions must be executed before the next translating or bypass store or load of any type. This is necessary to avoid corrupting data.

15.9.4 I-/D-MMU Synchronous Fault Status Registers (SFSR)

The I- and D-MMU each maintain their own SFSR register, which is defined as follows:

—	ASI	—	FT	E	CT	PR	W	OW	FV
63 24	23 16	15 14	13 7	6	5 4	3	2	1	0

Figure 15-7 I- and D-MMU Synchronous Fault Status Register Format

ASI: The ASI field records the 8-bit ASI associated with the faulting instruction. This field is valid for both D-MMU and I-MMU SFSRs and for all traps in which the FV bit is set. JMPL and RETURN *mem_address_not_aligned* traps set the default ASI, as does a trapping non-alternate load or store; that is, to ASI_PRIMARY for PSTATE.CLE=0, or to ASI_PRIMARY_LITTLE otherwise.

FT: The Fault Type field indicates the exact condition that caused the recorded fault, according to *Table 15-13*. In the D-MMU the Fault Type field is valid only for *data_access_exception* traps; there is no ambiguity in all other MMU trap cases. Note that the hardware does not priority-encode the bits set in the fault type register; that is, multiple bits may be set. The FT field in the D-MMU SFSR reads zero for traps other than *data_access_exception*. The FT field in the I-MMU SFSR always reads zero for *instruction_access_MMU_miss*, and either 01_{16}, 20_{16}, or 40_{16} for *instruction_access_exception*, as all other fault types do not apply.

Table 15-13 MMU Synchronous Fault Status Register FT (Fault Type) Field

FT<6:0>	Fault Type
01_{16}	Privilege violation
02_{16}	Speculative Load or Flush instruction to page marked with E-bit. This bit is zero for internal ASI accesses.
04_{16}	Atomic (including 128-bit atomic load) to page marked uncacheable. This bit is zero for internal ASI accesses, except for atomics to DTLB_DATA_ACCESS_REG ($5D_{16}$), or DTLB_DATA_IN_REG ($5C_{16}$), or DTLB_TAG_READ_REG ($5E_{16}$) which update according to the TLB entry accessed.
08_{16}	Illegal LDA/STA ASI value, VA, RW, or size. Excludes cases where 02_{16} and 04_{16} are set.
10_{16}	Access other than non-faulting load to page marked NFO. This bit is zero for internal ASI accesses.
20_{16}	VA out of range (D-MMU and I-MMU branch, CALL, sequential)
40_{16}	VA out of range (I-MMU JMPL or RETURN)

E: reports the side-effect bit (E) associated with the faulting data access or FLUSH instruction; set by FLUSH or translating ASI accesses (see Section 6.3, *Alternate Address Spaces* on page 39) mapped by the TLB with the E bit set and ASI_PHYS_BYPASS_EC_WITH_EBIT{_LITTLE} ASIs (15_{16} and $1D_{16}$). Other cases that update the SFSR (including bypass or internal ASI accesses) set the E bit to 0. It always reads as 0 in the I-MMU.

CT: Context register selection, as described in the following table; the context is set to 11_2 when the access does not have a translating ASI (see Section 6.3, *Alternate Address Spaces* on page 39).

Table 15-14 MMU SFSR Context ID Field Description

Context ID	I-MMU Context	D-MMU Context
00	Primary	Primary
01	Reserved	Secondary
10	Nucleus	Nucleus
11	Reserved	Reserved

PR: Privilege; set if the faulting access occurred while in Privileged mode; this field is valid for all traps in which the Fault Valid (FV) bit is set

W: Write; set if the faulting access indicated a data write operation (a store or atomic load/store instruction); always reads as 0 in the I-MMU SFSR

OW: Overwrite; set to one when the MMU detects a fault, if the Fault Valid bit has not been cleared from a previous fault; otherwise, it is set to zero

FV: Fault Valid; set when the MMU detects a fault; cleared only on an explicit ASI write of 0 to the SFSR register; when FV is not set, the values of the remaining fields in the SFSR and SFAR are undefined

The SFSR and the Tag Access registers both maintain state concerning a previous translation causing an exception. The update policy for the SFSR and the Tag Access registers is shown in *Table 15-6* on page 208.

Note – A *fast_{instruction,data}_access_MMU_miss* trap does not cause the SFSR or SFAR to be written. In this case the D-SFAR information can be obtained from the D Tag Access register.

15.9.5 I-/D-MMU Synchronous Fault Address Registers (SFAR)

15.9.5.1 I-MMU Fault Address

There is no I-MMU Synchronous Fault Address register. Instead, software must read the TPC register appropriately as discussed here.

For *instruction_access_MMU_miss* traps, TPC contains the virtual address that was not found in the I-MMU TLB.

For *instruction_access_exception* traps, "privilege violation" fault type, TPC contains the virtual address of the instruction in the privileged page that caused the exception.

For *instruction_access_exception* traps, "VA out of range" fault types, note that the TPC in these cases contains only a 44-bit virtual address, which is sign-extended based on bit VA<43> for read. Therefore, use the following methods to compute the virtual address that was out of range:

- For the branch, CALL, and sequential exception case, the TPC contains the lower 44 bits of the virtual address that is out of range. Because the hardware sign-extends a read of the TPC register based on VA<43>, the contents of the TPC register XORd with FFFF F000 0000 0000_{16} will give the full 64-bit out-of-range virtual address.
- For the JMPL or RETURN exception case, the TPC contains the virtual address of the JMPL or RETURN instruction itself. Software must disassemble the instruction to compute the out-of-range virtual address of the target.

15.9.5.2 D-MMU Fault Address

The Synchronous Fault Address register contains the virtual memory address of the fault recorded in the D-MMU Synchronous Fault Status register. There is no I-SFAR, since the instruction fault address is found in the trap program counter (TPC). The SFAR can be considered an additional field of the D-SFSR.

Figure 15-8 illustrates the D-SFAR.

Fault Address (VA<63:0>)
63 … 0

Figure 15-8 D-MMU Synchronous Fault Address Register (SFAR) Format

Fault Address: is the virtual address associated with the translation fault recorded in the D-SFSR. this field is valid only when the D-SFSR Fault Valid (FV) bit is set. This field is sign-extended based on VA<43>, so bits VA<63:44> do not correspond to the virtual address used in the translation for the case of a VA-out-of-range *data_access_exception* trap (for this case, software must disassemble the trapping instruction).

15.9.6 I- and D- Translation Storage Buffer (TSB) Registers

The TSB registers provide information for the hardware formation of TSB pointers and tag target, to assist software in handling TLB misses quickly. If the TSB concept is not employed in the software memory management strategy, and therefore the pointer and tag access registers are not used, then the TSB registers need not contain valid data.

Figure 15-9 illustrates the TSB register.

TSB_Base<63:13> (virtual)	Split	—	TSB_Size
63 13	12	11 3	2 0

Figure 15-9 I-TSB and D-TSB Register Format

I/D TSB_Base<63:13>: provides the base virtual address of the Translation Storage Buffer. Software must ensure that the TSB Base is aligned on a boundary equal to the size of the TSB, or both TSBs in the case of a split TSB.

Caution – Stores to the TSB registers are not checked for out-of-range violations. Reads from these registers are sign-extended based on TSB_Base<43>.

Split: When Split=1, the TSB 64 kB Pointer address is calculated assuming separate (but abutting and equally-sized) TSB regions for the 8 kB and the 64 kB TTEs. In this case, TSB_Size refers to the size of each TSB, and therefore the TSB 8 kB Pointer address calculation is not affected by the value of the Split bit. When Split=0, the TSB 64 kB Pointer address is calculated assuming that the same lines in the TSB are shared by 8 kB and 64 kB TTEs, called a “common TSB” configuration.

Caution – In the “common TSB” configuration (TSB.Split=0), 8 kB and 64 kB page TTEs can conflict, unless the TLB miss handler explicitly checks the TTE for page size. Therefore, do not use the common TSB mode in an optimized handler. For example, suppose an 8K page at VA=2000_{16} and a 64K page at VA=10000_{16} both exist, which is a legal situation. These both want to exist at the second TSB line (line 1), and have the same VA tag of 0. Therefore, there is no way for the miss handler to distinguish these TTEs based on the TTE tag alone, and unless it reads the TTE data, it may load an incorrect TTE.

I/D TSB_Size: The Size field provides the size of the TSB according to the following:

- Number of entries in the TSB (or each TSB if split)=512×2^{TSB_Size}.
- Number of entries in the TSB ranges from 512 entries at TSB_Size=0 (8 kB common TSB, 16 kB split TSB), to 64 kB entries at TSB_Size=7 (1 MB common TSB, 2 MB split TSB).

Note – Any update to the TSB register immediately affects the data that is returned from later reads of the Tag Target and TSB Pointer registers.

15.9.7 I-/D-TLB Tag Access Registers

In each MMU the Tag Access register is used as a temporary buffer for writing the TLB Entry tag information. The Tag Access register may be updated during either of the following operations:

1. When the MMU signals a trap due to a miss, exception, or protection. The MMU hardware automatically writes the missing VA and the appropriate Context into the Tag Access register to facilitate formation of the TSB Tag Target register. See *Table 15-6* on page 208 for the SFSR and Tag Access register update policy.
2. An ASI write to the Tag Access register. Before an ASI store to the TLB Data Access registers, the operating system must set the Tag Access register to the values desired in the TLB Entry. Note that an ASI store to the TLB Data In register for automatic replacement also uses the Tag Access register, but typically the value written into the Tag Access register by the MMU hardware is appropriate.

Note – Any update to the Tag Access registers immediately affects the data that is returned from subsequent reads of the Tag Target and TSB Pointer registers.

The TLB Tag Access Registers are defined *Figure 15-10*

VA<63:13>	Context<12:0>
63 … 13	12 … 0

Figure 15-10 I/D MMU TLB Tag Access Registers

I/D VA<63:13>: The 51-bit virtual page number. Note that writes to this field are not checked for out-of-range violation, but sign extended based on VA<43>.

Caution – Stores to the Tag Access registers are not checked for out-of-range violations. Reads from these registers are sign-extended based on VA<43>.

I/D Context<12:0>: is the 13-bit context identifier. This field reads zero when there is no associated context with the access.

15.9.8 I-TSB and D-TSB 8 kB/64 kB Pointer and Direct Pointer Registers

These registers are provided to help the software determine the location of the missing or trapping TTE in the software-maintained TSB. The TSB 8 kB and 64 kB Pointer registers provide the possible locations of the 8 kB and 64 kB TTE, respectively. The Direct Pointer register is mapped by hardware to either the 8 kB or 64 kB Pointer register in the case of a *fast_data_access_protection* exception according to the known size of the trapping TTE. In the case of a 512 kB or 4 MB page miss, the Direct Pointer register returns the pointer as if the miss were from an 8 kB page.

The TSB Pointer registers are implemented as a re-order of the current data stored in the Tag Access register and the TSB register. If the Tag Access register or TSB register is updated through a direct software write (via a STXA instruction), then the Pointer registers values will be updated as well.

The bit that controls selection of 8K or 64K address formation for the Direct Pointer register is a state bit in the D-MMU that is updated during a *data_access_protection* exception. It records whether the page that hit in the TLB was an 64K page or a non-64K page, in which case 8K is assumed.

The I-/D-TSB 8 kB/64 kB Pointer registers are defined as follows:

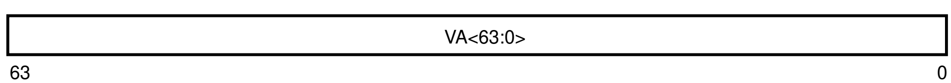


Figure 15-11 I-MMU and D-MMU TSB 8 kB/64 kB Pointer and D-MMU Direct Pointer Register

VA<63:0>: is the full virtual address of the TTE in the TSB, as determined by the MMU hardware. Described in Section 15.3.1, *Hardware Support for TSB Access* on page 201. Note that this field is sign-extended based on VA<43>.

15.9.9 I-TLB and D-TLB Data-In/Data-Access/Tag-Read Registers

Access to the TLB is complicated due to the need to provide an atomic write of a TLB entry data item (tag and data) that is larger than 64 bits, the need to replace entries automatically through the TLB entry replacement algorithm as well as provide direct diagnostic access, and the need for hardware assist in the TLB miss handler. *Table 15-15* shows the effect of loads and stores on the Tag Access register and the TLB.

Table 15-15 Effect of Loads and Stores on MMU Registers

Software Operation		Effect on MMU Physical Registers		
Load/Store	**Register**	**TLB tag**	**TLB data**	**Tag Access Register**
Load	Tag Read	No effect. Contents returned	No effect	No effect
	Tag Access	No effect	No effect	No effect. Contents returned
	Data In	Trap with *data_access_exception*		
	Data Access	No effect	No effect. Contents returned	No effect
Store	Tag Read	Trap with *data_access_exception*		
	Tag Access	No effect	No effect	Written with store data
	Data In	TLB entry determined by replacement policy written with contents of Tag Access Register	TLB entry determined by replacement policy written with store data	No effect
	Data Access	TLB entry specified by STXA address written with contents of Tag Access Register	TLB entry specified by STXA address written with store data	No effect
TLB miss		No effect	No effect	Written with VA and context of access

The Data In and Data Access registers are the means of reading and writing the TLB for all operations. The TLB Data In register is used for TLB-miss and TSB-miss handler automatic replacement writes; the TLB Data Access register is used for operating system and diagnostic directed writes (writes to a specific TLB entry). Both types of registers have the same format, as follows:

V	Size	NFO	IE	Soft2	Diag	PA<40:13>	Soft	L	CP	CV	E	P	W	G
63	62 61	60	59	58 50	49 41	40 13	12 7	6	5	4	3	2	1	0

Figure 15-12 MMU I-/D-TLB Data In/Access Registers

Refer to the description of the TTE data in Section 15.2, *Translation Table Entry (TTE)* on page 197, for a complete description of the above data fields.

Operations to the TLB Data In register require the virtual address to be set to zero. The format of the TLB Data Access register virtual address is as follows:

—	TLB Entry	000
63 … 9	8 … 3	2 … 0

Figure 15-13 MMU TLB Data Access Address, in Alternate Space

TLB Entry: The TLB Entry number to be accessed, in the range 0..63.

The format for the Tag Read register is as follows:

VA<63:13>	Context<12:0>
63 … 13	12 … 0

Figure 15-14 I-/D-MMU TLB Tag Read Registers

I/D VA<63:13>: is the 51-bit virtual page number. Page offset bits for larger page sizes are stored in the TLB and returned for a Tag Read register read, but ignored during normal translation; that is, VA<15:13>, VA<18:13>, and VA<21:13> for 64 kB, 512 kB and 4 MB pages, respectively. Note that this field is sign-extended based on VA<43>.

I/D Context<12:0>: is the 13-bit context identifier.

An ASI store to the TLB Data Access register initiates an internal atomic write to the specified TLB Entry. The TLB entry data is obtained from the store data, and the TLB entry tag is obtained from the current contents of the TLB Tag Access register.

An ASI store to the TLB Data In register initiates an automatic atomic replacement of the TLB Entry pointed to by the current contents of the TLB Replacement register "Replace" field. The TLB data and tag are formed as in the case of an ASI store to the TLB Data Access register described above.

Caution – Stores to the Data In register are not guaranteed to replace the previous TLB entry causing a fault. In particular, to change an entry's attribute bits, software must explicitly demap the old entry before writing the new entry; otherwise, a multiple match error condition can result.

An ASI load from the TLB Data Access register initiates an internal read of the data portion of the specified TLB entry.

An ASI load from the TLB Tag Read register initiates an internal read of the tag portion of the specified TLB entry.

ASI loads from the TLB Data In register are not supported.

15.9.10 I-/D-MMU Demap

Demap is an MMU operation, as opposed to a register operation as described above. The purpose of Demap is to remove zero, one, or more entries in the TLB. Two types of Demap operation are provided: Demap page, and Demap context. Demap page removes zero or one TLB entry that matches exactly the specified virtual page number. Demap page may in fact remove more than one TLB entry in the condition of a multiple TLB match, but this is an error condition of the TLB and has undefined results. Demap context removes zero, one, or many TLB entries that match the specified context identifier.

Demap is initiated by a STXA with ASI=57_{16} for I-MMU demap or $5F_{16}$ for D-MMU demap. It removes TLB entries from an on-chip TLB. UltraSPARC IIi does not support bus-based demap. *Figure 15-15* shows the Demap format:

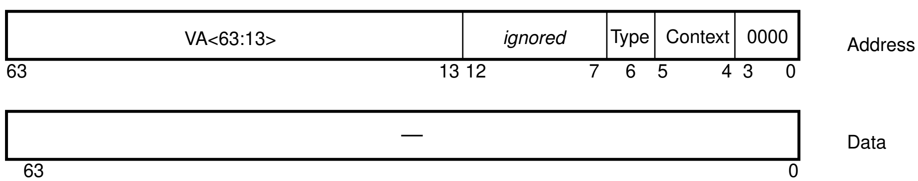


Figure 15-15 MMU Demap Operation Format

VA<63:12>: The virtual page number of the TTE to be removed from the TLB; This field is not used by the MMU for the Demap Context operation, but must be in-range. The virtual address for demap is checked for out-of-range violations, in the same manner as any normal MMU access.

Type: The type of demap operation, as described in *Table 15-16*

Table 15-16 MMU Demap operation Type Field Description

Type Field	Demap Operation
0	Demap Page
1	Demap Context

Context ID: Context register selection, as described in *Table 15-17*; Use of the *reserved* value causes the demap to be ignored.

Table 15-17 MMU Demap Operation Context Field Description

Context ID Field	Context Used in Demap
00	Primary
01	Secondary
10	Nucleus
11	*Reserved*

Ignored: This field is ignored by hardware. (The common case is for the demap address and data to be identical.)

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.

Note – A STXA to the data demap registers requires either a MEMBAR `#Sync`, FLUSH, DONE, or RETRY before the point that the effect must be visible to data accesses. A STXA to the I-MMU demap registers requires a FLUSH, DONE, or RETRY before the point that the effect must be visible to instruction accesses; that is, MEMBAR `#Sync` is not sufficient. In either case, one of these instructions must be executed before the next translating or bypass store or load of any type. This action is necessary to avoid corrupting data.

The demap operation does not depend on the value of any entry's lock bit; that is, a demap operation demaps locked entries just as it demaps unlocked entries.

The demap operation produces no output.

15.9.11 I-/D-Demap Page (Type=0)

Demap Page removes the TTE (from the specified TLB) matching the specified virtual page number and context register. The match condition with regard to the global bit is the same as a normal TLB access; that is, if the global bit is set, the contexts need not match.

Virtual page offset bits <15:13>, <18:13>, and <21:13>, for 64 kB, 512 kB, and 4 MB page TLB entries, respectively, are stored in the TLB, but do not participate in the match for that entry. This is the same condition as for a translation match.

Note – Each Demap Page operation removes only one TLB entry. A demap of a 64 kB, 512 kB, or 4 MB page does not demap any smaller page within the specified virtual address range.

15.9.12 I-/D-Demap Context (Type=1)

Demap Context removes all TTEs having the specified context from the specified TLB. If the TTE Global bit is set, the TTE is not removed.

15.10 MMU Bypass Mode

In a bypass access, the D-MMU sets the physical address equal to the truncated virtual address; that is, PA<40:0>=VA<40:0>. The physical page attribute bits are set as shown in *Table 15-18*.

Table 15-18 Physical Page Attribute Bits for MMU Bypass Mode

ASI	Physical Page Attribute Bits							
	CP	IE	CV	E	P	W	NFO	Size
ASI_PHYS_USE_EC ASI_PHYS_USE_EC_LITTLE	1	0	0	0	0	1	0	8 KB
ASI_PHYS_BYPASS_EC_WITH_EBIT ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE	0	0	0	1	0	1	0	8 KB

Bypass applies to the I-MMU only when it is disabled. See Section 15.7, *MMU Behavior During Reset, MMU Disable, and RED_state* on page 211 for details on the use of bypass when either MMU is disabled.

Compatibility Note – In UltraSPARC IIi the virtual address is longer than the physical address; thus, there is no need to use multiple ASIs to fill in the high-order physical address bits, as is done in SPARC-V8 machines.

15.11 TLB Hardware

15.11.1 TLB Operations

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

- Normal translation. The TLB receives a virtual address and a context identifier as input and produces a physical address and page attributes as output.

- Bypass. The TLB receives a virtual address as input and produces a physical address equal to the truncated virtual address 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. See Section 15.9.9, *I-TLB and D-TLB Data-In/Data-Access/Tag-Read Registers* on page 222.
- Write operation. The TLB simultaneously writes the CAM and RAM portion of the specified entry, or the entry given by the replacement policy described in Section 15.11.2.
- No operation. The TLB performs no operation.

15.11.2 TLB Replacement Policy

The dTLB and iTLB support a replacement algorithm based upon three status bits in each TLB entry, Locked, Used, and Valid. When software does a write of the I-TLB or the D-TLB Data In registers, using ASI 0x54 or 0x5C, the entry used for the write is selected depending upon the state of these bits.

The Valid bit is set when the TLB entry has valid data in it. The Used bit is set to 1 each time the entry is accessed for a translation. The Locked bit is set to lock the entry in the TLB.

Ordinarily the exact behavior of the Used bits is not of interest to software, and is only of interest in understanding the hardware. When there are no freely-available TLB entries (that is, with Valid == 0 or Used == 0), the hardware initiates a “Uclear” command to clear all the used bits in the TLB.

The TLB replacement algorithm begins with entry 0 and ends with entry 63. This selection algorithm is described in the following steps.

1. The first invalid entry is replaced, otherwise,
2. Use the first valid entry that is neither used nor locked, otherwise,
3. Use the first valid entry that is used but not locked, otherwise,
4. If all entries are valid, used, and locked, use table entry 63.

The exact LRU selection algorithm:

```
if (there exists x : x.v == 0) {
    first such x;
} elseif (there exists y: y.u == 0 && y.l == 0) {
    first such y;
```

```
} elseif (there exists z: z.l == 0) {
  first such z;
} else {
  entry 63;
}
```

A hardware "uclear", a clear of all the Used bits, can be triggered in just about any TLB cycle, even if the TLB is doing a write, for example. A uclear is triggered when: all entries are valid, and all entries have Lock==1 and Used==1,

So, for example, locking an entry that never gets the Used bit set, does not inhibit the uclear operation.

Any entry may have its lock bit set by software. However, the operation of the TLB is undefined if all entries have their lock bit set.

Due to the implementation of the UltraSPARC IIi pipeline, the MMU can and will set a TLB entry's used bit as if the entry were hit when the load or store is an annulled or mispredicted instruction. This can be considered to cause a very slight performance degradation in the replacement algorithm, although it may also be argued that it is desirable to keep these extra entries in the TLB.

See Erratum *58:* on page 454.

There is a case to consider in "lock-step" applications. An attempt by software to set the Used bit to 1 could result in an indeterminate value in this bit. This could cause "lock-step" CPUs to get out of sync., since the Used bit manipulations have to be exactly the same for two CPUs to operate identically.

Software should never write Used==1 (bit 0 of the Diag field, which is bit 41 of the Data In register), using Data In writes. This is because if a clear of the Used bits is being done in the same cycle by hardware, the results are indeterminate.

It appears there is no such constraint on Data Access writes.

15.11.3 TSB Pointer Logic Hardware Description

The hardware diagram in *Figure 15-16* on page 229 and the code fragment in *Code Example 15-1* on page 230 describe the generation of the 8 kB and 64 kB pointers in more detail.

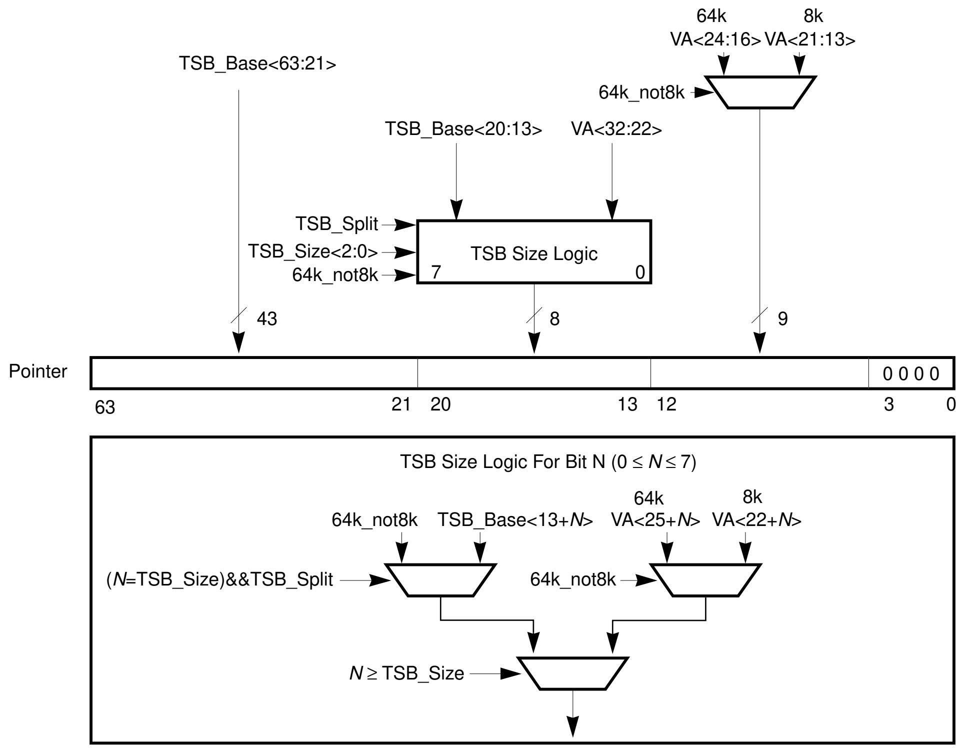


Figure 15-16 Formation of TSB Pointers for 8 kB and 64 kB TTEs

Code Example 15-1 Pseudo-code for UltraSPARC IIi D-MMU Pointer Logic

```
int64 GenerateTSBPointer(
  int64 va,            // Missing virtual address
  PointerType type,    // 8K_POINTER or 64K_POINTER
  int64 TSBBase,       // TSB Register<63:13> << 13
  Boolean split,       // TSB Register<12>
  int TSBSize)         // TSB Register<2:0>
{
  int64 vaPortion;
  int64 TSBBaseMask;
  int64 splitMask;
  // TSBBaseMask marks the bits from TSB Base Reg
  TSBBaseMask = 0xffffffffffffe000 <<
      (split? (TSBSize + 1) : TSBSize);

  // Shift va towards lsb appropriately and
  // zero out the original va page offset
  vaPortion = (va >> ((type == 8K_POINTER)? 9: 12)) &
      0xfffffffffffffff0;

  if (split) {
      // There's only one bit in question for split
      splitMask = 1 << (13 + TSBSize);
      if (type == 8K_POINTER)
          // Make sure we're in the lower half
          vaPortion &= ~splitMask;
      else
          // Make sure we're in the upper half
          vaPortion |= splitMask;
  }
  return (TSBBase & TSBBaseMask) | (vaPortion &
~TSBBaseMask);
}
```

CHAPTER **16**

Error Handling

This chapter describes error detection, correction, and error reporting mechanisms used in UltraSPARC IIi.

UltraSPARC IIi provides error checking for all memory access paths between the CPU, external cache (E-cache) and DRAM as well as for PCI data and address transfers. In particular:

- Memory accesses are protected by ECC.
- E-cache accesses are protected by parity checking.
- PCI data and address transfers are protected by parity checking.
- UPA64S address and data transfers do not employ error checking.

Errors are reported as system fatal errors, deferred traps or disrupting traps. System fatal errors are reported when the system must be reset before continuing. Deferred traps are reported for non-recoverable failures that require immediate attention without system reset. Disrupting traps are used to report errors that do not affect processor execution but which may need logging.

Non-fatal hardware errors may generate interrupts, set status register bits, or take no action.

Error information is logged in the Asynchronous Fault Address Register, Asynchronous Fault Status Register and the SDBH Error Register. See *ECU Asynchronous Fault Status Register* on page 243 and *SDBH Error Register* on page 247.

Errors are logged even if their corresponding traps are disabled.

16.1 System Fatal Errors

When an E-cache tag parity or system address parity error occurs, system coherency is lost and the system should be reset. When these errors occur and the corresponding error trap is enabled in the E-cache Error Enable Register, software should cause a power on reset. See *E-cache Error Enable Register* on page 242.

Compatibility Note – UltraSPARC automatically caused the reset through the UPA. The UltraSPARC IIi CPU currently does not cause an automatic reset.

16.2 Deferred Errors

Deferred errors may corrupt the processor state, and are normally irrecoverable. Such errors lead to termination of the currently executing process or result in a system reset if the system state has been corrupted. Software can detect this corrupted system state by interrogating error logging information.

A membar #Sync instruction provides an error barrier for deferred errors. It ensures that deferred errors from earlier accesses will not be reported after the membar. A membar #Sync should be used during context switching to provide error isolation between processes.

Note – After a deferred trap, the contents of TPC and TNPC are undefined (except for the special peek sequence described below). They do NOT generally contain the oldest non-executed instruction and its next PC. As a result, execution can not normally be resumed from the point that the trap is taken. Instruction access errors are reported before executing the instruction that caused the error, but TPC does not necessarily point to the corrupted instruction. Errors due to fetching user code after a DONE/RETRY are always reported after the DONE or RETRY. This guarantees that system code will not be aborted by a user mode instruction access.

When a deferred error occurs and the corresponding error trap is enabled in the E-cache Error Enable Register (see *E-cache Error Enable Register* on page 242), an *instruction_access_error* or *data_access_error* trap is generated. Deferred errors include:

- Data parity error during access from E-cache excluding writeback or copyback.
- Uncorrectable ECC error (UE) in memory access. Uncorrectable ECC errors on cache fills will be reported for any ECC error in the cache block, not just the referenced word.

- Time-out or bus error during a read access from the PCI bus.

When a deferred error occurs, trap handler execution is delayed until all outstanding accesses are completed. This delay avoids entering RED_state due to multiple errors. Any subsequent errors detected during this waiting period will be properly logged.

Errors that occur after the trap handler begins will be due to an access from inside the trap handle.

The instruction and data caches are disabled by clearing the IC and DC bits in the LSU control register. This is because corrupted data may be placed in the cache if the access was cacheable. The caches must be reenabled by software after flushing to remove the corrupted data. In case of an instruction error, the instruction returned to the CPU is marked for termination (to be aborted). This means that a bad instruction will not create programmer-visible side effects.

16.2.1 Probing PCI during boot using deferred errors

Intentional peeks and pokes to test presence and operation of devices are recoverable only if performed as follows.

- The access should be preceded and followed by membar #Sync instructions.
- The destination register of the access may be destroyed, but no other state will be corrupted.
- If TPC is pointing to the membar #Sync following the access, then the *data_access_error* trap handler knows that a recoverable error has occurred and resumes execution after setting a status flag.
- The trap handler will have to set TNPC to TPC + 4 before resuming, because the contents of TNPC are otherwise undefined.

16.2.2 General software for handling deferred errors

The following is a possible sequence for handling deferred errors within the trap handler.

1. Log the errors.

2. Reset the error logging bits in AFSR and SDB error registers if needed. Perform a membar #sync to complete internal ASI stores.

3. Panic if AFSR.PRIV is set and not performing an intentional peek/poke, otherwise try to continue.

4. Displacement flush the entire E-cache. This action will remove corrupted data from the I-cache, D-cache, and E-cache. This step is not necessary for known non-cacheable accesses.
5. Re-enable I and D caches by setting the IC and DC bits of the LSU control register. Perform a membar #sync to complete internal ASI stores.
6. Abort the current process.
7. If uncorrectable ECC error, and no other processes share the data, perform a block store to the block address in AFAR to reset ECC. Perform a membar #sync to complete the block store.
8. Resume execution.

16.3 Disrupting Errors

Disrupting errors are single-bit ECC Errors (which are corrected by the hardware) or E-cache data parity errors during write back. Disrupting errors should be handled by logging the error and resuming execution.

Recoverable ECC errors result from detection of a single-bit ECC error during a system transaction. Memory read errors are logged in the Asynchronous Fault Status Register (and possibly in the Asynchronous Fault Address Register). If the Correctable_Error (CEEN) trap is enabled in the E-cache Error Enable Register, a *corrected_ECC_error* trap is generated. This trap has trap type TT==0x63 and priority 33.

E-cache data parity errors are discussed in *E-cache Data Parity Error* on page 235. An E-cache data parity error during writeback is recoverable because the processor is not reading the affected data. As a result, UltraSPARC IIi takes a disrupting *data_access_error* trap with priority 33 instead of a deferred trap. This avoids panics when the system displaces corrupted user data from the cache.

Note – To prevent multiple traps from the same error, software should not re-enable interrupts until after the disrupting error status bit in AFSR is cleared.

16.4 E-cache, Memory, and Bus Errors

16.4.1 E-cache Tag Parity Error

Tag parity errors from internal or snoop transactions cause a system fatal error as described in *System Fatal Errors* on page 232.

The E-cache Tag RAM is protected by parity. Data stored in the E-cache Tag RAM includes 16 bits of E-cache tag, 2 bits of E-cache state, and 4 bits of parity. This is reduced, compared to UltraSPARC-I. (to save pins)

There are 2 parity bits for 16 bits of data.

- Parity<0>: E-cache Tag <7:0>
- Parity<1>: E-cache state[1:0] & E-cache Tag <13:8>

UltraSPARC IIi is normally enabled to trap if it detects an E-cache tag parity error.

16.4.2 E-cache Data Parity Error

The E-cache data bus connects the UltraSPARC IIi processor and E-cache data SRAM. The 64-bit wide data bus is protected by byte parity. Parity check failures on this bus can be caused by faulty devices or interconnects.

UltraSPARC IIi performs parity checking during;

1. Processor reads from E-cache

2. Reads due to snooping (copyback) and victimization (writeback).

A parity error detected during an E-cache data access can cause UltraSPARC IIi to trap.

An E-cache data parity error detected during an instruction access causes an *instruction_access_error* deferred trap. An E-cache parity error detected during a data read access causes a *data_access_error* deferred trap. When multiple errors occur, the trap type corresponds to the first detected error.

If an E-cache data parity error occurs while snooping, a bad ECC error is generated and sent to the requester. This causes an *instruction/data_access_error* trap at the master that requested the data. The slave processor logs error information that can be read by the master during error handling. The processor being snooped is not interrupted by this error condition.

Compatibility Note – If an E-cache data parity error occurs during a write-back, uncorrectable ECC is *not* forced to memory. However, the error information is logged in the AFSR and a disrupting *data_access_error* trap is generated.

16.4.3 DRAM ECC Error

UltraSPARC IIi supports ECC generation and checking for all accesses to and from the DRAM. Correctable errors (CE) are fixed and the data transfer continues. Uncorrectable ECC errors on cache fills are reported for any ECC error in the cache block, not just for the referenced word.

An uncorrectable error detected during an instruction access causes an *instruction_access_error* deferred trap. An uncorrectable error detected during a data access causes a *data_access_error* deferred trap. When multiple errors occur, the trap type corresponds to the first detected error.

16.4.4 CE/UE

If the Memory Control Unit detects a CE, data is corrected before it is used. This is done in these cases:

- PCI DMA reads from memory
- PCI DMA partial line writes to memory

DMA ECC errors are reported to the processor via interrupt as long as ECC checking and ECC interrupt are both enabled. Error information is logged in the DMA UE or CE AFSR/AFAR.

Processor UEs and CEs are reported via trap, and are separately maskable.

16.4.5 Timeout

An attempted read of an unsupported or nonexistent device results in a timeout (TO). For example, a TO results from a read of a PCI bus address unmapped to a PCI device. Writes to non-mapped PCI addresses are reported via a late interrupt.

16.4.6 PCI Timeout

A timeout is sent (TO in Section 16.6.2, *ECU Asynchronous Fault Status Register* on page 243) to the UltraSPARC IIi core under a variety of PIO read error cases. If no device is mapped (or responds) to the PCI address the transaction is terminated with a master-abort and the UltraSPARC IIi RMA Status bit is set.

If a device terminates a PIO read with too many retries (disconnect with no data transfer) UltraSPARC IIi stops retrying the access and causes a TO. A maximum of 512 retries (according to the contents of the PCI Configuration Space Retry Limit Counter Register) are allowed, although this limit can be disabled.

PCI has no timeout mechanism analogous to the S-Bus timeout. However, the PCI specification does recommend that all targets issue a retry when more that 16 PCI clocks will be consumed waiting for the first data transfer. When a device claims the transaction but never signals that it is ready to transfer data, the system hangs. This situation only occurs because of a device hardware error.

16.4.7 PCI Data Parity Error

PCI requires all devices to generate parity for the address/data and cmd/byte enable busses. A single even parity bit is used for 32 bits of address/data and 4-bit cmd/byte enable bus.

This section covers only parity errors on data phases, address parity errors are covered in *PCI Address Parity Error* on page 239.

Reporting of parity errors may be disabled using the PER bit described in section Section 19.3.1.3, *PCI Configuration Space Command Register* on page 291.

Setting PER enables UltraSPARC IIi to report PIO data parity errors to the processor and DMA data parity errors to the bus master. When a data parity error is detected or signalled, UltraSPARC IIi does not terminate the transaction prematurely but attempts to take it to completion.

If PER is enabled, a parity error detected on PIO read is reported with a BERR to the UltraSPARC IIi core, along with setting the DPE and DPD bits described in *PCI Configuration Space Status Register* on page 292. The PCI signal 'PERR#' is also asserted,

---

Compatibility Note – If PER is disabled, UltraSPARC IIi does not set DPE if it detects a parity error on PIO reads. This is inconsistent with the PCI 2.1 spec.

A parity error signalled via PERR# on a PIO write is logged if PER is enabled. In this case the DPD bit and the PCI PIO Write AFSR P_PERR/S_PERR bits are set in the PCI Configuration Space Status Register, the PCI PIO Write AFAR is loaded with the PIO address, and an interrupt is generated.

A parity error detected during a DMA write is logged if PER is enabled. The DPE bit in the PCI Configuration Space Status Register is set, and PERR# is asserted to the bus master. Subsequent action taken by the master is device dependent.

Compatibility Note – If PER is disabled, UltraSPARC IIi does not set DPE if it detects a parity error on DMA writes. This is inconsistent with the PCI 2.1 spec.

Data parity is not checked during DMA reads. Also, since UltraSPARC IIi is not the bus master, PERR# is ignored.

Note, however, that parity includes CBE#, which is driven to UltraSPARC IIi, and part of the parity bit generation. It is an interesting part of the protocol that parity includes bits (CBE#/AD) driven by two different parties. If the CBE# is only wrong to UltraSPARC IIi for a DMA read, the parity error goes unreported.

16.4.8 PCI Target-Abort

If an error occurs during an access of a PCI device, the device may terminate the transaction with a target-abort. Examples of causes of this result are unsupported byte enables, an address parity error, and device-specific errors. Any data that may have been transferred during the transaction before the target-abort occurred is corrupt and must not be used by the recipient.

A PIO read terminated with a target-abort results in a Bus Error (BERR in *ECU Asynchronous Fault Status Register* on page 243) to the UltraSPARC IIi core and the RTA bit being set in the PCI Configuration Space Status Register.

A PIO write that is terminated with a target-abort results in an asynchronous error. The P_TA/S_TA bit is set in the PCI PIO Write AFSR and the physical address loaded into the PCI PIO Write AFAR. The RTA bit in the PCI Configuration Space Status Register is also set for writes.

UltraSPARC IIi issues a target-abort upon detecting an address parity error, taking an IOMMU address translation error, and detecting a UE ECC error. The STA bit is set in the PCI Configuration Space Status Register but in all cases it is the responsibility of the bus master to report the error to system software (using SERR# or a device-specific interrupt).

16.4.9 DMA ECC Errors

The PCI DMA UE/CE AFSR/AFAR registers log DMA errors.

1. If UE interrupts are enabled, an interrupt is posted when UltraSPARC IIi detects a UE.
2. A UE on any of the data for a DMA read (up to a 64 byte prefetch if from memory) causes a target-abort to the PCI master device as soon as possible. This may be before the DMA read operation reaches the data transfer cycle with the UE data.
3. During DMA writes of less than 16 bytes, good data and check bits are provided for all 16 bytes when completing a Read-Modify-Write to memory. If a DMA transaction does not overwrite, or only partially overwrites, the UE data, note that bad data may then appear as good in memory.

16.4.10 IOMMU Translation Error

The IOMMU translates the PCI DMA address to a physical page address and checks for access violations. The IOMMU can detect the "access to a invalid page" and "access with protection violation" errors.

An invalid error occurs when the DMA page address lacks a valid physical page mapped to it. A protection error occurs when the PCI master attempts to write to a page that is marked as read-only. Both errors are reported with a target-abort to the device.

Compatibility Note – A new feature for UltraSPARC IIi, is that the VA of the offending DMA access is logged in the PCI DMA UE AFSR and AFAR, with the a bit set for identification as a DMA translation error.

Additional reporting of translation errors by the initiating PCI master is device dependent.

16.4.11 PCI Address Parity Error

PCI Address parity errors may be reported during PIO operations and detected or reported during DMA transfers. The PCI mechanism for reporting address parity errors is the "System Error". Address parity error reporting can be disabled (together with all parity error reporting) using the PER PCI Configuration Space Command Register bit.

After detecting a DMA address parity error, UltraSPARC IIi first sets the DPE bit in the PCI Configuration Space Status Register. If PER is enabled, it then issues a target-abort to the master, and generates a PCI Error interrupt with the PCI_SERR bit in the PCI Control and Status Register set.

If both PER and SERR_EN are enabled in the PCI Configuration Space Command Register, UltraSPARC IIi also asserts SERR# on the bus and sets the SSE bit in the PCI Configuration Space Status Register.

When a PIO address parity error is reported by a device via a SERR# assertion, UltraSPARC IIi reports the system error as described in *PCI System Error* on page 240. Upon detecting the address parity error the target device has the options:

1. Not claiming the transaction, causing a TO trap to UltraSPARC IIi core
2. Issuing a target-abort, resulting in an BERR trap to UltraSPARC IIi core for reads and an asynchronous error interrupt for writes
3. Completing the cycle as if there were no error and either generating a system error or an interrupt at some later time

16.4.12 PCI System Error

The PCI System Error (PCI bus SERR# assertion) may occur on address parity errors as well as on device specific fatal errors. The assertion of SERR# can be disabled by the SERR_EN PCI Configuration Space Command Register bit.

Any PCI device may assert SERR# at any time but only UltraSPARC IIi can detect and report it to system software. SERR# assertion causes a PCI Error Interrupt and sets the PCI_SERR bit in the PCI Control and Status Register.

Devices that assert the SERR# must set their SSE Status register bit. Multiple system errors generated before the system software clears the PCI CSR do not cause additional interrupts, so it is important that software check all device PCI Configuration Space Status registers.

16.5 Summary of Error Reporting

Register abbreviations are: PCI CSR for the PCI Control/Status Register, and PCI Status for the PCI Configuration space Status register. AFR indicates both an AFSR and an AFAR.

Table 16-1 Summary of Error Reporting

Transaction	Error Type	CPU Response	Error Register(s)	PCI Bus
Fetch, LD/ST, PCI DMA, Writeback	E$Tag/Data Ram Parity Error	ETP/EDP/WP/CP (ECU AFSR), Trap	ECU AFRs	-
PIO Read	Data parity	BERR (ECU AFSR), Trap	PCI CSR, PCI Status, ECU AFRs	Complete Transaction
	Master-abort	TO (ECU AFSR), Trap	PCI Status, ECU AFRs	Master-abort
	Target-abort	BERR (ECU AFSR), Trap	PCI Status, ECU AFRs	Target-abort
	Retry Limit	TO (ECU AFSR), Trap	PCI Status, ECU AFRs	Cease Retries
PIO Write	Master-abort	PCI Error Interrupt	PCI PIO Write AFRs, PCI Status	Master-abort
	Target-abort	PCI Error Interrupt	PCI PIO AFRs, PCI Status	Target-abort
	Retry Limit	PCI Error Interrupt	PCI PIO AFRs	Cease Retries
	Data Parity	PCI Error Interrupt	PCI PIO AFRs, PCI Status	Complete Transaction
Any PIO	Address Parity Error	-	-	Device dependent
DMA Read	UE-ECC	PCI UE Interrupt	PCI DMA UE AFRs, PCI Status	Target-abort
	CE-ECC	PCI CE Interrupt	PCI DMA CE AFRs	Complete Transaction
	Ecache Data Parity	CP (ECU AFSR), Trap	ECU AFSR	Complete Transaction
DMA Write	UE-ECC[1]	PCI UE Interrupt	PCI DMA UE AFRs	Complete Transaction
	CE-ECC	PCI CE Interrupt	PCI DMA CE AFRs	Complete Transaction
	Data Parity	-	PCI Status	Complete Transaction, PERR#
Any DMA	Address Parity	PCI Error Interrupt	PCI Status	Target-abort
	Translation Error	PCI UE Interrupt	PCI Status, PCI DMA UE AFRs IOMMU Control Reg	Target-abort
PCI System Error	SERR# assertion	PCI Error Interrupt	PCI CSR, PCI Status	-

1. Less than 16-byte aligned write to DRAM only

Unreported Errors

Some error conditions are not reported by the system. The following list gives examples of these errors:

- A write to a non-supported address.
- A write to a read-only register in UltraSPARC IIi is ignored.
- A non-cached write to memory.
- A read from a write-only register in UltraSPARC IIi returns unknown data.

This list may not be exhaustive.

16.6 E-cache Unit (ECU) Error Registers

Note – MEMBAR `#Sync` is generally needed after stores to error ASI registers.

16.6.1 E-cache Error Enable Register

Name: ASI_ESTATE_ERROR_EN_REG

ASI_ESTATE_ERROR_EN_REG: ASI== 0x4B, VA<63:0>==0x0

Table 16-2 E-cache Error Enable Register Format

Bits	Field	Use	Reset	RW
<63:4>	Reserved	—	0	R0
<4>	EPEN	Trap on ETP, EDP, WP, CP	0	RW
<3>	UEEN	Trap on UE	0	RW
<2>	Reserved		0	RW
<1>	NCEEN	Trap on TO, BERR, ETP, EDP, WP, CP, UE	0	RW
<0>	CEEN	Trap on correctable memory read error	0	RW

EPEN: Additional enable on ETP and EDP errors. See NCEEN.

UEEN: Additional enable on UE errors. See NCEEN.

NCEEN: If set, an uncorrectable error, time-out, bus error, SDB or E-cache data parity error causes an *{instruction, data}_access_error* trap and an E-cache tag parity error should cause a system fatal error; otherwise, the error is logged in the AFSR and ignored.

CEEN: If set, a correctable error detected during a memory read access causes a *correctable_ECC_error* disrupting trap; otherwise, the error is logged in the AFSR and ignored.

Examples:

- Disable all traps: [4:0] = xxx00
- Disable SRAM parity, Disable ECC, Enable Bus traps: [4:0] = 00x10
- Disable SRAM parity, Enable ECC, Enable Bus traps: [4:0] = 01x11
- Enable SRAM parity, Enable ECC, Enable Bus traps: [4:0] = 11x11

16.6.2 ECU Asynchronous Fault Status Register

The Asynchronous Fault Status Register (AFSR) logs all errors that occurred since its fields were last cleared. The AFSR is updated according to the policy described in *Overwrite Policy* on page 249.

The AFSR is logically divided into four fields:

- Bit <32>, the accumulating multiple-error (ME) bit, is set when multiple errors with the same sticky error bit have occurred except for correctable errors. Multiple errors of different types are indicated by setting more than one of the sticky error bits.
- Bit <31>, the accumulating privilege-error (PRIV), is set when an error occurs from an access generated by code executing with PSTATE.PRIV = 1. If this bit is set, system state has been corrupted.
- Bits <30:20> are sticky error bits that record the most recently detected errors. These sticky bits accumulate errors detected since the last write that cleared this register.
- Bits <17:16>, <7:0> contain the tag and data parity syndromes respectively. Syndrome bits are endian-neutral, that is, bit 0 corresponds to bits<7:0> of the E-cache data bus (i.e. bytes whose least significant four address bits are 0xf). The syndrome fields have the status of the first occurrence of the highest priority error related to that field. If no status bit is set that corresponds to that field, the contents of the syndrome field will be zero.

The AFSR must be explicitly cleared by software; it is *not* cleared automatically during a read. Writes to the AFSR sticky bits (<32:20>) with particular bits set clear the corresponding bits in the AFSR. Bits associated with disrupting traps must be cleared before re-enabling interrupts to prevent multiple traps for the same error. Writes to the AFSR sticky bits with particular bits clear will not affect the corresponding bits in the AFSR. If software attempts to clear error bits at the same time as an error occurs, the clear will be performed before applying logging the new error status. The syndrome field is read only and writes to this field are ignored.

Name: ASI_ASYNC_FAULT_STATUS

ASI_ASYNC_FAULT_STATUS: ASI== 0x4C, VA<63:0>==0x0

Table 16-3 Asynchronous Fault Status Register

Bits	Field	Use	Reset	RW
<63:33>	Reserved	—	0	R
<32>	ME	Multiple Error of same type occurred	0	RW1C
<31>	PRIV	Privileged code access error(s) has occurred	0	RW1C
<30>	Reserved	Read as 0	0	R0
<29>	ETP	Parity error in E-cache Tag SRAM	0	RW1C
<28>	Reserved	Read as 0	0	R0
<27>	TO	Time-Out from PCI PIO load or Inst. fetch	0	RW1C
<26>	BERR	Bus Error from PCI PIO load or Inst. fetch	0	RW1C
<25>	Reserved	Read as 0	0	R0
<24>	CP	PCI DMA E-cache Parity error	0	RW1C
<23>	WP	Data parity error from E-cache SRAMs for Write-back (victim)	0	RW1C
<22>	EDP	Data parity error from E-cache SRAMs	0	RW1C
<21>	UE	Uncorrectable ECC error (E_SYND in SDB registers)	0	RW1C
<20>	CE	Correctable memory read ECC error (E_SYND in SDB registers)	0	RW1C
<19:18>	Reserved	Read as 0	0	R0
<17:16>	ETS	E-cache Tag parity Syndrome	0	R
<15:8>	Reserved	Read as 0	0	R0
<7:0>	P_SYND	Parity Syndrome	0	R

.

Table 16-4 E-cache Data Parity Syndrome Bit Orderings

Byte address	E- cache data bus bits	Syndrome Bit
0x7	<7:0>	0
0x6	<15:8>	1
0x5	<23:16>	2
0x4	<31:24>	3
0x3	<39:32>	4
0x2	<47:40>	5
0x1	<55:48>	6
0x0	<63:56>	7
	Always 0	15:8

Table 16-5 E-cache Tag Parity Syndrome Bit Orderings

E-cache Tag bus bits	Syndrome Bit
<7:0>	0
<15:8>	1
Always 0	3:2

16.6.3 ECU Asynchronous Fault Address Register

This register is valid when one of the Asynchronous Fault Status Register (AFSR) error status bits that capture address is set (for example, for correctable or uncorrectable memory ECC error, bus time-out or bus error). The address corresponds to the first occurrence of the highest priority error in AFSR that captures address See *AFAR Overwrite Policy* on page 249. Address capture is reenabled by clearing all corresponding error bits in AFSR. If software attempts to write to these bits at the same time as an error that captures address occurs, the error address is stored.

Name: ASI_ASYNC_FAULT_ADDRESS

ASI_ASYNC_FAULT_ADDRESS: ASI== 0x4D, VA<63:0>==0x0

Table 16-6 Asynchronous Fault Address Register

Bits	Field	Use	RW
<63:41>	Reserved	—	R0
<40:3>	PA<40:3>	Physical address of faulting transaction	RW
<2:0>	Reserved	—	R0

PA: Address information for the most recently captured error

Table 16-7 Error Detection and Reporting in AFAR and AFSR

Error Type	PA	SYNDROME[5]	Trap	PRIV captured?	Trap Type[6]	Updated status	SW Cache flush
Uncorrectable ECC	Y	E_SYND[3]	Deferred	Y	I[4], D	UE	Yes if cacheable
Correctable ECC	Y	E_SYND	Disrupting	N	C	CE	No
E$ parity: UltraSPARC IIi LD/Fetch	N[2]	P_SYND	Deferred	Y	I, D	EDP	Yes
E$ parity: writeback	N	P_SYND	Disrupting	N	D	WP	No
E$ parity: DMA read	N	P_SYND	Disrupting	N	D	CP	No
Bus Error[1]	Y	—	Deferred	Y	I, D	BERR	Never for Cacheable
Time-out	Y	—	Deferred	Y	I, D	TO	Never for Cacheable
Tag parity	N	ETS	Deferred	N	I, D	ETP	power on clear

1. PCI transactions can cause Bus Error and Time-out. See *Summary of Error Reporting* on page 240.
2. No address captured on parity errors.
3. E_SYND i s ECC syndrome; P_SYND i s parity syndrome; ETS i s E-cache Tag Parity Syndrome
4. I is *instruction_access_error* trap; D is *data_access_error* trap; C is *corrected_ECC_error* trap; POR is power-on reset trap

---

Compatibility Note – UltraSPARC IIi does not Target Abort on a a parity error resulting from a DMA read of E-cache. UltraSPARC caused a UE at the receiver of the data. Currently it is only reported with the same priority/trap as WP (but CP bit set).

Compatibility Note – UltraSPARC IIi causes a Deferred Trap similarly to UltraSPARC for ETS, without a system reset. Software can determine if a system reset is necessary.

16.6.4 SDBH Error Register

Compatibility Note – The SDB name is inherited from UltraSPARC. It logs information about memory errors caused by the CPU core. Only the SDBH register is used. Current Solaris software interrogates if SDBL is non-zero, and ORs in a 1 to the logged pa[3] (which is always zero on UltraSPARC, but valid on UltraSPARC IIi).

For implementation efficiency, the UltraSPARC Data Buffer (SDB) error and control registers were physically separated into upper half and lower half registers.

Separate ASIs are used for reading (0x7F) and writing (0x77) the SDB registers.

If software attempts to clear these bits at the same time as an error occurs, the appropriate error bit is set to avoid losing error information.

On UltraSPARC IIi, writes to SDBL registers have no effect, and reads of SDBL registers always return zeros.

Name: ASI_SDBH_ERROR_REG_WRITE

ASI 0x77, VA<63:0>==0x0

Name: ASI_SDBH_ERROR_REG_READ

ASI 0x7F, VA<63:0>==0x0

Table 16-8 SDBH Error Register Format

Bits	Field	Use	Reset	RW
<63:10>	Reserved	—	0	R0
<9>	UE	If set, UE has occurred	0	RW1C
<8>	CE	If set, CE has occurred	0	RW1C
<7:0>	E_SYNDR	ECC syndrome from system.	-	R

E_SYNDR: ECC syndrome for correctable error from system. In case of multiple outstanding errors, only the first is recorded.

Bits <9:8> are sticky error bits that record the most recently detected errors. These bits accumulate errors detected since the last write that cleared this register.

The SDB error registers are *not* cleared automatically during a read. Writes to these registers with bit-8 or bit-9 set clear the corresponding bits in the error register.

Writes to the error register with particular bits clear will not affect the corresponding bits in the error register. The syndrome field is read only and writes to this field are ignored.

Note – A recorded correctable error may be overwritten by an uncorrectable error.

16.6.5 SDBL Error Register

Name: ASI_SDBL_ERROR_REG_WRITE

ASI 0x77, VA<63:0>==0x18

Name: ASI_SDBL_ERROR_REG_READ

ASI 0x7F, VA<63:0>==0x18

Writes have no effect, Reads return 0. This property allows existing US-I and US-II software to work without change.

16.6.6 SDBH Control Register

Name: ASI_SDBH_CONTROL_REG_WRITE

ASI 0x77, VA<63:0>==0x20

Name: ASI_SDBH_CONTROL_REG_READ

ASI 0x7F, VA<63:0>==0x20

Table 16-9 SDBH Control Register Format

Bits	Field	Use	Reset	RW
<63:17>	Reserved	—	0	R
<16:13>	Undefined	Reserved	-	R
<12:9>	VERSION	Always 0	0	R
<8>	F_MODE	Force ECC error	0	RW
<7:0>	FCBV	Force check bit vector	0	RW

VERSION: reads as 0 on UltraSPARC IIi.

F_MODE: If set, the contents of the FCBV field are sent with the out-going transaction, instead of the generated ECC.

FCBV: Force check bit vector.

16.6.7 SDBL Control Register

Name: ASI_SDBL_CONTROL_REG_WRITE

ASI 0x77, VA<63:0>==0x38

Name: ASI_SDBL_CONTROL_REG_READ

ASI 0x7F, VA<63:0>==0x38

Writes have no-effect, Reads return 0. This allows existing US-I and US-II software to work without change.

16.6.8 PCI Unit Error Registers

See *DMA Error Registers* on page 316 and *PCI PIO Write Asynchronous Fault Status/ Address Registers* on page 284.

16.7 Overwrite Policy

This section describes the overwrite policy for error bits when multiple errors conditions have occurred. Errors are captured in the order that they are detected, not necessarily in program order.

If an error occurs while error bits are being cleared by software, the overwrite control includes the effect of the software clear. For example, if ETP were set (which blocks E-cache tag syndrome updates) and software clears the ETP bit at the same time as an E-cache tag parity error occurs, the E-cache tag syndrome is updated.

16.7.1 AFAR Overwrite Policy

The Priority for AFAR updates is UE > CE > {TO, BE}

The physical address of the first error within a class (UE, CE, {TO, BE}) is captured in the AFAR until the associated error status bit is cleared in AFSR, or an error from a higher priority class occurs. A CE error overwrites prior TO or BE errors. A UE error overwrites prior CE, TO and BE errors.

16.7.2 AFSR Parity Syndrome (P_SYND) Overwrite Policy

Parity information for the first occurrence of any error is captured in the P_SYND field of the AFSR. Error logging is re-enabled by clearing the EDP, CP, and WP fields. Any set bits in these fields inhibit update to the P_SYND field.

16.7.3 AFSR E-cache Tag Parity (ETS) Overwrite Policy

Parity information for the first occurrence of any error is captured in the ETS field of the AFSR register. Error logging in this field can be re-enabled by clearing the ETP field.

16.7.4 SDB ECC Syndrome (E_SYND) Overwrite Policy

Priority for E_SYND updates is: UE > CE

The ECC syndrome of the first error within a class (UE, CE) is captured in the E_SYND field of the SDB Error Register until the associated error status bit is cleared in the SDB error register or an error from a higher priority class occurs. A UE error overwrites prior CE errors.

CHAPTER 17

Reset and RED_state

17.1 Overview

A reset is anything that causes an entry to RED_state. UltraSPARC IIi system resets are generated either from signals sourced from the external system or from resets generated and observed only by the UltraSPARC IIi core. In addition to forcing entry to RED_state, various resets cause different effects in initializing processor state.

The power supply, push-button, scan interface, software, error conditions, and power management logic can create externally sourced resets. Their signals are converted into power-on-reset (POR) or externally initiated reset (XIR) signals that pass to the core with different levels of effect on the system. Information from peripheral logic is stored in UltraSPARC IIi's Reset_Control register for software to determine the cause of the external reset. Software-Initiated Reset (SIR) and Watchdog Reset (WDR) resets result from core conditions and are generated and observed only by the processor core. Resets are used to force all or part of the system into a known state. UltraSPARC IIi distributes the resets to all subsystems, including the UPA64S device and the primary PCI bus reset. If APB is present, it propagates this reset to the secondary PCI buses.

Resets in general drive the processor into RED_state—described in Section 17.3, *RED_state*—with the exceptions described in that section.

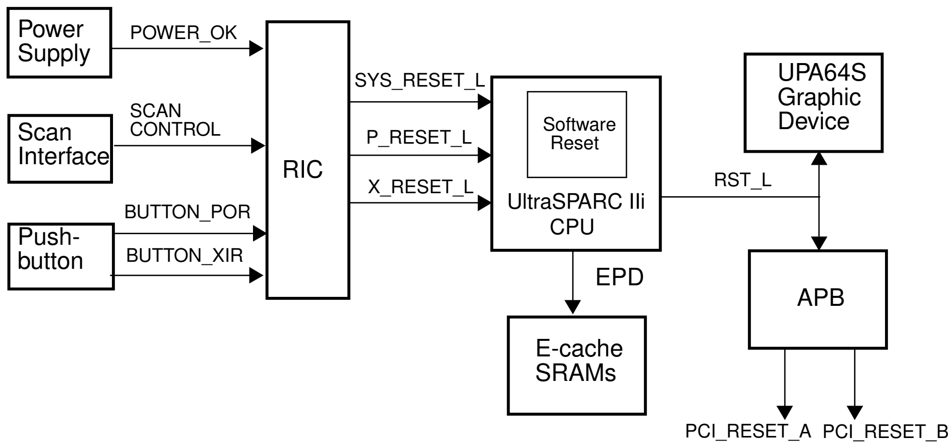


Figure 17-1 Reset Block Diagram

The assertion of RST_L is asynchronous to UPA clock. PCI specifies an asynchronous, monotonic, deassertion for RST_L.

Note – Most existing UPA64S devices can tolerate an asynchronous deassertion of UPA_RESET_L (the UPA spec says it should be a synchronous deassertion).

17.2 Resets

17.2.1 Power-on Reset (POR) and Initialization

A Power-on Reset occurs when the POR signal is asserted and stays until the CPU voltages reach their operating specifications and POR becomes inactive. When the POR pin is active, all other resets and traps are ignored. Power-on Reset has a trap type of 001_{16} at physical address offset 20_{16}. Any pending external transactions are cancelled.

After a Power-on Reset, software must initialize values specified as *unknown* in Section 17.4, *Machine State after Reset and in RED_state*. In particular, the Valid and LRU bits in the I-cache (Section A.7, *I-cache Diagnostic Accesses* on page 373), the Valid bits in the D-cache (*D-cache Diagnostic Accesses* on page 378), and all E-cache tags and data (*E-cache Diagnostics Accesses* on page 380) must be cleared before enabling the caches. The iTLB and dTLB also must be initialized as described in Section 15.7, *MMU Behavior During Reset, MMU Disable, and RED_state* on page 211.

Reset priorities from highest to lowest are: POR, XIR, WDR, SIR. See the following sections for explanations of each reset.

Note – Each register must be initialized before it is used. For example, CWP must be initialized before accessing any windowed registers, since the CWP register selects which register window to access. Failure to initialize registers or states properly prior to use may result in unpredicted or incorrect results.

17.2.2 Externally Initiated Reset (XIR)

An Externally Initiated Reset is sent to the CPU via the XIR pin; it causes a SPARC-V9 XIR, which has a trap type of 003_{16} at physical address offset 60_{16}. It has higher priority than all other resets except POR. XIR is used for system debug.

17.2.3 Watchdog Reset (WDR) and error_state

A SPARC-V9 processor enters error_state when a trap occurs and TL = MAXTL. The processor signals itself internally to take a *watchdog_reset* (WDR) trap at physical address offset 40_{16}. This reset affects only one processor, rather than the entire system. CWP updates due to window traps that cause watchdog traps are the same as the no watchdog trap case.

17.2.4 Software-Initiated Reset (SIR)

A Software-Initiated Reset is invoked by a SIR instruction within the processor core. This processor reset has a trap type of 004_{16} at physical address offset 80_{16} and affects only the processor, not IO or the external system. A Signal Monitor (SIGM) instruction generates an SIR trap on the local processor.

17.2.5 Hardware Reset Sources

The RIC chip detects five different resets: POWER_OK from the power supply, Push-button PowerOnReset, Push-button XIR, Scan PowerOnReset, and ScanXIR. RIC chip combines the 5 reset conditions into 3 signals to the UltraSPARC IIi. Based on these signals from RIC, UltraSPARC IIi will set bits in the Reset_Control Register to allow software identify the source of reset. If the RIC IC is not used, other logic should perform a similar power-up reset function.

17.2.5.1 Power Supply

After the system power supply is turned on and before its output stabilizes, it drives the POWER_OK signal inactive to put the system in a reset state. When the supply voltage reaches a level that can power a functional system within specifications, the power supply sets POWER_OK active.

RIC chip uses this signal to generate power-on-reset (POR) during the period POWER_OK is inactive to reset the system. It extends the reset period for 20K cycles at 7.159Mhz (approximately. 2.8ms) after the POWER_OK signal becomes active.

The extra time is needed to allow the PLL circuitry on UltraSPARC IIi to stabilize. RIC chip asserts SYS_RESET_L to UltraSPARC IIi during the whole reset period.

After the deassertion of SYS_RESET_L, UltraSPARC IIi keeps RST_L (the reset signal for peripheral logic) asserted for 1666668 processor clocks which represents at least 5.5 ms at 300 MHz.

17.2.5.2 Push-button Power On Reset

Two alternative external push-buttons allow user-triggered system resets: Push-button POR and Push-button XIR. Push-button POR has the same effect as a POR from the power supply. The only difference between these two resets is the resultant status bits in the UltraSPARC IIi Reset_Control Register and the state of refresh (unchanged with Push-Button POR). The B_POR bit is set to indicate that the reset is caused by push-button POR.

17.2.5.3 Push-button XIR

Push-button XIR allows a user-reset of part of the processor without resetting the whole system. UltraSPARC IIi sets the B_XIR bit in the Reset_Control Register when a Push-button XIR is detected. XIR affects the UltraSPARC core only without affecting the rest of the system, such as UltraSPARC IIi IO, memory and I/O devices.

The effect of XIR on the UltraSPARC processor is different from that of POR—see Section 17.2.1, *Power-on Reset (POR) and Initialization*, Section 17.2.2, *Externally Initiated Reset (XIR)*, and *Table 17-5*.

Note – Do not assert Button POR and Button XIR while coming out of a system reset (power on condition). This action activates a special test mode used for acquiring test patterns and this mode runs a shortened reset sequence.

17.2.6 Software Reset

17.2.6.1 Software Power On Reset

Software can also generate a POR-equivalent reset by setting the SOFT_POR bit in the UltraSPARC IIi Reset_Control Register. This reset is different from the SIR supported in the UltraSPARC core.

Note – As for prior UltraSPARC-based systems, refresh is not disabled

---

17.2.6.2 Soft XIR

Software can also issue XIR to the processor by setting the SOFT_XIR bit in the UltraSPARC IIi Reset_Control Register. SOFT XIR has the same effect as other XIRs. Once set the bit remains set until software clears it. This allow software to discover what caused a previous XIR.

17.2.6.3 Error Reset

None

17.2.6.4 Wake-up Reset

Compatibility Note – There is no Wakeup Reset support for power management, unlike that in prior UltraSPARC-based systems.

UltraSPARC IIi, in common with UltraSPARC, can enter power-down mode by executing a SHUTDOWN instruction but refresh is stopped in this condition. Providing a reset is the only way to leave power-down mode and resume normal operation but UltraSPARC IIi does not automatically generate this reset.

17.2.7 Effects of Resets

The effects of Resets are visible to software. Reset operation also provides sequencing to ensure proper hardware operation. For example, all busses are tristated at power up.

17.2.7.1 Major Activities as a Function of Reset

Table 17-1 Effects of Resets

Reset Sources	Bit Set	Mem. Refresh[2]	Reset PCI Devices	Reset UPA64S	Effect on UltraSPARC IIi CPU/PCI
POWER_OK	POR	Disable	Yes	Yes	POR
Push-button POR	B_POR	NC	Yes	Yes	POR
Push-button XIR[1]	B_XIR	NC	No	No	XIR
Soft POR	SOFT_POR	NC	Yes	Yes	POR
Soft XIR	SOFT_XIR	NC	No	No	XIR

1. causes jump to XIR trap vector
2. NC = No Change.

17.2.7.2 Bus Conditions at Power up

UPA64S Address Bus

This bus is always driven

UPA64S 64 bit Data Bus

This bus is shared by the UPA64S (graphics) interface and the memory transceiver ICs and it tristates on POR. The Fast Frame Buffer (FFB) ICs asynchronously tristate their data busses at reset.

Memory Data Bus

Driven by DRAM and the memory XCVR chips. The RAS* and CAS* signals driven by UltraSPARC IIi are asynchronously deasserted. UltraSPARC IIi cause the XCVR to tristate its data output pins during reset.

PCI

UltraSPARC IIi IO asynchronously tristates this bus. It also asynchronously deasserts control signals.

17.2.7.3 Reset_Control Register (0x1FE.0000.F020)

The UltraSPARC IIi Reset_Control indicates the source of a reset and provides control of software reset generation.

Table 17-2 Reset_Control Register

Field	Bits	Value	Description	Type
Reserved	63:32	0	Reserved	R0
POR	31	*[1]	Set if the last reset was due to the assertion of Sys_Reset_L	R/W1C
SOFT_POR	30	*	Setting to 1 causes a POR reset; stays set until software clears it	R/W
SOFT_XIR	29	*	Setting to 1 causes an XIR trap; stays set until software clears it	R/W
B_POR	28	*	Set if the last reset was due to the assertion of P_Reset_L	R/W1C
B_XIR	27	*	Set if the last reset was due to the assertion of an X_Reset_L	R/W1C
Reserved	26:0	0	Reserved	R0

1. The highest priority reset source has its bit set. Only the bits marked with “*” are set.

Only one of the reset bits is set. If multiple resets occur simultaneously, the following priority order is used:

1. POR
2. B_POR
3. SOFT_POR
4. B_XIR
5. SOFT_XIR

POR - Power On Reset This bit is set if the last reset was due to the assertion of SYS_RESET_L pin and occurs whenever the machine power cycles.

SOFT_POR - Soft Power On Reset Writing a 1 to this bit has the same effect as power-on reset, except that a different status bit in the Reset_Control Register is set. Memory refresh is not affected. Writing a 0 to this bit clears it and has no other effect.

SOFT_XIR - Soft Externally Initiated Reset Writing a 1 to this bit causes the UltraSPARC IIi to send a XIR trap to the UltraSPARC IIi core. Writing a 0 to this bit clears it and has no other effect.

B_POR - Button Reset This bit is set as a result of a "button" reset which is caused by an external switch and the subsequent assertion of the P_RESET_L pin. It can also be caused by scan in the RIC chip. Memory refresh is not affected. The actions and results of this reset are identical to that of Power-on Reset, except for a different status bit being set.

B_XIR - XIR Button Reset This bit is set as a result of a "button" XIR Reset caused by an external switch asserting the X_RESET_L signal pin. This bit can also be set by scan in the RIC chip. The actions and results of this reset are identical to that of SOFT_XIR, except that a different status bit is set.

17.3 RED_state

17.3.1 Description of RED_state

RED_state is an acronym for Reset, Error, and Debug State. It serves two mutually exclusive purposes:

- Indication, during trap processing, that there are no more available trap levels—that is, if another nested trap is taken, the processor will enter error_state and halt. RED_state provides system software with a restricted execution environment
- Provision of an execution environment for all reset processing

This state is entered under any of the occurrences:

- Trap taken when TL = MAXTL - 1
- Reset requests: POR, XIR, WDR
- Reset request: SIR *if* TL < MAXTL (If TL = MAXTL, the processor enters error_state)
- Implementation-dependent trap, *internal_processor_error* exception, or *catastrophic_error* exception
- Setting of PSTATE.RED by system software

RED_state is indicated by the PSTATE.RED bit being set, regardless of the value of TL. Executing a DONE or RETRY instruction in RED_state restores the stacked copy of the PSTATE register,which clears the PSTATE.RED flag if it was cleared for the stacked copy. System software can also set or clear the PSTATE.RED flag with a WRPR instruction, which also forces the processor to enter or exit RED_state respectively. In this case, the WRPR instruction should be placed in the delay slot of a jump, so that the PC can be changed in concert with the state change.

Note – Setting TL = MAXTL using a WRPR instruction neither sets RED_state nor alters any other machine state. Ther values of RED_state and TL are independent.

A reset or trap that sets PSTATE.RED (including a trap in RED_state) clears the LSU_Control_Register, including the enable bits for the I-cache, D-cache, I-MMU, D-MMU, and virtual and physical watchpoints.

The default access in RED_state is noncacheable, so the system must contain some noncacheable scratch memory. The D-cache, watchpoints, and D-MMU can be enabled by software in RED_state, but any trap that occurs will disable them again. The I-MMU and consequently the I-cache are always disabled in RED_state. This overrides the enable bits in the LSU_Control_Register.

When PSTATE.RED is explicitly set by a software write, there are no side effects other than disabling the I-MMU. Software may need to create the effects that are normally created when resets or traps cause the entry to RED_state.

The caches continue to snoop and maintain coherence if DVMA or other processors are still issuing cacheable accesses.

Note – Exiting RED_state by writing 0 to PSTATE.RED in the delay slot of a JMPL is not recommended. A noncacheable instruction prefetch may be made to the JMPL target, which may be in a cacheable memory area. This may result in a bus error on some systems, which will cause an *instruction_access_error* trap. The trap can be masked by setting the NCEEN bit in the ESTATE_ERR_EN Register to zero, but this will mask all non-correctable error checking. Exiting RED_state with DONE or RETRY will avoid this problem.

Note – While in RED_state, the Return Address Stack (RAS) is still active, and instruction fetches following JMPL, RETURN, DONE, or RETRY instructions use the address from the top of the RAS. Unless it is re-initialized with a series of CALLs, the RAS contains virtual addresses obtained prior to entry into RED_state. When these are passed through the now disabled I-MMU, invalid addresses may result. Note that this effect includes the predicted use of these four instructions. If such accesses cannot be tolerated, software should fill the RAS with valid addresses using CALL instructions before using a JMPL, RETURN, DONE, or RETRY instruction in RED_state. Note that the RAS is cleared after Power-on Reset. Section 21.2.10, *Return Address Stack (RAS)* on page 335 discusses the RAS in detail. The following code fragment fills the RAS with valid addresses:

```
mov   %o7,%g1
set   4,%g2
1:call 2f
subcc %g2,1,%g2
2:bnz 1b
mov %g1,%o7
```

There are other cases that use RAS for prefetch. For instance, immediately after writing to the LSU control register to enable the IMMU. The RAS should be initialized for this case as well.

Be sure there are no JMPs in the initial trap address tables. Software should use branch instructions to go to an area where the RAS can be initialized, before using a JMP to get a long displacement.

17.3.2 RED_state Trap Vector

When a SPARC-V9 processor processes a reset or trap that enters RED_state, it takes a trap at an offset relative to the RED_state_trap_ vector base address (RSTVaddr). The trap offset depends on the type of RED mode trap and takes the values:

- POR 0x20
- EIR 0x30
- TL5 0x40
- SIR 0x80
- other 0x50

in UltraSPARC IIi the RSTV base address is given in *Table 17-3*.

Table 17-3 RSTV Base Address

Virtual Address$_{16}$	Equivalent Physical Address$_{16}$ PA[40:0]
FFFF FFFF F000 0000	1FF F000 0000

UltraSPARC IIi has a pin to select a second RSTV to allow use of PC compatible SuperIO chips on a PCI bus. The second RSTV base address is given in *Table 17-4*.

Table 17-4 Second RSTV Base Address

Virtual Address$_{16}$	Equivalent Physical Address$_{16}$ PA[40:0]
FFFF FFFF FFFF 0000	1FF FFFF 0000

17.4 Machine State after Reset and in RED_state

Table 17-5 shows core CPU state created as a result of any reset, or after entering RED_state. See Section 6.4, *Summary of CSRs mapped to the Noncacheable address space* on page 47 for pointers to the reset state of the MCU and PCI areas.

Table 17-5 Machine State After Reset and in RED_state

Name	Fields	POR	WDR	XIR	SIR	RED_state‡
Integer registers		Unknown	Unchanged			
Floating Point registers		Unknown	Unchanged			
RSTV value		VA=FFFF FFFF F000 0000$_{16}$, PA=1FF F000 0000$_{16}$ VA=FFFF.FFFF.FFFF.00$_{16}$; PA=1FF.FFFF.00$_{16}$nn				
PC nPC		RSTV | 20$_{16}$ RSTV | 24$_{16}$	RSTV | 40$_{16}$ RSTV | 44$_{16}$	RSTV | 60$_{16}$ RSTV | 64$_{16}$	RSTV | 80$_{16}$ RSTV | 84$_{16}$	RSTV | A0$_{16}$ RSTV | A4$_{16}$
PSTATE	MM	0 (TSO)				
	RED	1 (RED_state)				
	PEF	1 (FPU on)				
	AM	0 (Full 64-bit address)				
	PRIV	1 (Privileged mode)				
	IE	0 (Disable interrupts)				
	AG	1 (Alternate globals selected)				
	CLE	0 (current little endian)				
	TLE	0 (trap little endian)				
	IG	0 (Interrupt globals not selected)				
	MG	0 (MMU globals not selected)				
TBA<63:15>		Unknown	Unchanged			
Y		Unknown	Unchanged			

Table 17-5 Machine State After Reset and in RED_state *(Continued)*

Name	Fields	POR	WDR	XIR	SIR	RED_state‡
PIL		Unknown	Unchanged			
CWP		Unknown	*Unchanged* except for register window traps			
TT[TL]		1	trap type	3	4	trap type
CCR		Unknown	Unchanged			
ASI		Unknown	Unchanged			
TL		MAXTL	**min**(TL+1, MAXTL)			
TPC[TL] TNPC[TL]		Unknown Unknown	PC nPC	PC Unknown	PC nPC	PC nPC
TSTATE	CCR ASI PSTATE CWP PC nPC	Unknown Unknown Unknown Unknown Unknown Unknown	CCR ASI PSTATE CWP PC nPC			
TICK	NPT counter	1 Restart at 0	Unchanged count	Unchanged Restart at 0	Unchanged count	
CANSAVE		Unknown	Unchanged			
CANRESTORE		Unknown	Unchanged			
OTHERWIN		Unknown	Unchanged			
CLEANWIN		Unknown	Unchanged			
WSTATE	OTHER NORMAL	Unknown Unknown	Unchanged Unchanged			
VER	MANUF IMPL MASK MAXTL MAXWIN	0017_{16} UltraSPARC-I=0010_{16} UltraSPARC-II=0011_{16} mask-dependent 5 7				
FSR	all	0	Unchanged			
FPRS	all	Unknown	Unchanged			
Non-SPARC-V9 ASRs						
SOFTINT		Unknown	Unchanged			
TICK_COMPARE	INT_DIS TICK_CMPR	1 (off) Unknown	Unchanged Unchanged			
PERF_CONTROL	S1 S0 UT (trace user) ST (trace system) PRIV (priv access)	Unknown Unknown Unknown Unknown Unknown	Unchanged Unchanged Unchanged Unchanged Unchanged			
PERF_COUNTER		Unknown	Unchanged			

Table 17-5 Machine State After Reset and in RED_state *(Continued)*

Name	Fields	POR	WDR	XIR	SIR	RED_state‡
GSR		Unknown	Unchanged			
Non-SPARC-V9 ASIs						
UPA_PORT_ID *	FC ECC_VALID ONEREAD PINT_RDQ PREQ_DQ PREQ_RQ UPACAP ID	FC_{16} 0 1 1 0 1 $1B_{16}$ TBD				
UPA_CONFIG	ELIM MID	0 0	Unchanged 0			
LSU_CONTROL	all	0 (off)	0 (off)			
DISPATCH CONTROL		0	Unchanged			
VA_WATCHPOINT		Unknown	Unchanged			
PA_WATCHPOINT		Unknown	Unchanged			
I-& D-MMU_SFSR,	ASI FT E CTXT PRIV W OW(overwrite) FV (SFSR valid)	Unknown Unknown Unknown Unknown Unknown Unknown Unknown 0	Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged Unchanged			
D-MMU_SFAR		Unknown	Unchanged			
UDBH_ERR, UDBL_ERR	UE CE E_SYNDR	Unknown Unknown Unknown	Unchanged Unchanged Unchanged			
UDBH_CONTROL, UDBL_CONTROL	FMODE FCBV	Unknown Unknown	Unchanged Unchanged			
INTR_DISPATCH	NACK BUSY	Unknown 0	Unchanged Unchanged			
INTR_RECEIVE	BUSY	0	Unchanged			
	MID	Unknown	Unchanged			
ESTATE_ERR_EN	ISAPEN (sys addr err) NCEEN (non CE) CEEN (CE)	0 (off) 0 (off) 0 (off)	Unchanged Unchanged Unchanged			
AFAR	PA	Unknown	Unchanged			
AFSR	all	Unchanged	Unchanged			

Table 17-5 Machine State After Reset and in RED_state *(Continued)*

Name	Fields	POR	WDR	XIR	SIR	RED_state[‡]
Other UltraSPARC IIi Specific States						
Processor and E-cache tags and data		Unknown	Unchanged			
Cache snooping		Enabled				
Instruction Buffers		Empty				
Load/Store Buffers, all outstanding accesses		Empty	Unchanged			Empty
iTLB, dTLB	Mappings E-bit (side-effect) NC-bit (noncacheable)	Unknown 1 1	Unchanged 1 1			
RAS	all	RSTV | 20_{16}	Unchanged			

‡ Processor states are updated according to this table only when RED_state is entered on a reset or trap. If software explicitly sets PSTATE.RED to 1, it must create the appropriate states itself.

CHAPTER 18

MCU Control and Status Registers

The MCU control and status registers program the operation of the shared memory and UPA64S interfaces which are integrated into the UltraSPARC IIi CPU.

Register Access

Register accesses should always be of eight bytes at a time. The physical addresses for the MCU control and status registers are shown in *Table 18-1*.

Table 18-1 MCU CSRs

PA	Register Name	Associated I/O Port
1FE.0000.F000	FFB_Config	FFB
1FE.0000.F010	Mem_Control0	Memory Control Unit
1FE.0000.F018	Mem_Control1	Memory Control Unit

Reads of any size up to eight byte to any register are supported regardless of whether reads of that size makes sense.

Writes of any size up to eight bytes are also supported regardless of whether writes of that size makes sense. Writes of any size *may* corrupt unwritten bits in the register. (that is., writes may result in all eight bytes being written regardless of the indicated write size.)

Software must ensure that only the proper-sized—that is, equal to the register size—accesses are used. No hardware checking is performed. Block (64-byte) access erroneously causes a UPA64S or PCI transaction with an undefined address.

Misaligned access due to not setting the “E” bit correctly in the TTE also yields unpredictable results.

Note – Register bits that are designated as read only (RO) are not affected by writes. No errors are reported. Fields with reserved definitions should not be used. Some combinations of bits are not valid. Disable refresh before changing memory control registers.

Compatibility Note – Prior UltraSPARC-based systems used other hardware and programming models to control the UPA and memory interfaces.

Reset

Memory Control registers are reset to their initial values only during PowerOnReset (POR). POR is often generated by logic connected to the POWER_OK signal from the power supply. Refreshing operates continuously during and after other reset conditions.

The SYS_RESET signal is asserted by the system POR condition. The P_RESET signal is asserted for many reasons and does not affect many of the memory control register bit values.

Table headings with POR mean the register is affected by SYS_RESET only. Table headings with Reset mean the register is affected by SYS_RESET or P_RESET.

Resets are described more fully in Section 17.2, *Resets* on page 252.

18.1 FFB_Config Register

Table 18-2 FFB_Config Register—0x1FE.0000.F000

Field	Bits	Description	Reset	Type
Reserved	63:28		0	RO
SPRQS	27:24	Slave P_request queue size. Initialize to max size in 2 Cycle Packets of the corresponding slave request queue.	1	R/W
Reserved	23:15		0	RO
Oneread	14	Always set to ‘one’;. UPA slave interface does not support multiple outstanding reads.	1	RO
Reserved	13:0		0	RO

The Data Queue Size is not tracked separately, and the UPA64S device must be able to receive 64 bytes per allowed outstanding request.

18.2 Mem_Control0 Register

Table 18-3 Mem_Control0 Register—0x1FE.0000.F010

Field	Bits	Description	POR	Type
Reserved	63:32		0	RO
RefEnable	31	Refresh enable	0	R/W
Reserved	30:29		0	RO
ECCEnable	28	Enable all ECC functions	0	R/W
Trace_Delay	27	SME1430: short trace enable	0	R/W
Reserved	27	SME1040	0	R/W
FFBwrToDRAMrdDly	26:25	SME1430: memdata bus turnaround	0	R/W
Reserved	24:13	SME1430		RO
Reserved	26:13	SME1040	0	RO
11-bit Column Address	12	Enables 11-bit column address mode.	0	R/W
DIMMPairPresent<3:0>	11:8	Determines which DIMM pairs to refresh.	0xF	R/W
RefInterval<7:0>	7:0	Interval between refreshes. Each encoding is 32 processor clocks	0x30	R/W

ECCEnable

This instruction enables the MCU to perform single-bit detect and correct, and notification of single or multi-bit errors to the ECU and PBM, for possible logging and trap/interrupt generation. In general this should always be set to 1, unless DIMMs that do not support check bits are used.

There are further enables for ECC related trap and interrupt generation in the ECU and PBM. See Section 16.6.1, *E-cache Error Enable Register* on page 242 and DMA UE/CE interrupt mapping registers in *Partial Interrupt Mapping Registers* on page 303 and ERRINT_EN in *PCI Control/Status Register* on page 283.

Trace Delay (SME1430 only)

Trace_Delay is defined in mem_control1<27>. Set to 1 for SME1040 type timing. Set to 0 for shorter trace delays. This bit defaults to 0 at power-up. The impact on programming is shown in the Mem_Control1 Register RCD field—see page 274 and the FFBwrToDRAMrdDly section below.

FFBwrToDRAMrdDly (SME1430 only)

FFBwrToDRAMrdDly is defined in mem_control1<26:25>. An FFB write operation followed by a DRAM read operation is optimized by using this 2-bit field. This bit field value is adjusted according to CPU clock rate and DRAM speed.

FFBwrToDRAMrdDly value adjusts the delay in the initiation of a DRAM read after an FFB write operation so that there is no collision on the memory bus between data returning from DRAM and data written to the FFB. The value "11" gives the smallest delay whereas the value "00" gives the largest delay. *Table 18-4* shows the use of FFBwrToDRAMrdDly for various configurations of processor frequency and setting of CASRW.

Table 18-4 Use of FFBwrToDRAMrdDly

Processor Frequency	CASRW	Trace_Delay=0	Trace_Delay=1
360	4	01[1]	na[2]
	5	01	10
400	4	11[1]	na
	5	11	11
440	5	10[1]	na
	6	10	10
480	5	10[1]	na
	6	11	11

1. requires 50 ns DRAMs
2. not applicable or valid

RefEnable

Main memory is composed of dynamic RAMs, which require periodic "refreshing" to maintain the contents of the memory cells. RefEnable == 1 is used to enable refresh of main memory. RefEnable == 0 disables refresh in memory.

POR is the only reset condition that clears RefEnable (and initializes the rest of the Mem_Control0/1). SOFT_POR, B_POR, B_XIR, and SOFT_XIR leave RefEnable unchanged and refresh continues normally.

Any refresh operation in progress is aborted at the time of clearing this bit. The truncated memory signals in this case could lead to loss of data.

11-bit Column Address

The default memory addressing only supports 10-bit column address DRAMs. An additional mode was added to support a 11-bit column address. Since the total available address bits in the memory controller is constant (1 Gbyte max. addressable), the maximum number of DIMM pairs in this mode is cut in half. See Chapter 7, *UltraSPARC IIi Memory System*.

DIMMPairPresent<3:0>

Indicates the presence/absence of DIMMS to enable performance degradation caused by refreshing unpopulated DIMMs to be eliminated. One bit position corresponds to each DIMM bank. "DIMM pair 0" corresponds to "DimmPairPresent[0]," and so on. A zero indicates not present, a 1 indicates present. These bits are set by software after probing. Note that in 11-bit Column Address mode, only DIMM Pair 0 and 2 can be marked present. Pairs 1 and 3 should always be marked not present when in 11-bit column addressing mode.

Note – Refresh must be disabled first by clearing the RefEnable bit before changing the memory controller register values. Refresh may be enabled again simultaneously with writing DIMMPairPresent and RefInterval. Failure to follow this rule may result in unpredictable behavior.

Table 18-5 Various Memory Configurations

DIMM size	Base device	No. of devices	System memory min/max config
8 MB	1M x 4	18	16 MB/64 MB
16 MB	2M x 8	9	32 MB/128 MB
32 MB	4M x 4	18	64 MB/256 MB
64 MB	4M x 4(banked)	36	128 MB/512 MB
64 MB	8M x 8	9	128 MB/512 MB

Table 18-5 Various Memory Configurations *(Continued)*

DIMM size	Base device	No. of devices	System memory min/max config
128 MB	8M x 8(banked)	18	256 MB/1 GB
128 MB	16M x 4	18	256 MB/1 GB
256 MB	16M x 4(banked)	36	512 MB/1 GB

RefInterval

RefInterval specifies the interval time between refreshes, in quanta of 32 CPU clocks. SW should program RefInterval according to *Table 18-6*. Values given are in hexadecimal and derived from this formula:

$$refValue = \frac{refreshPeriod}{numberOfRows \times ClockPeriod \times 32 \times numberOfPairs}$$

that is: (32 * frequency * 1000) / (2048 * 32 * DIMM pairs).

This data is based on using 16 MB(2048 rows/32ms) EDO drams only; this configuration matches the composite DIMM specification. See *Table 18-6*.

Table 18-6 Refresh Period (in 32x CPU clock periods) as a Function of Frequency

Total DIMM pairs enabled	CPU Frequency in MHz											
	125–166	167–200	201–224	225–250	251–270	271–300	301–330	331–360	361–400	401–440	441–450	451–480
1	0x51	0x61	0x6C	0x7A	0x83	0x92	0xA1	0xAF	0xC3	0xD6	0xDB	0xEA
2	0x28	0x30	0x37	0x3D	0x41	0x49	0x50	0x57	0x61	0x6B	0x6D	0x75
3	0x1B	0x20	0x25	0x28	0x2B	0x30	0x35	0x3A	0x41	0x47	0x49	0x4E
4	0x14	0x18	0x1D	0x1E	0x20	0x24	0x28	0x2B	0x30	0x35	0x36	0x3A

18.3 Mem_Control1 Register

Memory Control Register 1 contains fields that control the read, write, and refresh timing for the DRAM DIMMs. They allow software to optimize the memory access timing for a particular system frequency.

The contents of Memory Control Register 1 can be changed as required by an electrical tuning of memory timing often based on SPICE analysis. The Mem_Control1 register bits are listed in *Table 18-7*.

Table 18-7 Mem_Control1 Register—0x1FE.0000.F018

Field	Bits	POR State	Description	Type
Reserved	63:32	0	reserved. Read as zero, write 0	RO
AMDC<3>	31	0	SME1430 Advance Memdata Clock	R/W
ARDC<3>	30	0	SME1430 Advance Read Data Clock	R/W
Reserved	31:30	0	SME1040 reserved; read as zero, write 0	RO
AMDC<2:0>	29:27	0	Advance Memdata Clock	R/W
ARDC<2:0>	26:24	0	Advance Read Data Clock	R/W
CSR[1]	23:21	2	CAS to RAS delay for CBR refresh cycles.	R/W
CASRW[1, 2]	20:18	2	CAS length for read/write	R/W
RCD[1]	17:15	4	RAS to CAS Delay	R/W
CP	14:12	2	CAS Precharge	R/W
RP[1]	11:9	4	RAS Precharge	R/W
RAS[1]	8:6	5	Length of RAS for Refresh	R/W
CASRW[1, 2]	5:3	2	Must be same as 20:18	R/W
RSC[2]	2:0	0	RAS after CAS hold time	R/W

1. Bit definitions differ between SME1430 and SME1040. See bit-definition sections that follow
2. Originally had separate fields for CAS during reads and CAS during writes. However, memory timing is optimal if writes and reads use the same CAS width. Additionally, an errata caused the read CAS width to be used in one part of the write control logic. Both fields are now given the same name, and must be programmed to the same value. Results are undefined if they are different.

Even though many bits are set by reset, it is good practice to have the boot PROM program all memory control register bits when initializing the system.

AMDC- Advance Memdata Clock

This instruction moves the relative timing between a transceiver clock transition and the point at which the processor latches read data driven by that transceiver (using the MEMDATA bus)

This timing adjustment allows for earlier data clocking for slower clock cycles. (advance) or for later data clocking for fast clock cycles.

Delaying this clocking by a cycle (relative to the recommended values) may be useful if timing is critical but it reduces hold time margin.

Table 18-8 AMDC Timing Arguments—Mem_Control1<31>,<29:27>

Argument	SME1040	SME1430	Timing
0000	0	0	default MemData clocking
0001	1	1	CPU clocks of sampling delay
0010	2	2	CPU clocks of sampling delay
0011	3	3	CPU clocks of sampling delay
0100	4	4	CPU clocks of sampling advancement
0101	3	3	CPU clocks of sampling advancement
0110	2	2	CPU clocks of sampling advancement
0111	1	1	CPU clocks of sampling advancement
1000–1010	reserved	reserved	
1011	reserved	5	CPU clocks of sampling advancement
1100–1111	reserved	reserved	

ARDC- Advance Read Data Clock

Maintaining a minimum EDO DRAM CAS cycle is difficult if the DIMM loading is widely variable. Light loading on the CAS and DATA lines can make the data disappear before it is clocked and produce a hold-time problem.

The system board reference design specifies buffering to make the RAS/CAS/WE delays independent of the number of DIMMs in circuit. However, the ADDR and DATA delays do vary with DIMM population.

If necessary, this field can be used to advance the clock that latches read data in the transceivers. This may be necessary when only one or two DIMM pairs are populated. It can also be used to delay the clock for heavily loaded DIMM populations.

Current simulations indicate that the ARDC value need not be varied for the supported range and combinations of DIMM configurations.

Table 18-9 ARDC Timing Arguments—Mem_Control1<30>, <26:24>

Argument	SME1040	SME1430	Timing
0000	0	0	default DRAM Read data clocking based on CAS assertion time
0001	1	1	CPU clocks of sampling delay
0010	2	2	CPU clocks of sampling delay

Table 18-9 ARDC Timing Arguments—Mem_Control1<30>, <26:24> *(Continued)*

Argument	SME1040	SME1430	Timing
0011	3	3	CPU clocks of sampling delay
0100	4	4	CPU clocks of sampling advancement
0101	3	3	CPU clocks of sampling advancement
0110	2	2	CPU clocks of sampling advancement
0111	1	1	CPU clocks of sampling advancement
1000	reserved	8	CPU clocks of sampling advancement
1001	reserved	7	CPU clocks of sampling advancement
1010	reserved	6	CPU clocks of sampling advancement
1011	reserved	5	CPU clocks of sampling advancement
1100–1111	reserved	reserved	

CSR - CAS before RAS delay timing

This Instruction controls the CAS assertion to RAS assertion delay for CAS before RAS (CBR) refresh cycles

Table 18-10 CSR Delay Timing—Mem_Control1<23:21>

Argument	SME1040	SME1430	Timing
000	3	3	CPU clocks between CAS and RAS assertions
001	4	4	CPU clocks between CAS and RAS assertions
010	5	5	CPU clocks between CAS and RAS assertions
011	6	6	CPU clocks between CAS and RAS assertions
100	7	7	CPU clocks between CAS and RAS assertions
101	8	8	CPU clocks between CAS and RAS assertions
110	reserved	2	CPU clocks between CAS and RAS assertions
111	reserved	reserved	

CASRW- CAS assertion for read/write cycles

CASRW controls the minimum CAS assertion time for reads and writes.

There were originally separate fields for CAS during reads and CAS during writes. However, memory timing is optimal if writes and reads use the same CAS width. Additionally, an erratum caused the read CAS width to be used in one part of the write control logic. Both fields are now given the same name, and must be programmed to the same value. Results are undefined if they are different.

Table 18-11 CASRW Assertion Time—Mem_Control1<20:18> (same settings at <5:3>)

Argument	SME1040	SME1430	Timing
000	3	reserved	clocks for which CAS is in low state
001	4	4	clocks for which CAS is in low state
010	5	5	clocks for which CAS is in low state
011	reserved	6	clocks for which CAS is in low state
100–111	reserved	reserved	

RCD - RAS to CAS Delay

RCD controls the RAS to CAS delay during the initial part of the read or write memory cycle.

In the SME1430, the RCD operation is determined by the RCD and Trace_Delay (Mem_Control1<27>) values.

Table 18-12 RCD Delay Timing—Mem_Control1<17:15>

Argument	SME1040	SME1430		Timing
		Trace_Delay=1	Trace_Delay=0	
000	6	13	13	CPU clocks between RAS and CAS assertion
001	7	17	17	
010	8	20	19	
011	11	11	12	
100	12	19	18	
101	14	14	14	
110	15	15	15	
111	reserved	reserved	reserved	

CP - CAS Precharge

CP controls the CAS precharge time in between page cycles. The argument in this field must equal the common argument in each CASRW field. (Mem_Control1<20:18> and Mem_Control1<5:3>)

Table 18-13 CP – CAS Precharge Time—Mem_Control1<14:12>

Argument	SME1040	SME1430	Timing
000	3	reserved	CPU clocks of CAS Precharge
001	4	4	CPU clocks of CAS Precharge
010	5	5	CPU clocks of CAS Precharge
011	reserved	6	
100–111	reserved	reserved	

RP - RAS Precharge

RP controls the RAS precharge time between memory cycles.

Table 18-14 RP Timing—Mem_Control1<11:9>

Argument	SME1040	SME1430	Timing
000	8	12	
001	9	13	
010	10	21	
011	11	17	CPU clocks of RAS Precharge
100	12	19	
101	14	14	
110	15	15	
111	reserved	16	

RAS

RAS is used to control the length of time that RAS is asserted during refresh cycles.

Table 18-15 RAS Duration Time—Mem_Control1<8:6>

Argument	SME1040	SME1430	Timing
000	13	20	CPU clocks of RAS assertion.
001	15	15	
010	18	18	
011	22	22	
100	23	23	
101	24	24	
110	reserved	27	
111	reserved	31	

RSC-RAS after CAS delay timing

RSC controls time to deassert RAS after CAS at the end of a memory cycle.

Table 18-16 RSC – RAS Deassert Time—Mem_Control1<2:0>

Argument	SME1040	SME1430	Timing
000	4	4	CPU clocks RAS asserted after CAS
001	5	5	CPU clocks RAS asserted after CAS
010	6	6	CPU clocks RAS asserted after CAS
011	7	7	CPU clocks RAS asserted after CAS
100	8	8	CPU clocks RAS asserted after CAS
101	9	9	CPU clocks RAS asserted after CAS
110	reserved	10	CPU clocks RAS asserted after CAS
111	reserved	reserved	

18.4 Programming Mem_Control1

Table 18-17 gives program values to support one, two, three, or four DIMM pairs, with one or two banks of DRAM on each DIMM. These values are given as a function of the internal CPU operating frequency.

These tabulated values depend upon the conditions:

- The system board meeting the min/max delay specifications for RAS/CAS/MEMADDR/DATA/MEMDATA, and all transceiver control and clock signals;
- The design specifications for max skew between RAS/CAS/MEMADDR/DATA being met.
- The specified DIMMs being used. (buffered CAS/WE/ADDR)

Memory Control Register programming may also be used to utilize memory subsystems whose performance lies outside the suggested design specifications.

Because all skew and hold time relationships for the DRAMs are not programmable, it is recommended that all designs meet the etch length specifications and employ DIMMs that meet the composite specification.

It is possible that alternate values may give higher performance from 50 ns DRAM. The minimum CAS cycle with this programming is 26.5 ns (13.25 ns CAS assertion) at 300 Mhz.

Table 18-17 Mem_Control1 hexadecimal values as a function of CPU frequency

CPU Frequency in MHz	SME1040[2]		SME1430[3]	
	60 ns DRAM	50 ns DRAM	60 ns DRAM	50 ns DRAM
480			5E28.25D3	4E69.AF93
450 to 479			544E.B9DA	4E69.A792
401 to 449			544C.B9DD	410A.AD54
361 to 400			0C4B.2794	0005.92CB
334 to 360	0C4A.AB14	0645.9ACB	0C4A.AB14	0645.9ACB
271 to 333	0645.9ACB	0645.9ACB	–	–
225 to 270	0626.168A	0626.168A	–	–
167 to 224	3800.8241	3800.8241	–	–
125 to 166	3E00.8000	3E00.8000	–	–
0-1[1]	3D00.0000	5D00.0000	–	–

1. This programming is included for emulation. The PLLs should be bypassed, and an external means of supplying DRAM refresh should be provided.
2. Excepting the 334 to 360 MHz range, SME1040 programming for 50 and 60 ns DRAMs is currently the same. Either DRAM rating can be used with the same performance.
3. Speeds above 440 MHz have not been tested.

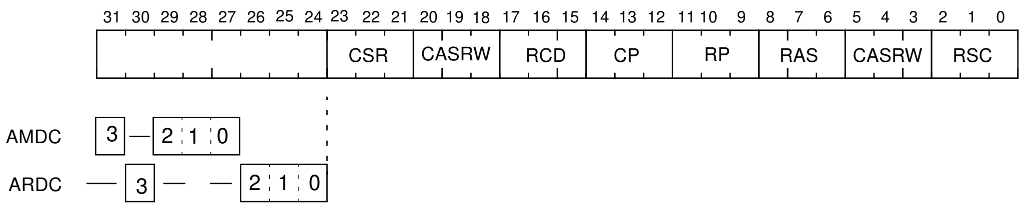


Figure 18-1 Mem_Control1 Register Bit Allocation

Initialization of the Mem_Control registers should be performed in accordance with the probing algorithm described in Section A.10, *Memory Probing and Initialization* on page 383.

Note – The Mem_Control register must be initialized before *any* memory operation, including refresh. Before modifying the register, software must complete and inhibit all memory references and disable refresh. Wait 100 clock periods after disabling refresh to guarantee completion of any refresh in progress.

18.5 UPA Configuration Register

The UPA_CONFIG register can be accessed at ASI 0x4A, VA==0. This is a 64-bit register; non-64-bit aligned accesses cause a *mem_address_not_aligned* trap.

Much of the UltraSPARC-I and UltraSPARC-II functionality in this register is removed.

UltraSPARC IIi uses a register in the Memory Control Unit to restrict the number of outstanding UP64S slave requests, instead of this register.

The new ELIM field is copied from UltraSPARC-II.

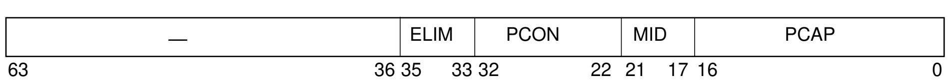


Figure 18-2 UPA_CONFIG Register Format

ELIM: This field can be used to zero upper bits of the E-cache tag address, if more address pins are used on the tag RAM than necessary. It can also be used to force the use of a smaller E-cache size than is supplied with the UltraSPARC IIi system.

Resets to 000. Must be set to a size not bigger than the E-cache data SRAMs provide, otherwise incorrect E-cache operation will result.

000 has no effect on the E-cache tag address.

111 and 110 zero the 3 MSBs to create a 256-kilobyte E-cache, regardless of the SRAM size or connections to the E-tag.

101 allows a 512-kilobyte E-cache, if the SRAMs used are sized appropriately Otherwise, the E-cache is the size allowed by the SRAMs.

100 allows a 1-Mbyte E-cache

011 allows a 2-Mbyte E-cache, the largest supported by UltraSPARC IIi

Behavior for other encodings is Reserved.

PCON[7:0]: Unused on UltraSPARC IIi; Read as 0

MID[4:0]: Module (processor) ID register; Read as 0

PCAP[16:0]: Read as 0 on UltraSPARC IIi

Resets are described more fully in Section 17.

CHAPTER **19**

UltraSPARC IIi PCI Control and Status

19.1 Terms and Abbreviations Used

R -Read only

R0 -Read zero always

W -Write only

R/W -Read / Write

R/W1C -Read / Write with 1 to clear

In this section, unless otherwise noted, all references to UltraSPARC IIi and its registers refer to UltraSPARC IIi's functional IO, as opposed to the UltraSPARC IIi core. The term UltraSPARC IIi IO is sometimes used to emphasize this point.

Caution – Registers that are designated write only may be read, but the data returned is *undefined*. and no error is reported for the access. Software should never rely on the value returned. Writes to read only registers also have no effect with no error reported.

19.2 Access Restrictions

Register accesses to UltraSPARC IIi IO can be in any size from one byte to 8 bytes. Sizes and locations for the registers are given in the following sections.

Reads of any size up to 8 bytes to any register are supported regardless of whether reads of that size makes sense. Writes of any size up to 8 bytes are also supported regardless of whether writes of that size makes sense. Writes of any size *may* corrupt unwritten bits in the register (that is, writes may result in all 8 bytes being written regardless of the indicated write size).

Software must ensure that only the proper sized accesses are used. No hardware checking is performed. Block (64 byte) access to UltraSPARC IIi IO registers cause a PCI or UPA64S transaction to an unspecified address.

Misaligned access due to not correctly setting the "E" bit in the TTE also yields unpredictable results.

19.3 PCI Bus Module Registers

These registers control aspects of UltraSPARC IIi's PCI operations that are not defined by the PCI specification. The registers defined by the PCI specification are listed in *Table 19-12* on page 289.

Table 19-1 PBM Registers

Register	PA	Access Size
PCI Control/Status Register	0x1FE.0000.2000	8 bytes
PCI PIO Write AFSR	0x1FE.0000.2010	8 bytes
PCI PIO Write AFAR	0x1FE.0000.2018	8 bytes
PCI Diagnostic Register	0x1FE.0000.2020	8 bytes
PCI Target Address Space Register	0x1FE.0000.2028	8 bytes
PCI DMA Write Synchronization Register	0x1FE.0000.1C20	8 bytes
PIO Data Buffer Diagnostics Access	0x1FE.0000.5000 - 0x1FE.0000.5038	8 bytes
DMA Data Buffer Diagnostics Access	0x1FE.0000.5100 - 0x1FE.0000.5138	8 bytes
DMA Data Buffer Diagnostics Access (72:64)	0x1FE.0000.51C0	8 bytes

Compatibility Note – APB has a similar additional state for each of its PCI busses. See the APB User's Manual for details.

Note – The bit definitions that follow assume "big-endian" type accesses.

19.3.0.1 PCI Control/Status Register

Table 19-2 PCI Control and Status Register

Field	Bits	Description	POR state	RW
Reserved	63:37	Reserved, read as 0	0	R0
PCI_MRLM_EN	36	1 = enable the generation of PCI Memory Read Line for Block loads, and Memory Read Multiple for 8 byte loads and noncacheable instruction fetch. 0 = force use of PCI Memory Read for all PIO reads. 1 provides a performance benefit due to APB prefetch capability for these commands	0	RW
Reserved	35	Read as 0	0	R0
PCI_SERR	34	Set when SERR# signal is asserted on the PCI bus	0	R/ W1C
Reserved	33:22	Reserved, Read as 0,	0	R0
ARB_PARK	21	PCI bus arbitration parking enable. 0 = UltraSPARC IIi parks when idle 1 = previous bus owner parked (including UltraSPARC IIi)	0	RW
CPU_PRIO[1]	20	UltraSPARC IIi arbitration priority 0 = no extra priority for CPU 1 = CPU will be granted every other bus cycle if requested.	0	RW
ARB_PRIO[1]	19:16	Slot arbitration priority (1 bit per slot) 0 = no extra priority 1 = slot will be granted every other bus cycle if requested.	0	RW
Reserved	15:9	Reserved, read as 0.	0	R0
ERRINT_EN	8	Enable PCI error interrupt. 0 = PCI error interrupt disabled 1 = PCI error interrupt enabled	0	RW

Table 19-2 PCI Control and Status Register *(Continued)*

Field	Bits	Description	POR state	RW
RETRY_WAIT_EN	7	Two flow control mechanisms exist for DMA. 1 = Retry if a prior DMA write is still completing. 0 = Wait if possible (some cases still retry because of unavailability of address registers). Because of the inability to provide fairness with the retry protocol, overall system performance is generally better with 0.	0	RW
Reserved	6:4	Reserved, read as 0.	0	R0
ARB_EN<3:0>	3:0	PCI arbitration enable. One independent bit for each supported device on the bus. 0 = Bus requests from corresponding PCI device are ignored 1 = Bus requests from corresponding PCI device are honored.	0	RW

1. Software must ensure that at most one bit of {CPU_PRIO, ARB_PRIO[3:0]} is set to 1. The result of setting multiple bits is undefined and can potentially result in some PCI devices being unfairly starved.

Recommended value is 0x10.0020.0101 for systems, using APB:

- PCI_MRLM_EN==1
- PCI_SERR==0
- ARB_PARK==1
- CPU_PRIO=0
- ARB_PRIO=0
- ERRINT_EN=1
- RETRY_WAIT_EN=0
- ARB_EN=1

19.3.0.2 PCI PIO Write Asynchronous Fault Status/Address Registers

The PCI PIO Write AFSR/AFARs record error information related to PIO writes to PCI slave devices. Only asynchronous errors reported through interrupts are recorded in these registers. Asynchronous errors include any PIO write access terminated by Master Abort, Target Abort, or excessive retries, as well as any PIO write during which a parity error was signaled on the PCI bus.

Although status bits for Master Abort, Target Abort and Parity Error exist in the PCI Configuration Registers for each PBM, they are duplicated in these registers to allow software to identify the chronological order of multiple errors and to associate an address with each one.

This register contains primary error status bits <63:60> and secondary error status bits <59:56>.*Only* one of the primary error status bits can be set at any time. Primary error status can be set only when

- None of the primary error conditions exists prior to this error *or*
- A new error is detected at the same time as software is clearing the primary error; "at the same time" means on coincident clock cycles. Setting takes precedence over clearing.

Secondary bits are set whenever a primary bit is set. The secondary bits are cumulative and always indicate that information has been lost because no address information has been captured. Setting of the primary error bits is independent.

The AFAR and bits <47:37> of AFSR log the address and status of the primary PCI PIO error. A new PCI PIO error is not logged into these bits until software clears the primary error to make the AFAR and part of the AFSR available for logging the new error.

Table 19-3 PCI PIO Write AFSR

Field	Bits	Description	POR state	RW
P_MA	63	Set if primary error detected is Master Abort	0	R/W1C
P_TA	62	Set if primary error detected is Target Abort	0	R/W1C
P_RTRY	61	Set if primary error detected is excessive retries	0	R/W1C
P_PERR	60	Set if primary error detected is parity error	0	R/W1C
S_MA	59	Set if secondary error detected is Master Abort	0	R/W1C
S_TA	58	Set if secondary error detected is Target Abort	0	R/W1C
S_RTRY	57	Set if secondary error detected is excessive retries	0	R/W1C
S_PERR	56	Set if secondary error detected is parity error	0	R/W1C
Reserved	55:48	Reserved, read as 0	0	R0
BYTEMASK	47:32	47:40 are always 0. 39:32 map identify the bytes stored, modulo 8 bytes. Bit 32 is byte 0.	0	R
BLK	31	Set to 1 if failed primary transfer was a block write	0	R
Reserved	30:0	Reserved, read as 0	0	R0

An interrupt is generated whenever

- a primary error is logged, and
- the PBM Error Interrupt is enabled by its mapping register, and
- ERRINT_EN is set in the PCI Control/Status Register

Note – The logged PA may point to the error PA + 4, if the PIO write is more than 4 bytes and the error is not on the last data beat of the PCI transaction.

Table 19-4 PCI PIO Write AFAR

Field	Bits	Description	POR state	RW
Reserved	63:41	Reserved, read as 0.	0	R0
PA	40:2	Physical address of error transaction.	Undefined	R
0	1:0	Always zero	0	R0

19.3.0.3 PCI Diagnostic Register

Table 19-5 PCI Diagnostic Register

Field	Bits	Description	POR state	RW
Reserved	63:7	Reserved, read as 0.	0	R0
DIS_RETRY	6	Disable retry limit. When set to 1, UltraSPARC IIi does not abort PIO operations after 512 retries, but continues indefinitely.	0	RW
Reserved	5:4	Reserved.	0	R0
I_PIO_A_PAR	3	Invert PIO address parity 0 = Correct parity asserted 1 = Incorrect parity asserted for all PCI PIO address phases.	0	RW
I_PIO_D_PAR	2	Invert PIO data parity 0 = Correct parity asserted 1 = Incorrect parity asserted for all PCI PIO write data phases.	0	RW
I_DMA_D_PAR	1	Invert DMA data parity 0 = Correct parity asserted 1 = Incorrect parity asserted for all PCI DMA read data phases.	0	RW
LPBK_EN	0	Not supported. Read as 0	0	R0

19.3.0.4 PCI Target Address Space Register

The PCI Target Address Space Register selectively enables 512 MByte regions as target PCI addresses for UltraSPARC IIi.

Table 19-6 PCI Target Address Space Register

Field	Bits	Description	POR state	RW
Reserved	63:8	Reserved, read as 0.	0	R0
EF_enable	7	Respond to 0xE000.0000-0xFFFF.FFFF	0	RW
CD_enable	6	Respond to 0xC000.0000-0xDFFF.FFFF	0	RW
AB_enable	5	Respond to 0xA000.0000-0xBFFF.FFFF	0	RW
89_enable	4	Respond to 0x8000.0000-0x9FFF.FFFF	0	RW
67_enable	3	Respond to 0x6000.0000-0x7FFF.FFFF	0	RW
45_enable	2	Respond to 0x4000.0000-0x5FFF.FFFF	0	RW
23_enable	1	Respond to 0x2000.0000-0x3FFF.FFFF	0	RW
01_enable	0	Respond to 0x0000.0000-0x1FFF.FFFF	0	RW

UltraSPARC IIi examines single-cycle PCI addresses and responds as a target if address[31:28] select an enabled region. Dual-cycle addresses are not selectively enabled as a target for UltraSPARC IIi. Only address[63:50]==0x3FFF indicates that UltraSPARC IIi is the target.

Note that more than one region can be enabled, and holes are allowed. No other PCI device should be enabled to respond to the UltraSPARC IIi target address space.

19.3.0.5 PCI DMA Write Synchronization Register

Normally, interrupt delivery to the UltraSPARC IIi core activates a Drain/Empty protocol to APB, to guarantee that any DMA writes received by APB prior to the interrupt arrival complete to memory. If another bus bridge exists behind APB, this procedure is insufficient. Software must execute a PIO load to the far side of that bus bridge,, to flush any of its posted DMA writes to APB, and then do a read of this register to synchronize with the posted writes in APB.

Table 19-7 PCI DMA Write Synchronization Register

Field	Bits	Description	RW
Reserved	63:0	Reserved, read as 0.	R0

Completion of the load instruction (with load-use dependency or MEMBAR) signifies that synchronization is complete.

19.3.0.6 PIO Data Buffer Diagnostic Access

The PIO R/W Data Buffer Diagnostics Access provides direct PIO accesses to 8 entries of PIO data RAM.

Table 19-8 PIO Data Buffer Diagnostics Access

Field	Bits	Description	Type
Data	63:0	PIO read/write buffer data	RW

Note – Generally, usage must be a Write then a Read of a single entry. The Write uses a PIO Data Buffer entry, so it is not possible to write all entries then read all entries.

19.3.0.7 DMA Data Buffer Diagnostic Access

The DMA Data Buffer Diagnostics Access provides direct PIO accesses to 8 entries of DMA data RAM.

Table 19-9 DMA Data Buffer Diagnostics Access

Field	Bits	Description	Type
Data	63:0	DMA read/write buffer data	RW

The (72:64) register is loaded as a side-effect of every read of one of the previous eight addresses. The data loaded is bits [72:64] of the relevant data buffer. On writes to the previous eight addresses, the contents of this register is used to write bits [72:64] of the relevant data buffer.

19.3.0.8 DMA Data Buffer Diagnostics Access

Table 19-10 DMA Data Buffer Diagnostics Access (72:64)

Field	Bits	Description	Type
Data	63:8	Reserved. Undefined data when read.	R
Data	7:0	DMA read/write buffer data	RW

19.3.1 PCI Configuration Space

The PBM contains a configuration header whose format is specified by the PCI Specification. The registers in the configuration header are accessed through PCI Configuration Address Space. The PBM is considered to be device 0 and function 0 on bus 0.

Table 19-11 PBM PCI Configuration Space

Register	PA
PBM Configuration Space. (Bus 0, Device 0, Function 0)	0x1FE.0100.0000 - 0x1FE.0100.00FF

Note – The PCI Configuration Address Space is little-endian. When accessing configuration space registers, software should take advantage of one of the SPARC V9 little-endian support mechanisms to get proper byte ordering. These mechanisms include little-endian ASIs or MMU support for marking pages little-endian. A load or store instruction of the same size as the register, for example, a byte or a halfword, should always be used.

The configuration header registers are defined by the PCI specification and PCI System Design Guide and are listed in *Table 19-12*. Some of the registers are not implemented in UltraSPARC IIi – indicated by shading in the table. The rule used is that any optional register for which equivalent information exists elsewhere is not implemented.

Table 19-12 Configuration Space Header Summary

Register	PA[40:0]	Size
Required PCI Device Configuration Header:		
Vendor ID	0x1FE.0100.0000	2 bytes
Device ID	0x1FE.0100.0002	2 bytes
Command	0x1FE.0100.0004	2 bytes
Status	0x1FE.0100.0006	2 bytes
Revision ID	0x1FE.0100.0008	1 byte
Programming I/F Code	0x1FE.0100.0009	1 byte
Sub-class Code	0x1FE.0100.000A	1 byte
Base Class Code	0x1FE.0100.000B	1 byte
Cache Line Size	0x1FE.0100.000C	1 byte
Latency Timer	0x1FE.0100.000D	1 byte

Table 19-12 Configuration Space Header Summary *(Continued)*

Register	PA[40:0]	Size
Header Type	0x1FE.0100.000E	1 byte
BIST	0x1FE.0100.000F	1 byte
Base Address	0x1FE.0100.0010- 0x1FE.0100.0027	Varies
Reserved	0x1FE.0100.0028- 0x1FE.0100.002F	n/a
Expansion ROM	0x1FE.0100.0030	4 bytes
Reserved	0x1FE.0100.0034- 0x1FE.0100.003B	n/a
Interrupt Line	0x1FE.0100.003C	1 byte
Interrupt Pin	0x1FE.0100.003D	1 byte
MIN_GNT	0x1FE.0100.003E	1 byte
MAX_LAT	0x1FE.0100.003F	1 byte
Optional Bridge Configuration Header:		
Bus Number	0x1FE.0100.0040	1 byte
Subordinate Bus Number	0x1FE.0100.0041	1 byte
Reserved	0x1FE.0100.0042- 0x1FE.0100.00FF	n/a
Disconnect Counter	Unspecified	1 byte
Bridge Command/Status	Unspecified	4 bytes
Bridge Memory Base Address	Unspecified	4 bytes
Bridge Memory Limit Address	Unspecified	4 bytes
DOS Read Attributes	Unspecified	2 bytes
DOS Write Attributes	Unspecified	2 bytes
Bridge I/O Base Address	Unspecified	2 bytes
Bridge I/O Limit Address	Unspecified	2 bytes

Note – *Table 19-12* lists the logical size for each register but PIO access to the registers can be in any size from 1 to 8 bytes.

19.3.1.1 PCI Configuration Space Vendor ID

Read only; VendorID<15:0> = 0x108E

19.3.1.2 PCI Configuration Space Device ID

Read only; DeviceID<15:0> = 0xA000

Compatibility Note – This device ID is different from that of prior PCI-based UltraSPARC systems.

19.3.1.3 PCI Configuration Space Command Register

Table 19-13 Command Register

Field	Bits	Description	POR state	RW
Reserved	15:10	Reserved, read as 0.	0	R0
FAST_EN	9	Enable fast back-to-back cycles to different targets. Hardwired to 0 (disabled).	0	R0
SERR_EN	8	Enable driving of SERR# pin.	0	RW
WAIT	7	Enable use of address/data stepping Hardwired to 0 (disabled).	0	R0
PER	6	Enable reporting of parity errors	0	RW
VGA	5	Enable VGA palette snooping Hardwired to 0 (disabled).	0	R0
MWI	4	Enables use of Memory Write & Invalidate Hardwired to 0 (disabled).	0	R0
SPCL	3	Enables monitoring of special cycles Hardwired to 0 (disabled).	0	R0
MSTR	2	Enables ability to be bus master Hardwired to 1 (enabled).	1	R1
MEM	1	Enables response to PCI MEM cycles Hardwired to 1 (enabled).	1	R1
IO	0	Enables response to PCI I/O cycles. Hardwired to 0 (disabled).	0	R0

19.3.1.4 PCI Configuration Space Status Register

Table 19-14 Status Register

Field	Bits	Description	POR state	RW
DPE	15	Set if PBM detects a parity error	0	R/W1C
SSE	14	Set if PBM signalled a system error. (detects address parity error).	0	R/W1C
RMA	13	Set if PBM receives a master-abort	0	R/W1C
RTA	12	Set if PBM receives a target-abort	0	R/W1C
STA	11	Set if PBM generates target-abort	0	R/W1C
DVSL	10:9	Timing of DEVSEL#. Hardwired to 01 (medium speed response)	1	R01
DPD	8	Set when parity error occurs while PBM is bus master, if PER in command register also set.	0	R/W1C
FASTCAP	7	Indicates ability to accept fast back-to-back cycles as target, when the back-to-back transactions are not to the same target. Hardwired to 1 (allowed)	1	R1
UDF_SUPPORT	6	User Definable Feature Support Hardwired to 0 (no user definable features)	0	R0
66MHZ_CAPABLE	5	Indicates ability to run at 66MHz clock speed. Hardwired to 1 (66MHz capable) for PBM.	1	R1
Reserved	4:0	Reserved, read as 0	0	R0

19.3.1.5 PCI Configuration Space Revision ID Register

Read only; RevisionID<7:0> = 0x00; this register always reads as 0

19.3.1.6 PCI Configuration Space Programming I/F Code Register

Read only; ProgrammingIFCode<7:0> = 0x00

19.3.1.7 PCI Configuration Space Sub-class Code Register

Read only; SubclassCode<7:0> = 0x00 (specifies host bridge device)

19.3.1.8 PCI Configuration Space Base Class Code Register

Read only; BaseClassCode<7:0> = 0x06 (specifies bridge device)

19.3.1.9 PCI Configuration Space Latency Timer Register

This 8-bit read/write register specifies the value of the latency timer for the PBM as a bus master. Only the top five bits are implemented, giving a timer granularity of 8 PCI clocks. The bottom three bits read as 0 and should be written as 0. The maximum PIO transfer is 64 bytes, so the latency timer may apply for transfers that insert many wait states to slow targets.

Compatibility Note – A value of 0 means there is no latency timeout.

Table 19-15 Latency Timer Register

Field	Bits	Description	POR state	RW
LAT_TMR_HI	7:3	Programmable portion of latency timer.	0	RW
LAT_TMR_LO	2:0	Read only portion of latency timer. Hardwired to 0.	0	R0

19.3.1.10 PCI Configuration Space Header Type Register

Table 19-16 Header Type Register

Field	Bits	Description	RW
MULTI_FUNC	7	Indicates whether the PBM is a multi-function PCI device. Hardwired to 0 (not multi-function).	R0
HDR_TYPE	6:0	Defines layout of configuration header bytes 0x10-0x3F. Hardwired to 0 (the only defined value in PCI specification)	R0

19.3.1.11 PCI Configuration Space Bus Number

This 8-bit read/write register specifies the number of the PCI bus on which this bridge is found. Although programmable, it is not used. UltraSPARC IIi always assumes it is on bus 0 when decoding a PIO PA to determine whether to create Type 0 or Type 1 configuration cycles.

Table 19-17 Bus Number Register

Field	Bits	Description	POR state	RW
BUS	7:0	Bus number	0	RW

19.3.1.12 PCI Configuration Space Subordinate Bus Number

This 8-bit read/write register specifies the highest subordinate bus number beneath this bridge. Although programmable, it has no effect on UltraSPARC IIi.

Table 19-18 Subordinate Bus Number Register

Field	Bits	Description	POR state	RW
SUB_BUS	7:0	Highest subordinate bus number	0	RW

19.3.1.13 PCI Configuration Space Unimplemented Registers

The following registers are defined in the PCI Specification or PCI System Design Guide, but are not implemented in UltraSPARC IIi's PBM for the indicated reasons.

Cache Line Size The cache line size is fixed at 64-bytes.

BIST Built-In-Self-Test is not implemented in UltraSPARC IIi.

Base Address Registers The bridge has neither memory nor I/O space. Its configuration space is accessible only from the host and is hard-mapped.

Interrupt Line, Interrupt Pin Do not apply; interrupt lines are handled by the RIC ASIC.

Min_Gnt, Max_Lat There is no regular traffic pattern to programmed I/O. Values of zero (true) indicate there are no stringent requirements.

19.3.2 IOMMU Registers

Table 19-19 IOMMU Registers

Register	Offset	Access Size
IOMMU Control Register	0x1FE.0000.0200	8 bytes
IOMMU TSB Base Address Reg.	0x1FE.0000.0208	8 bytes
IOMMU Flush Register	0x1FE.0000.0210	8 bytes
IOMMU Virtual Addr. Diag. Reg.	0x1FE.0000.A400	8 bytes
IOMMU Tag Compare Diag.	0x1FE.0000.A408	8 bytes
IOMMU LRU Queue Diag.	0x1FE.0000.A500 - 0x1FE.0000.A57F	8 bytes
IOMMU Tag Diag.	0x1FE.0000.A580 - 0x1FE.0000.A5FF	8 bytes
IOMMU Data RAM Diag.	0x1FE.0000.A600 - 0x1FE.0000.A67F	8 bytes

19.3.2.1 IOMMU Control Register

The Control Register affects diagnostic mode, IOMMU TSB size and page size.

Table 19-20 IOMMU Control Register

Field	Bits	Description	POR state	Type
RESERVED	63:24	Reserved, read as zeros	0	R0
ERRSTS	26:25	If ERR is set, indicates the type of error logged in the IOMMU state.	0	R/ W1C
ERR	24	Set when IOMMU is written with an ERR	0	R/ W1C
LRU_LCKEN	23	LRU Lock Enable Bit. When set, only the IOMMU entry specified by the Lock Pointer can be replaced.	0	RW
LRU_LCKPTR	22:19	LRU Lock Pointer. Works in conjunction with the LRU Lock Enable bit to limit IOMMU replacement to a single entry.	RW	RW
TSB_SIZE	18:16	IOMMU TSB table size. Number of 8 byte entries: 0=1K, 1=2K, 2=4K, 3=8K, 4=16K, 5=32K, 6=64K, 7=128K.	0	RW
RESERVED	15:3	Reserved, read as zeros	0	R0

Table 19-20 IOMMU Control Register *(Continued)*

Field	Bits	Description	POR state	Type
TBW_SIZE[1]	2	Assumed page size during IOMMU TSB lookup. 0 = 8K page 1 = 64K page	0	RW
MMU_DE	1	Diagnostic mode enable, when set it enables the diagnostic mode. See description of IOMMU tag diagnostics.	0	RW
MMU_EN	0	IOMMU enable bit, when set it enables the translation.	0	RW

1. If DMA mappings are always 8K pages, or mixed 8K and 64K pages, set this bit to '0' so that the index is constructed for 8K lookup. If all DMA mappings are to 64K pages, set this bit to '1' so that the index is based on 64K pages. When this bit is '0', a 64K mapping should be placed in all eight TSB entries in which it is indexed.

Compatibility Note – ERR and ERRSTS are not present in prior PCI-based UltraSPARC systems.

Table 19-21 Address Space Size And Base Address Determination.

	TBW_SIZE == 0		TBW_SIZE == 1	
TSB_SIZE	VA Space Size	TSB Index	VA Space Size	TSB_Index
0	8 MB	VA<22:13>,000	64 MB	VA<25:16>,000
1	16 MB	VA<23:13>.000	128 MB	VA<26:16>,000
2	32 MB	VA<24:13>,000	256 MB	VA<27:16>,000
3	64 MB	VA<25:13>,000	512 MB	VA<28:16>,000
4	128 MB	VA<26:13>,000	1 GB	VA<29:16>,000
5	256 MB	VA<27:13>,000	2 GB	VA<30:16>,000
6	512 MB	VA<28:13>,000	not allowed[1]	--
7	1GB	VA<29:13>,000	not allowed[1]	--

1. Hardware does not prevent illegal combinations from being programmed. If an illegal combination is programmed into the IOMMU, all translation requests will be rejected as invalid.

Address space size and TSB offset are affected by TSB_SIZE and TBW_SIZE as shown in *Table 19-21*.

IOMMU locking

For diagnostics and debugging, the IOMMU has the capability of restricting itself to use just a single entry of the IOMMU. This is controlled by the LRU_LCKEN and LRU_LCKPTR fields of the IOMMU Control Register. To properly turn locking on the following sequence is required:

- Set MMU_EN to 0
- Set LRU_LCKEN to 1 (must be a separate PIO write)
- Set LRU_LCKPTR to desired value (may be combined with previous PIO)
- Set MME_DE to 1 (may be combined with previous PIO)
- Invalidate all IOMMU entries
- Set MMU_EN to 1 and MMU_DE to 0.

To unlock the IOMMU:

- Set LRU_LCKEN to 0

19.3.2.2 IOMMU TSB Base Address Register

The IOMMU TSB Base Address Register contains the pointer to the first-entry of the IOMMU TSB table. Together with part of the virtual address it uniquely identifies the address from which hardware should fetch the TTE from the IOMMU TSB table. The IOMMU TSB table has to be aligned on an 8K boundary. The lower order 13 bits are assumed to be 0x0 during IOMMU TSB table lookup. Tables larger than 8K bytes are only constrained to be on 8K boundaries rather than having to be size aligned.

Table 19-22 IOMMU TSB Base Address Register

Field	Bits	Description	Type
RESERVED	63:41	Reserved, read as zeros	R0
ZERO	40:13	Bits 40:34 of the TSB physical address are always zero	R0
TSB_BASE	33:13	Bits [33:13] of the TSB physical address. 33:30 should always be zero, since only 1-Gbyte of physical memory is supported.	RW
RESERVED	12:0	Reserved, read as zeros	R0

19.3.2.3 Flush Address Register

This is a write-only pseudo-register to allow software perform address-based flush of a mapping from IOMMU. The data written to this address contains the page number to be flushed. A IOMMU entry with matched page number is invalidated.

Table 19-23 Flush Address Register

Field	Bits	Description	Type
RESERVED	63:32	Reserved, write has no effect	W
FLUSH_VPN	31:13	31:16 = virtual page number if 64K page; bits 15:13 are *don't care* 31:13 = virtual page number if 8K page	W
RESERVED	12:0	Reserved, write has no effect	W

Note – No hardware mechanisms exist to solve the potential race between a DMA translation needing a IOMMU entry and the write to the Flush Address Register intended to flush that entry. Software must manage the interlock by guaranteeing that no DMA transfers can involve the page being flushed.

19.3.2.4 IOMMU TAG Diagnostics Access

The IOMMU Tag Diagnostics Access provides a diagnostics path to the 16-entry IOMMU Tag when the MMU_DE bit in the IOMMU Control Register is turned on.

Table 19-24 IOMMU Tag Diagnostics Access

Field	Bits	Description	Type
RESERVED	63:25	Reserved, read as zeros	R0
ERRSTS	24:23	Error Status: 00 = Reserved 01 = Invalid Error 10 = Reserved 11 = UE Error on TTE read	RW
ERR	22	When set to 1, indicates that there is an error associated with this IOMMU entry. The specific error is indicated by the ERRSTS field.	RW
W	21	Writable bit. when set, the page mapped by the IOMMU has write permission granted.	RW
S	20	Stream bit. (unused)	RW
SIZE	19	Page Size, 0=8K and 1=64K.	RW
VPN	18:0	VPN[31:13]	RW

Note – Diagnostic accesses should ensure that multiple match conditions are not generated. The result of multiple matches is unpredictable.

Compatibility Note – Unlike prior PCI-based UltraSPARC systems, UltraSPARC IIi arbitrates between IOMMU CSR access and DMA access. This property may allow software more flexibility.

19.3.2.5 IOMMU Data RAM Diagnostic Access

The IOMMU Data Diagnostics Access provides direct PIO accesses to 16 entries of IOMMU Data RAM. The MMU_DE bit in the IOMMU Control Register must be turned on to perform the accesses. *Table 19-25* shows the information included in the returned data.

Table 19-25 IOMMU Data RAM Diagnostics Access

Field	Bits	Description	Type
RESERVED	63:31	Reserved, read as zeros	R0
V	30	Valid bit, when set, the TLB data field is meaningful	RW
U	29	Used bit. Affects the LRU replacement.	RW
C	28	Cacheable bit. 1=Cacheable access, 0=Non-cacheable.	RW
PA[40:34]	27:21	Not stored. All 1's if Noncacheable, All 0's if Cacheable.	R
PA[33:13]	20:0	21-bit Physical Page Number	RW

Compatibility Note – The Used bit does not exist in prior PCI-based UltraSPARC systems, and is used by the pseudo-LRU replacement algorithm.

19.3.2.6 Virtual Address Diagnostic Register

This register is used to set up the virtual address for the IOMMU compare diagnostic. The virtual address is written to this register and enables the compare results to be read from the IOMMU.

Table 19-26 Virtual Address Diagnostic Register

Field	Bits	Description	Type
RESERVED	63:32	Reserved, read as 0.	R0
VPN	31:13	Virtual page number.	R/W
RESERVED	12:00	Reserved, read as 0.	R0

19.3.2.7 IOMMU Tag Compare Diagnostic Access

Table 19-27 IOMMU Tag Comparator Diagnostics Access

Field	Bits	Description	Type
RESERVED	63:16	Reserved, read as zeros	R0
COMP	15:0	IOMMU tag comparator output for each entry.	R

Note – The IOMMU Tag Compare Diagnostics Access provides the diagnostics path to the 16-entry IOMMU Tag Comparator when the MMU_DE bit in the IOMMU Control Register is turned on. Bit 0 represents the comparison result of the first IOMMU Tag entry, and bit 15 represents the last.

19.3.3 Interrupt Registers

Interrupts load the Interrupt Vector Data registers with the data shown in *Figure 19-1*. See Section 11.10.4, *Incoming Interrupt Vector Data*<2:0> on page 120.

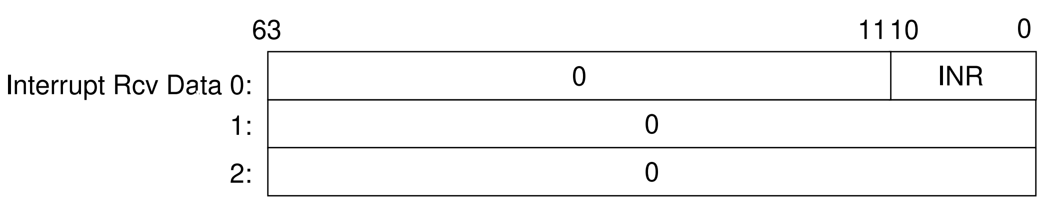


Figure 19-1 Interrupt Vector Data Registers Contents

INR is an 11 bit interrupt number that indicates the source of the interrupt. Where possible, the interrupt is precise (that is, it points to only one interrupt source). This singularity permits the dispatch of the proper interrupt service routine without any register polling.

Bits [11] through [63]of the first word are guaranteed to be 0 for all UltraSPARC IIi IO generated interrupts. Words 1 and 2 of the interrupt packet are also guaranteed to be 0.

Each interrupt source has a mapping register, containing the INR value used for the interrupt. The INR has two parts: IGN and INO. The Interrupt Group Number (IGN) is the upper 5 bits of the INR, and for most interrupts is 0x1f.

Compatibility Note – The IGN on UltraSPARC IIi is not programmable for the Partial Interrupt Mapping Registers, and is fixed to 0x1f.

The lower 6 bits of the INR are the Interrupt Number Offset (INO). This value is hardcoded by UltraSPARC IIi for each interrupt source, as shown in *Table 19-28*, and is read-only in the mapping register. For PCI slot interrupt mapping registers, INO<1:0> is always read as 00.

For Graphics (FFB) and UPA64S expansion interrupts, the full 11-bit INR field is writable, and under software control.

Table 19-28 Interrupt Number Offset Assignments

INO (binary)	INO (hex)	Interrupt Source
0bssnn	00-1F	PCI Bus b Slot ss Interrupt nn b = 0 for bus A, 1 for bus B ss = 00-11 for bus A or B slots, nn = 00-11 for INTA#,INTB#,INTC#,INTD#
100000	20	SCSI
100001	21	Ethernet
100010	22	Parallel port
100011	23	Audio Record
100100	24	Audio Playback
100101	25	Power Fail
100110	26	Keyboard/mouse/serial
100111	27	Floppy
101000	28	Reserved (spare HW int)
101001	29	Keyboard
101010	2A	Mouse

Table 19-28 Interrupt Number Offset Assignments *(Continued)*

INO (binary)	INO (hex)	Interrupt Source
101011	2B	Serial
101100	2C	Reserved
101101	2D	Reserved
101110	2E	DMA UE
101111	2F	DMA CE
110000	30	PCI Bus Error
110001	31	Reserved
110010	32	Reserved
111111	3F	Reserved

Each interrupt source has an associated state register that can be either of type "level" or of type "pulse."

In the level sensitive case, the state register has two bits and there are three valid states: IDLE, RECEIVED, and PENDING.

- IDLE: No interrupt in progress.
- RECEIVED: An Interrupt has been detected and will be delivered to the processor if the valid bit is set in the mapping register.
- PENDING: Interrupt has been delivered to the UltraSPARC IIi core. Any subsequent detection of the same interrupt is ignored until software resets the state machine back to IDLE.

Software can set the state register for each level sensitive interrupt to any of these states using the Clear Interrupt Registers.

In the pulse case, the state register consists of a single bit, with two states: IDLE and RECEIVED. These states have the same meaning as those for the level sensitive case. There is no PENDING state, so the state machine transitions from RECEIVED back to IDLE when the interrupt is dispatched to a processor.

Diagnostic access is provided to allow software to read the state register for all interrupt sources.

Compatibility Note – There is no RECEIVED state for DMA CE, DMA UE, or PCI Error Interrupts. They cause their interrupt FSMs to go from the IDLE to the PENDING state directly, when present and enabled.

19.3.3.1 Partial Interrupt Mapping Registers

The offset of each partial Interrupt Mapping Register can be derived from the associated INO. There are two cases:

```
PCI Interrupts: IMR address = 0x1FE.0000.0C00 + (INO & 0x3C) << 1

OBIO Interrupts:IMR address = 0x1FE.0000.1000 + (INO & 0x1F) << 3
```

Table 19-29 Partial Interrupt Mapping Registers

Register	PA	Access Size
PCI Bus A Slot 0 Int Mapping Reg	0x1FE.0000.0C00	8 bytes
PCI Bus A Slot 1 Int Mapping Reg	0x1FE.0000.0C08	8 bytes
PCI Bus A Slot 2 Int Mapping Reg	0x1FE.0000.0C10	8 bytes
PCI Bus A Slot 1 Int Mapping Reg	0x1FE.0000.0C18	8 bytes
PCI Bus B Slot 0 Int Mapping Reg	0x1FE.0000.0C20	8 bytes
PCI Bus B Slot 1 Int Mapping Reg	0x1FE.0000.0C28	8 bytes
PCI Bus B Slot 2 Int Mapping Reg	0x1FE.0000.0C30	8 bytes
PCI Bus B Slot 3 Int Mapping Reg	0x1FE.0000.0C38	8 bytes
SCSI Int Mapping Reg	0x1FE.0000.1000	8 bytes
Ethernet Int Mapping Reg	0x1FE.0000.1008	8 bytes
Parallel Port Int Mapping Reg	0x1FE.0000.1010	8 bytes
Audio Record Int Mapping Reg	0x1FE.0000.1018	8 bytes
Audio Playback Int Mapping Reg	0x1FE.0000.1020	8 bytes
Power Fail Int Mapping Reg	0x1FE.0000.1028	8 bytes
Kbd/mouse/serial Int Mapping Reg	0x1FE.0000.1030	8 bytes
Floppy Int Mapping Reg	0x1FE.0000.1038	8 bytes
Spare HW Int Mapping Reg	0x1FE.0000.1040	8 bytes
Keyboard Int Mapping Reg	0x1FE.0000.1048	8 bytes
Mouse Int Mapping Reg	0x1FE.0000.1050	8 bytes
Serial Int Mapping Reg	0x1FE.0000.1058	8 bytes
Reserved	0x1FE.0000.1060	8 bytes
Reserved	0x1FE.0000.1068	8 bytes
DMA UE Int Mapping Reg	0x1FE.0000.1070	8 bytes
DMA CE Int Mapping Reg	0x1FE.0000.1078	8 bytes
PCI Error Int Mapping Reg	0x1FE.0000.1080	8 bytes

The format for each partial interrupt mapping register is shown in *Table 19-30*

Table 19-30 Format of Partial Interrupt Mapping Registers

Field	Bits	Description	POR state	Type
Reserved	63:32	Reserved, read as 0	0	R0
V	31	Valid bit When set to 0, interrupt will not be dispatched to CPU. Has no other impact on interrupt state.	0	R/W
Reserved	30:11	Reserved, read as 0	0	R0
IGN	10:6	Read as 0x1f	0x1F	R
INO	5:0	Interrupt Number Offset The value of this field is hardwired for each mapping register, as shown in *Table 19-28*	-	R

Note that these registers have only 1 RW bit defined per address.

19.3.3.2 Full Interrupt Mapping Registers

There are only two full Interrupt Mapping Registers in UltraSPARC IIi. See *Table 19-31*.

Table 19-31 Full Interrupt Mapping Registers

Register	PA	Access Size
On board graphics Int Mapping Reg	0x1FE.0000.1098 and 0x1FE.0000.6000[1]	8 bytes
Expansion UPA64S Int Mapping Reg	0x1FE.0000.10A0 and 0x1FE.0000.8000	8 bytes

1. Accesses to either of these addresses behave identically; in other words, the registers are double mapped.

The format for the full Interrupt Mapping Registers, shown in *Table 19-32*, is the same as that of the partial Interrupt Mapping Registers, except for the INR field.

Table 19-32 Format of Full Interrupt Mapping Registers

Field	Bits	Description	POR state	Type
Reservd	63:32	Reserved, read as 0	0	R0
V	31	Valid bit When set to 0, interrupt will not be dispatched to CPU. Has no other impact on interrupt state.	0	R/ W
Reservd	30:11	Reserved, read as 0	0	R0
INR	10:0	Interrupt Number	-	R/ W

19.3.3.3 Clear Interrupt Registers

The address of each Clear Interrupt Register can be derived from the associated INO. There are two cases:

PCI Interrupts: CIR address = 0x1FE.0000.1400 + (INO & 0x1F) << 3

OBIO Interrupts: CIR address = 0x1FE.0000.1800 + (INO & 0x1F) << 3

The graphics and UPA expansion interrupts do not have associated Clear Interrupt Registers because they are pulse type interrupts that are automatically cleared when sent.

Table 19-33 Clear Interrupt Pseudo Registers

Register	PA	Access Size
PCI Bus A Slot 0 Clear Int Regs	0x1FE.0000.1400 - 0x1FE.0000.1418	8 bytes
PCI Bus A Slot 1 Clear Int Regs	0x1FE.0000.1420 - 0x1FE.0000.1438	8 bytes
PCI Bus A Slot 2 Clear Int Regs	0x1FE.0000.1440 - 0x1FE.0000.1458	8 bytes
PCI Bus A Slot 3 Clear Int Regs	0x1FE.0000.1460 - 0x1FE.0000.1478	8 bytes
PCI Bus B Slot 0 Clear Int Regs	0x1FE.0000.1480 - 0x1FE.0000.1498	8 bytes
PCI Bus B Slot 1 Clear Int Regs	0x1FE.0000.14A0 - 0x1FE.0000.14B8	8 bytes
PCI Bus B Slot 2 Clear Int Regs	0x1FE.0000.14C0 - 0x1FE.0000.14D8	8 bytes

Table 19-33 Clear Interrupt Pseudo Registers *(Continued)*

Register	PA	Access Size
PCI Bus B Slot 3 Clear Int Regs	0x1FE.0000.14E0 - 0x1FE.0000.14F8	8 bytes
SCSI Clear Int Reg	0x1FE.0000.1800	8 bytes
Ethernet Clear Int Reg	0x1FE.0000.1808	8 bytes
Parallel Port Clear Int Reg	0x1FE.0000.1810	8 bytes
Audio Record Clear Int Reg	0x1FE.0000.1818	8 bytes
Audio Playback Clear Int Reg	0x1FE.0000.1820	8 bytes
Power Fail Clear Int Reg	0x1FE.0000.1828	8 bytes
Kbd/mouse/serial Clear Int Reg	0x1FE.0000.1830	8 bytes
Floppy Clear Int Reg	0x1FE.0000.1838	8 bytes
Spare HW Clear Int Reg	0x1FE.0000.1840	8 bytes
Keyboard Clear Int Reg	0x1FE.0000.1848	8 bytes
Mouse Clear Int Reg	0x1FE.0000.1850	8 bytes
Serial Clear Int Reg	0x1FE.0000.1858	8 bytes
Reserved	0x1FE.0000.1860	8 bytes
Reserved	0x1FE.0000.1868	8 bytes
DMA UE Clear Int Reg	0x1FE.0000.1870	8 bytes
DMA CE Clear Int Reg	0x1FE.0000.1878	8 bytes
PCI Async Error Clear Int Reg	0x1FE.0000.1880	8 bytes

One such register exists per interrupt source. The lower 2 bits of the data word written to this register specify the operation as shown in *Table 19-34*. All other bits should be written as 0 to guarantee future compatibility.

Table 19-34 Clear Interrupt Register

Field	Bits	Description	Type
RESERVED	63:02	Reserved.	W
STATE	01:00	State bits for the interrupt state machine associated with this interrupt. The following values may be written: 00 - Set state machine to IDLE state 01 - Set state machine to RECEIVED state 10 - Reserved 11 - Set state machine to PENDING state	W

Note – The Interrupt Clear Registers are write only. To determine the current interrupt state, use the interrupt state diagnostic registers instead.

19.3.3.4 Interrupt State Diagnostic Registers

Table 19-35 Interrupt State Diagnostic Registers

Register	PA	Access Size	POR state	Type
PCI Int State Diag Reg	0x1FE.0000.A800	8 bytes	0	R
OBIO and Misc Int State Diag Reg	0x1FE.0000.A808	8 bytes	0	R

The Interrupt State Diagnostic Register bit assignments are shown in *Table 19-36* and in *Table 19-37*.

The locations of each set of state bits can also be derived from the associated INO (except for Graphics and UPA expansion interrupts, for which the INO is fully programmable):

Code Example 19-1 State Bit Locations from INO

```
Register: if (INO & 0x20) then OBIO Int Diag Reg else PCI Int Diag Reg

Bits: Int Diag Reg [ ((INO & 0x1F)<<1)+1 : ((INO & 0x1F)<<1) ]
```

The Graphics and UPA64S expansion interrupts are pulse type interrupts; all others are level type interrupts.

Table 19-36 Level Interrupt State Assignment

Field	Description
INT_STATE<1:0>	00 - IDLE state; no interrupt received or pending. 01 - RECEIVED state; interrupt detected, but not dispatched. 11 - PENDING state; interrupt is received and dispatched. 10 - Illegal state.

Table 19-37 Pulse Interrupt State Assignment

Field	Description
INT_STATE<0>	0 - IDLE state; no interrupt received 1 - RECEIVED state; interrupt detected, but not dispatched.

Definitions of the registers are shown in a general way in the table below. Refer to *Code Example 19-1* above for specific bit positions. As an example, the bit position for PCI Bus B Slot 1, INTB# is <43:42>.

Table 19-38 PCI Interrupt State Diagnostic Register Definition

Bits	Description
7:0	PCI Bus A Slot 0 INT# DCBA
15:8	PCI Bus A Slot 1 INT# DCBA
23:16	PCI Bus A Slot 2 INT# DCBA
31:24	PCI Bus A Slot 3 INT# DCBA
39:32	PCI Bus B Slot 0 INT# DCBA
47:40	PCI Bus B Slot 1 INT# DCBA
55:48	PCI Bus B Slot 2 INT# DCBA
63:56	PCI Bus B Slot 3 INT# DCBA

Table 19-39 OBIO and Misc Int Diag Reg Definition

Bits	Description
1:0	SCSI Int State
3:2	Ethernet Int State
5:4	Parallel Port Int State
7:6	Audio Record Int State
9:8	Audio Playback Int State
11:10	Power Fail Int State
13:12	Kbd/mouse/serial Int State
15:14	Floppy Int State
17:16	Spare HW Int State
19:18	Keyboard Int State
21:20	Mouse Int State
23:22	Serial Int State
29:28	DMA UE Int State
31:30	DMA CE Int State
33:32	PCI Error Int State
35:34	Reserved (return 0 on read)
37:36	Reserved (return 0 on read)

Table 19-39 OBIO and Misc Int Diag Reg Definition *(Continued)*

Bits	Description
34	Graphics Int State
35	Expansion UPA64S Int State
63:36	Reserved (return 0 on read)

Compatibility Note – Note the "Graphics Int State" and Expansion UPA64S Int State" bits are moved from bits 38 and 39 (position in prior UltraSPARC systems) to bits 34 and 35 respectively.

19.3.4 PCI INT_ACK Generation

UltraSPARC IIi can generate an interrupt acknowledge in response to a PCI Interrupt.

Name: ASI_INT_ACK (Privileged)

ASI: 0x7F, VA<63:32>==0x1FF VA<31:0>== (any address to PCI)

Table 19-40 PCI INT_ACK Register Format

Bits	Field	Use	RW
<7:0>	DATA<7:0>	INT_ACK data from PCI	R

BUSY: This bit is set when an interrupt vector is received.

DATA<7:0>: Data returned on PCI byte 0 during INT_ACK cycle.

Non-privileged access to this register causes a *privileged_action* trap.

The address generated on the PCI bus is equal to VA[31:0])

VA[23:21] should be set to specific values when the APB MAP_INTACK_A/B functions are enabled, to control the forwarding of the INT_ACK to the A or B bus. The particular VA[23:21] depends on the way IO space is divided, since the same mapping register is used in APB for IO space, and MAP_INTACK_A/B forwarding. VA[23:21] are don't care if the APB ROUTE_INTACK_A/B functions are used to hardwire the INT_ACK forwarding. All other VA[31:24],[20:0] can be random values; zeros are recommended.

If software does anything other than a byte/halfword/word load with ASI_INT_ACK, UltraSPARC IIi/APB operation is undefined. A byte load should be correct for most systems.

All error logging and events for PCI loads apply equally to this INT_ACK cycle generated by UltraSPARC IIi.

19.4 PCI Address Space

PCI devices can be connected directly to the UltraSPARC IIi PCI bus.

UltraSPARC IIi can also be used with an external PCI bridge, the Advanced PCI Bridge (APB), that can connect to separate PCI A and PCI B PCI buses. UltraSPARC IIi support of multiple PCI buses includes interrupt management and flexible address mapping.

APB provides a generalized address decode facility and a flexible target address space definition for DMA. Both PCI A and B can each support four PCI devices. There are no separate UltraSPARC IIi CSRs for the A and B buses created by APB but only the single set of CSRs for the PCI bus connected to UltraSPARC IIi

19.4.1 PCI Address Space—PIO

Several regions of UltraSPARC IIi's physical address space are used to access devices on the PCI bus that it supports.

For the non-block transfers, any legal combination of bits in the bytemask may be set (that is, arbitrary bytemasks for writes, aligned 1, 2, 4, 8 or 16 byte bytemasks for reads), within the size restrictions listed below. The PCI byte enables generated by UltraSPARC IIi are identical to those generated by the UltraSPARC core.

The PCI specification, version 2.1 requires AD[1:0] to point to the first byte enable for I/O writes. This requirement is not met by UltraSPARC IIi during:

- compression of byte or halfword stores (Ebit==0) or
- use of the PSTORE instruction to generate random byte enables.

Generally, software should use only normal, non-compressed loads and stores to I/O space, and UltraSPARC IIi meets the AD[1:0] requirement for those instructions.

Also note that UltraSPARC IIi can generate multiple data beat Configuration Read or Writes.

Table 19-41 Physical Address Space to PCI Space Mappings

PCI Address Space	PA[40:0]	CPU Commands Supported	PCI Commands Generated
PCI Configuration Space	0x1FE.0100.0000- 0x1FE.01FF.FFFF	NC read (any) NC write (any)	Configuration Read Configuration Write (may also be Special Cycle)
PCI Bus I/O Space	0x1FE.0200.0000- 0x1FE.02FF.FFFF	NC read (any) NC write (any)	I/O Read I/O Write
Do not use	0x1FE.0300.0000- 0x1FE.FFFF.FFFF		May wrap to Configuration or I/O Space behavior
PCI Bus Memory Space	0x1FF.0000.0000- 0x1FF.FFFF.FFFF	NC read (4 byte) NC read (8 byte) NC Block read NC write NC Block write NC Instruction fetch	Memory Read Memory Read Multiple Memory Read Line Memory Write Memory Write Memory Read

Note – All PCI address spaces use little-endian address byte ordering. Any accesses made to a PCI address space should use one of the SPARC V9 little-endian support mechanisms to get proper byte ordering. These mechanisms include little-endian ASIs or MMU support for marking pages little-endian

19.4.1.1 PCI Configuration Space

PCI configuration cycles can be generated by UltraSPARC IIi in response to PIO reads and writes to addresses in the PCI Configuration Space. UltraSPARC IIi generates both Type 0 and Type 1 configuration cycles. Type 0 configuration cycles are used to configure devices on the UltraSPARC IIi primary PCI bus, including APB. Type 1 configuration cycles are used to configure devices on secondary PCI busses via APB.

UltraSPARC IIi does not implement either of the two means of generating PCI configuration cycles defined by the PCI Specification but instead uses the following means.

An UltraSPARC IIi PIO causes a type 0 configuration cycle on the primary PCI bus if PA[32:24] equals 0x001 and PA[23:16] (Bus Number) equals 0, and the Device Number is not 0. A Device Number of 0 designates the PBM itself, and the configuration cycle does not appear on the PCI bus.

Figure 19-2 shows how address bits 15:0 map to the PCI configuration cycle address.

32 24	23 16	15 11	10 8	7 2	1	0
0 0 0 0 0 0 0 0 1	Bus Number	Device Number	Function Number	Register Number	0	0

Configuration Space Address

31 11	10 8	7 2	1	0
$2^{\text{Device Number}}$ (Only one ‘1’)	Function Number	Register Number	0	1

PCI Configuration Cycle Address

Figure 19-2 Type 0 Configuration Address Mapping

The UltraSPARC IIi PCI bus has no IDSEL# pins so device IDSEL# lines must be resistively tied to individual AD[31:11] lines. It is recommended that slot 0 be device 1, tied to AD[12]; slot 1 be device 2; tied to AD[13], and so on.

Compatibility Note – The UltraSPARC IIi PCI bus is hardwired to Bus Number == 0

A type 1 configuration cycle is generated when the bus number field of the configuration space address is not zero (that is, the UltraSPARC IIi Bus Number).

The type 1 configuration cycle address is constructed from the configuration space address as shown in *Figure 19-3*.

32 24	23 16	15 11	10 8	7 2	1	0
0 0 0 0 0 0 0 0 1	Bus Number	Device Number	Function Number	Register Number	0	0

Configuration Space Address

31 24	23 16	15 11	10 8	7 2	1	0
Reserved	Bus Number	Device Number	Function Number	Register Number	0	0

PCI Configuration Cycle Address

Figure 19-3 Type 1 Configuration Address Mapping

Note – APB looks at type 0 and type 1 configuration cycle addresses, and either routes type 1 transactions to one of the secondary busses, or to its own configuration space. See the APB User's Manual for details.

Compatibility Note – UltraSPARC IIi aliases Functions 1-7 of its PCI Configuration space to its Function 0 PCI Configuration space. (Bus 0, Device 0). The PCI specification requires that zeros be returned and stores ignored. Since this address space is only accessible to UltraSPARC IIi PIO instructions, specifically boot PROM code, this aliasing should not be problematic because the boot PROM should never reference the UltraSPARC IIi Function 1-7 addresses (see *Type 0 Configuration Address Mapping* on page 312 for the address decode scheme).

19.4.1.2 PCI I/O Space

PCI I/O cycles are generated by UltraSPARC IIi in response to PIO reads and writes to addresses in one of the PCI I/O Spaces (one for each bus). For each access to I/O space, an I/O Read or I/O Write command is issued on the appropriate PCI bus. Bits 31:24 of the address on the PCI bus will be 0, and bits 23:0 will be a copy of physical address bits 23:0.

Note – It is expected that all PCI resources will be mapped by software into PCI Memory space, and not PCI I/O space. UltraSPARC IIi does provide a larger I/O space than did prior PCI-based UltraSPARC systems, so that devices that do use I/O space can be mapped to separate 8K pages for easier driver maintenance.

19.4.1.3 PCI Memory Space

PCI Memory cycles are generated by UltraSPARC IIi in response to PIO reads and writes to addresses in one of the PCI Memory Spaces.

As a bus master, UltraSPARC IIi will never generate Dual-Address-Cycles; all PCI addresses generated will be bits [31:0] of the 41 bit UltraSPARC IIi physical address.

The memory command used for the PCI transaction depends on the PIO transaction type, as shown in*Table 19-41*.

For PCI transactions with multiple data phases, UltraSPARC IIi will always use Linear Incrementing mode as defined by the PCI specification. Cache Line Toggle Mode is not used.

Compatibility Note – Unlike prior PCI-based UltraSPARC systems, UltraSPARC IIi does not use bit 31 of the PCI address for outgoing memory transactions, or bit 17 for outgoing IO transactions. APB also similarly preserves bits 31 and 17.

19.4.2 PCI Address Space—DMA

19.4.2.1 PCI Configuration Space

UltraSPARC IIi does not respond to any Configuration Read or Configuration Write cycles. UltraSPARC IIi/APB is the central resource for each PCI bus, and is expected to be the only device generating configuration cycles.

UltraSPARC IIi PIO accesses to target configuration registers within the PBM are serviced without generating a configuration cycle on the PCI bus.

Peer-to-peer transfers between two PCI devices on the same bus using Configuration Read or Configuration Write commands cannot be prohibited by UltraSPARC IIi or APB, but are not expected to occur, since UltraSPARC IIi/APB are the only devices that can drive the IDSEL# lines correctly.

19.4.2.2 PCI I/O Space

UltraSPARC IIi does not respond to I/O Read or I/O Write commands on the PCI bus.

Peer-to-peer transfers between two PCI devices on the same bus using I/O Read or I/O Write commands cannot be prohibited by UltraSPARC IIi, but they are not expected to occur, since all PCI resources are intended to be mapped into Memory Space.

19.4.2.3 PCI Memory Space

DMA, DMA (IOMMU bypass), and PCI peer-to-peer activity occurs in PCI Memory Space.The final destination and address translation of a PCI Memory transaction is based on these functions:

- Addressing mode used: 64-bit (DAC) vs. 32-bit (SAC)
- Whether the PCI address[31:29] is enabled as UltraSPARC IIi address space, by the PCI Target Address Space Register.
- Value of MMU_EN in the IOMMU Control Register

- Value of PCI address bits <63:50> in DAC mode

Table 19-42 shows the various ways that UltraSPARC IIi deals with PCI addresses as a PCI target device.

Table 19-42 PCI DMA Modes of Operation

Mode	Target Space Hit	MMU_EN	Addr<63:50>	Result
SAC	no	X	N/A	PCI peer-to-peer (Ignored by UltraSPARC IIi)
SAC	yes	0	N/A	Pass-through
SAC	yes	1	N/A	IOMMU Translation (DMA)
DAC	X	X	0x0000-0x3FFE	Ignored by UltraSPARC IIi
DAC	X	X	0x3FFF	Bypass (DMA)

Pass-through

In pass-through mode, physical addr<40:32> = 0x000, physical addr<31:0> = PCI_Addr<31:0>. Pass-through transfers always generate cacheable transactions.

Compatibility Note – Unlike prior PCI-based UltraSPARC systems, Pass-through does not zero PCI_Addr[31]

IOMMU Translation mode

In IOMMU translation mode, the physical address is obtained by performing a virtual to physical translation through the IOMMU. The value of the C bit in the TTE for the virtual page determines whether the transaction generated is cacheable or non-cacheable.

PCI peer-to-peer mode

In peer-to-peer mode, two devices on the same PCI bus transfer data without any involvement from UltraSPARC IIi. There is no address translation involved – the master device simply puts out the PCI address to which the target device has been mapped. If no device has been mapped there, the PCI master device terminates its cycle with a Master-Abort.

Bypass mode

In bypass mode, the physical address<33:0> = PCI_Addr<33:0>. Whether a cacheable or non-cacheable transaction is made is determined by the value of PCI_Addr<34>; a 0 in this bit specifies a cacheable transaction.

Compatibility Note – Prior PCI-based UltraSPARC systems used PCI_Addr<40>, but note that [40:34] are all 1's for UPA64S addresses.

19.4.2.4 Memory Burst Order

In all cases, UltraSPARC IIi only supports bursts as a target device in Linear Incrementing mode. If any of the reserved burst orders are used, UltraSPARC IIi will issue a target disconnect after the first data phase.

19.4.3 DMA Error Registers

Table 19-43 DMA Error Registers

Register	PA	Access Size
DMA UE AFSR	0x1FE.0000.0030	8 bytes
DMA CE AFSR	0x1FE.0000.0040	8 bytes
DMA UE/CE AFAR	0x1FE.0000.0038 or 0x1FE.0000.0048	8 bytes

19.4.3.1 DMA UE Asynchronous Fault Status/Address Register

UltraSPARC IIi IO logs any uncorrectable ECC error that it detects in the DMA UE AFSR/AFAR.

Uncorrectable errors can result from DMA read or DMA partial writes when memory does not Read-Modify-Write because it does not see an entire 16-bytes of write data. IOMMU errors can result from any DMA operation.

This register contains primary error status bits <63:61> and secondary error status bits <60:58>. *Only* one of the primary error status bits can be set at any time. Primary error status can only be set when:

- None of the primary error conditions exists prior to this error *or*

- A new error is detected at the same time as software is clearing the primary error; "at the same time" means on coincident clock cycles. Setting takes precedence over clearing.

Secondary bits are set whenever a primary bit is set. The secondary bits are cumulative and always indicate that information has been lost because no address information has been captured. Setting of the primary error bits is independent.

Compatibility Note – A PCI DMA UE interrupt is generated whenever a primary DMA UE or Translation Error bit is set, even if by a CSR write. Ensure that software clears the AFSR before clearing the interrupt state and re-enabling the PCI Error Interrupt. (This behavior is similar to that of the ECU AFSR)

Table 19-44 DMA UE AFSR

Field	Bits	Description	POR state	Type
Reserved	63	Read as 0	0	R0
P_DRD	62	Set if primary DMA UE or TE is caused by PCI read	0	R/ W1C
P_DWR	61	Set if primary DMA UE or TE is caused by PCI write	0	R/ W1C
Reserved	60	Reserved, read as 0	0	R0
S_DRD	59	Set if secondary DMA UE or TE is caused by PCI read.	0	R/ W1C
S_DWR	58	Set if secondary DMA UE or TE is caused by PCI write	0	R/ W1C
S_DTE	57	Set if secondary error is PCI DMA Translation Error	0	R/ W1C
P_DTE	56	Set if primary error is PCI DMA Translation Error	0	R/ W1C
Reserved	55:48	Read as 0	0	R0
BYTEMASK	47:32	0x00FF or 0xFF00, depending on [29] ==0 or 1	00FF	R
DW_OFFSET	31:29	DMA UE/CE AFAR bits [5:3]	0	R
Reserved	28:24	Read as 0	0	R0
BLK	23	Set if primary error is caused by PCI read	0	R
Reserved	22:0	Reserved, read as 0	0	R0

The AFAR and bits <47:23> of AFSR log the address and status of the primary DMA UE or error. A new DMA UE error is not logged into these bits until software clears the primary error to make the AFAR and part of the AFSR available to log the new error.

UltraSPARC IIi extension to DMA UE AFSR operation

To facilitate debug, errors due to invalid TTE entries in the IOMMU TSB or write protection errors are also logged in the DMA UE AFSR and AFAR. See the shaded entries in AFSR *Table 19-44*.

Compatibility Note – This feature is absent in prior PCI-based UltraSPARC systems but should be compatible with existing Solaris code.

The DWR, DRD bits, and a new bit, DTE, are set for this new case. Software should also get an error report from the DMA master that receives the Target Abort. This action provides the advantage of getting t the VA of the error in the DMA UE AFAR. Since this error indicates a software problem with the IOMMU TSB, software should be able to sort out the two possible error indications.

Note that the STA bit in the PCI Configuration Space Status register is also set, since UltraSPARC IIi generated a Target Abort.

19.4.3.2 DMA UE/CE Asynchronous Fault Address Register

The AFAR and bits <47:23> of AFSR log the address and status of the primary DMA UE or IOMMU error, and of the primary DMA CE.

After logging an address associated with a primary DMA UE, a further DMA UE error is not logged until software clears the DMA UE AFSR primary UE or IOMMU error bits, to make the AFAR and part of the AFSR available to log a new error.

This AFAR is also used for primary DMA CE address logging. Further DMA CE are not logged into these bits until software clears the primary error to make the AFAR and part of the AFSR available to log a new error. DMA UE or IOMMU errors, however, can always overwrite a value saved by a DMA CE primary error.The PA of the TTE entry is saved on Invalid, Protection (IOMMU miss), and TTE UE errors. If the Protection error had an IOMMU hit, the translated PA from the IOMMU is saved instead. This may occur if a prior DMA read caused the IOMMU entry to be installed.

Table 19-45 DMA UE/CE AFAR

Field	Bits	Description	POR state	Type
Reserved	63:41	Reserved, read as 0.	0	R0
UE/CE_PA	40:0	Physical address of error transaction.	0	R
0	2:0	Always 0	0	R0

19.4.3.3 DMA CE Asynchronous Fault Status/Address Register

UltraSPARC IIi logs the correctable ECC error in the DMA CE AFSR/AFAR. Correctable errors can occur during DMA read or DMA partial write operations. This register contains primary error status bits <63:61> and secondary error status bits <60:58>. *Only* one of the primary error status bits can be set at any time. Primary error status can be set only when:

- None of the primary error conditions exists prior to this error *or*
- A new error is detected at the same time as software is clearing the primary error; "at the same time" means on coincident clock cycles. Setting takes precedence over clearing.

Secondary bits are set whenever a primary bit is set. The secondary bits are cumulative and always indicate that information has been lost because no address information has been captured. Setting of the primary error bits is independent.

Compatibility Note – A DMA CE interrupt is generated whenever a primary DMA CE bit is set, even if by a CSR write. Ensure that software clears the AFSR before it clears the interrupt state and re-enables the PCI Error Interrupt. (This behavior is similar to that of the ECU AFSR).

Table 19-46 DMA CE AFSR

Field	Bits	Description	POR state	Type
Reserved	63	Reserved, read as 0	0	R0
P_DRD	62	Set if primary DMA CE is caused by PCI read	0	R/W1C
P_DWR	61	Set if primary DMA CE is caused by PCI write	0	R/W1C
Reserved	60	Reserved, read as 0	0	R0
S_DRD	59	Set if secondary DMA CE is caused by PCI read.	0	R/W1C

Table 19-46 DMA CE AFSR *(Continued)*

Field	Bits	Description	POR state	Type
S_DWR	58	Set if secondary DMA CE is caused by PCI write	0	R/ W1C
Reserved	57:56	Reserved, read as 0	0	R0
E_SYND	55:48	DMA CE Syndrome bits, logged on primary error.	0	R
BYTEMASK	47:32	0x00FF or 0xFF00, depending on [29] ==0 or 1	00FF	R
DW_OFFSET	31:29	DMA UE/CE AFAR bits [5:3]	0	R
Reserved	28:24	Read as 0	0	R0
BLK	23	Set if primary error is caused by PCI read	0	R
Reserved	22:00	Reserved, read as 0	0	R0

CHAPTER **20**

SPARC-V9 Memory Models

20.1 Overview

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 multi-processor environments. UltraSPARC IIi supports all three memory models.

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 control explicitly 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 processor enters RED_state.

A memory location is identified by an 8-bit Address Space Identifier (ASI) and a 64-bit virtual address. The 8-bit ASI may be obtained from a 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 UltraSPARC IIi 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 non-privileged 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 (non-cached with and without side-effects) spaces. 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.

20.2 Supported Memory Models

The following sections contain brief descriptions of the three memory models supported by UltraSPARC IIi. These definitions are for general illustration. Detailed definitions of these models can be found in *The SPARC Architecture Manual, Version 9*. The definitions in the following sections apply to system behavior as seen by the programmer. A description of MEMBAR can be found in Section 8.3.2, *Memory Synchronization: MEMBAR and FLUSH* on page 70.

Note – Stores to UltraSPARC IIi Internal ASIs, block loads, and block stores are outside the memory model; that is, they need MEMBARs to control ordering. See *Instruction Prefetch to Side-Effect Locations* on page 77 and Section 13.5.3, *Block Load and Store Instructions* on page 164.

Note – Atomic load-stores are treated as both a load and a store and can only be applied to cacheable address spaces.

20.2.1 TSO

UltraSPARC IIi implements the following programmer-visible properties in Total Store Order (TSO) mode:

- Loads are processed in program order; that is, there is an implicit MEMBAR `#LoadLoad` between them.
- Loads may bypass earlier stores. Any such load that bypasses such earlier stores must check (snoop) the store buffer for the most recent store to that address. A MEMBAR `#Lookaside` is not needed between a store and a subsequent load at the same noncacheable address.
- A MEMBAR `#StoreLoad` must be used to prevent a load from bypassing a prior store, if Strong Sequential Order is desired.

- Stores are processed in program order.
- Stores cannot bypass earlier loads.
- Accesses with the E-bit set (that is, those having side-effects) are all strongly ordered with respect to each other.
- An E-cache update is delayed on a store hit until all outstanding stores reach global visibility. For example, a cacheable store following a noncacheable store is not globally visible until the noncacheable store has reached global visibility; there is an implicit MEMBAR `#MemIssue` between them.

20.2.2 PSO

UltraSPARC IIi implements the following programmer-visible properties in Partial Store Order (PSO) mode:

- Loads are processed in program order; that is, there is an implicit MEMBAR `#LoadLoad` between them.
- Loads may bypass earlier stores. Any such load that bypasses such earlier stores must check (snoop) the store buffer for the most recent store to that address. For SPARC-V9 compatibility, a MEMBAR `#Lookaside` should be used between a store and a subsequent load to the same non-cacheable address.
- Stores cannot bypass earlier loads.
- Stores are not ordered with respect to each other. A MEMBAR must be used for stores if stronger ordering is desired. A MEMBAR `#MemIssue` is needed for ordering of cacheable after non-cacheable stores.
- Non-cacheable accesses with the E-bit set (that is, those having side-effects) are all strongly ordered with respect to each other, but not with non-E-bit accesses.

Note – The behavior of partial stores to noncacheable addresses (pages with the TTE.CP=0) is dependent on the system and I/O device implementation. UltraSPARC IIi generates a P_NCWR_REQ operation with a byte mask corresponding to the *rs2* mask of the partial store instruction. If the system interconnect or I/O device is unable to perform the write operation of the bytes specified by the byte mask, an error is *not* signaled back to the processor.

20.2.3 RMO

UltraSPARC IIi implements the following programmer-visible properties in Relaxed Memory Order (RMO) mode:

- There is no implicit order between any two memory references, either cacheable or non-cacheable, except that non-cacheable accesses with the E-bit set (that is, those having side-effects) are all strongly ordered with respect to each other.

- A MEMBAR must be used between cacheable memory references if stronger order is desired. A MEMBAR `#MemIssue` is needed for ordering of cacheable after non-cacheable accesses. A MEMBAR `#Lookaside` should be used between a store and a subsequent load at the same noncacheable address.

CHAPTER **21**

Code Generation Guidelines

21.1 Hardware / Software Synergy

One of the goals set for UltraSPARC IIi was for the processor to execute SPARC-V8 binaries efficiently, providing approximately three times the performance of existing machines running the same code. A significantly larger performance gain can be obtained if the code is re-compiled using a compiler specifically designed for UltraSPARC IIi. Several features are provided on UltraSPARC IIi that can only be taken advantage of by using modern compiler technology. This technology was not available previously, mainly because the hardware support was not sufficient to justify its development.

21.2 Instruction Stream Issues

21.2.1 UltraSPARC IIi Front End

The front end of the processor consists of the Prefetch Unit, the I-cache, the next field RAM, the branch and set prediction logic, and the return address stack. The role of the front end is to supply as many valid instructions as possible to the grouping logic and eventually to the functional units (the ALUs, floating-point adder, branch unit, load/store pipe, etc.).

21.2.2 Instruction Alignment

21.2.2.1 I-cache Organization

The 16 Kb I-cache is organized as a 2-way set associative cache, with each set containing 256 eight-instruction lines (*Figure 21-1*). The 14 bits required to access any location in the I-cache are composed of the 13 least significant address bits (since the minimum page size is 8K, these 13 bits are always part of the page offset and need not be translated) and one bit used to predict the associativity number (way) in which instructions reside. Out of a line of 8 instructions, up to 4 instructions are sent to the instruction buffer, depending on the address. If the address points to one of the last three instructions in the line, only that instruction and the ones (0-2) until the end of the line are selected (for simplicity and timing considerations, hardware support for getting instructions from two adjacent lines was not included). Consequently, on average for random accesses, 3.25 instructions are fetched from the I-cache. For sequential accesses, the fetching rate (4 instructions per cycle) equals or exceeds the consuming rate of the pipeline (up to 4 instructions per cycle).

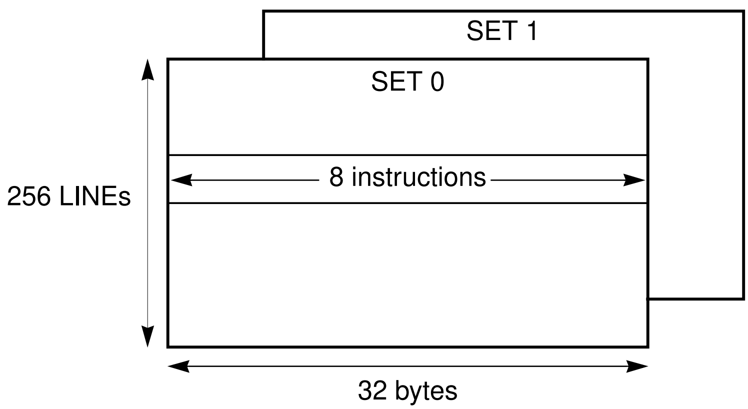


Figure 21-1 I-cache Organization

21.2.2.2 Branch Target Alignment

Given the restriction mentioned above regarding the number of instructions fetched from an I-cache access, it is desirable to align branch targets so that enough instructions are fetched to match the number of instructions issued in the first group of the branch target. For instance, if the compiler scheduler indicates that the target can only be grouped with one more instruction, the target should be placed anywhere in the line except in the last slot, since only one instruction would be fetched in that case. If the target is accessed from more than one place, it should be aligned so that it accommodates the largest possible group. If accesses to the I-cache are expected to miss, it may be desirable to align targets on a 16-byte (even 32-byte) boundary so that 4 instructions are forwarded to the next stage. Such an alignment can at least assure that four (eight for 32-byte alignment) instructions can be processed between cache misses, assuming that the code does not branch out of the sequence of instructions (which is generally *not* the case for integer programs).

21.2.2.3 Impact of the Delay Slot on Instruction Fetch

If the last instruction of a line is a branch, the next sequential line in the I-cache must be fetched even if the branch is predicted taken, since the delay slot must be sent to the grouping logic. This leads to inefficient fetches, since an entire E-cache access must be dedicated to fetching the missing delay slot. Take care not to place delayed CTIs (control transfer instructions) that are predicted *taken* at the end of a cache line.

21.2.2.4 Instruction Alignment for the Grouping Logic

UltraSPARC IIi can execute up to four instructions per cycle. The first three instructions in a group occupy slots that in most cases are interchangeable with respect to resources. Only special cases of instructions that can only be executed in IEU_1 followed by IEU_0 candidates violate this interchangeability (described in Section 22.5, *Integer Execution Unit (IEU) Instructions* on page 348). The fourth slot can only be used for PC-based branches or for floating-point instructions. Consequently, in order to get the most performance out of UltraSPARC IIi, the code should be organized so that either a floating-point operation (FPOP) or a branch is aligned with the fourth slot. For floating-point code, it should be relatively easy for the compiler to take advantage of the added execution bandwidth provided by the fourth slot. For integer code, aligning the branch so that it is issued fourth in a group must be balanced with other factors that may be more important, such as not placing a branch at the end of a cache line. Moreover if dependency analysis shows that a group of four instructions could be issued, but the fourth instruction is not a branch or an FPop while one of the first three is a branch, before switching the two instructions (assuming no data dependency), the compiler must evaluate the following trade-off:

- Moving the fourth instruction ahead of the branch (cross-block scheduling) and generating possible compensation code for the alternate path.
- Breaking the group and scheduling the ALU instruction with the next group. Notice that this may not lengthen the critical path (in terms of number of cycles executed) if the next group can accommodate this extra instruction without adding any new group.

21.2.2.5 Impact of Instruction Alignment on PDU

There is one branch prediction entry for every two instructions in the I-cache. Each entry, consisting of a two-bit field, indicates if the branch is predicted taken or not-taken (the state machine is described in Section 21.2.6). In addition to the branch prediction field, there is a *next field* associated with every four instructions. The next field contains the index of the line and the associativity number (or way) of the line that should be fetched next. For sequential code, the next field points to the next line in the I-cache. If a predicted taken branch is among the four instructions, the next field contains the index of the target of the branch.

The following cases represent situations when the prediction bits and/or the next field do not operate optimally:

1. When the target of a branch is word 1 or word 3 of an I-cache line (*Figure 21-2*) and the fourth instruction to be fetched (instruction 4 and 6 respectively) is a branch, the branch prediction bits from the wrong pair of instructions are used.

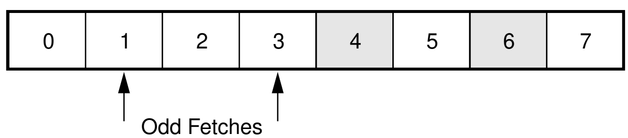


Figure 21-2 Odd Fetch to an I-cache Line

2. If a group of four instructions (instructions 0-3 or instructions 4-7) contains two branches and can be entered at a different position than the beginning of the group (other than instruction 0 and 4 respectively), the next field will contain the update from the latest branch taken in this group of four instructions, which may not be the one associated with the branch of interest (*Figure 21-3*).

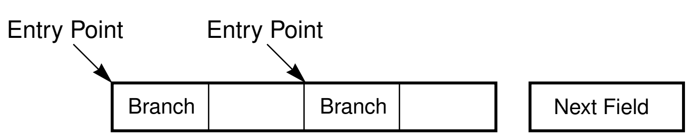


Figure 21-3 Next Field Aliasing Between Two Branches

3. Since there is one set of prediction bits for every two instructions, it is possible to have two branches (a CTI couple) sharing prediction bits. Under normal circumstances, the bits are maintained correctly; however, the bits may be updated based on the wrong branch if the second branch in the CTI couple is the target of another branch (*Figure 21-4*).

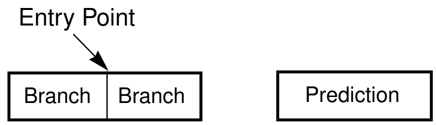


Figure 21-4 Aliasing of Prediction Bits in a Rare CTI Couple Case

As stated in Chapter 22, *Grouping Rules and Stalls*, if the addresses of the instructions in a group cross a 32-byte boundary, an implicit branch is “forced” between instructions at address 31 and 32 (low order bits). That rule has a performance impact only if a branch is in that specific group. Care should be taken not to place a branch in a group that crosses this boundary. *Figure 21-5* shows an example of this rule. A group containing instructions I0 (branch), I1, I2, and I3 will be broken, because an artificial branch is forced after address 31 and there is already a branch in the group.

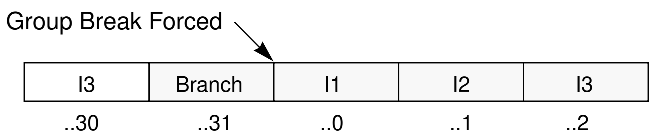


Figure 21-5 Artificial Branch Inserted after a 32-byte Boundary

21.2.3 I-cache Timing

If accesses to the I-cache hit, the pipeline rarely starves for instructions. Only in pathological cases is the PDU unable to provide a sufficient number of instructions to keep the functional units busy. For example, a taken branch to a taken branch sequence without any instructions between the branches (except for the delay slot) could only be executed at a peak rate of two instructions per cycle. Otherwise, up to 4 instructions are sent to the D Stage to be decoded and eventually dispatched in the G Stage and executed starting in the E Stage.

An I-cache miss does not necessarily result in bubbles being inserted into the pipeline. Part of the I-cache miss processing, or even all of it, can be overlapped with the execution of instructions that are already in the instruction buffer and are waiting to be grouped and executed. Moreover, since the operation of the PDU is somewhat separated from the rest of the pipeline, the I-cache miss may have occurred when the pipeline was already stalled (for example, due to a multi-cycle integer divide, floating-point divide dependency, dependency on load data that missed the D-cache, etc.). This means that the miss (or part of it) may be transparent to the pipeline.

When an I-cache miss is detected, normal instruction fetching is disabled and a request is sent to the E-cache for the line that is missing in the I-cache. A full line of eight instructions (32 bytes) is brought into the processor in two parts (the interface to the E-cache is 16-bytes wide). The critical part (that is, the 16 bytes containing the instruction that caused the miss) is brought in first. If a predicted taken branch is in the second 16-byte block brought into the I-cache, there will be a one cycle delay before the next fetch (this is the time needed to compute the next address).

Because of the possibility of stalling the processor for in the case when the pipeline is waiting for new instructions, it is desirable to try to make routines fit in the I-cache and avoid hot spots (collisions). UltraSPARC IIi provides instrumentation to profile a program and detect if instruction accesses generate a cache miss or a cache hit. For example, one can program performance counters to monitor I-cache accesses and I-cache misses. Then, by checkpointing the counters before and after a large section of code, combined with profiling the section of code, one can determine if the frequently executed functions generally hit or miss the I-cache. Instrumentation can be used in a similar manner to determine if a trap handler generally resides in the I-cache or causes a cache miss.

21.2.4 Executing Code Out of the E-cache

When frequently executed routines do not fit in the I-cache, it is possible to organize the code so that the main routines reside in the much larger E-cache and do not significantly affect the execution time. As an example we look at *fpppp*. Of the fourteen floating-point programs in SPECfp92, *fpppp* shows the highest I-cache miss rate (about 21%) per cache access, or about 6.0% per instruction. For comparison, the next highest is *doduc* with about a 3% miss per cache access, 1% per instruction. Even though the I-cache miss rate is significant, UltraSPARC IIi is barely affected by it (the impact is on CPI only 0.0084). It performs so well for the reasons:

- The code is organized as a large sequential block.
- Branches are predicted very well (over 90%).
- The instruction buffer almost always contains several instructions when an I-cache miss occurs (an average of about 6.6).
- The instruction buffer is filled faster (up to 4 instructions per cycle) than it is emptied.

All these factors contribute to reducing the apparent I-cache miss latency to 0.14 cycles on average for *fpppp*; that is, on average, the pipeline is stalled for 0.14 cycles when an I-cache miss occurs.

The effectiveness of the instruction buffer and the prefetcher on *fpppp* demonstrated that techniques (such as loop unrolling) that create large sequential blocks of code can be used efficiently on UltraSPARC IIi, even if these blocks do not fit in the

I-cache. On the other hand, for code properly scheduled to take advantage of the four issue slots on UltraSPARC IIi, the rate of instruction "consumption" may easily exceed the rate of instruction fetching, thus making I-cache misses more apparent.

21.2.5 uTLB and iTLB Misses

The one-entry uTLB contains the virtual page number and the associated physical page number of the line accessed last. If the line currently accessed is to the same page, the instructions from that line are simply forwarded to the next stage. If the line is from a different virtual page, the translation is obtained from the iTLB a cycle later. The cost of crossing a page boundary is thus one cycle (the smallest possible page size, 8K bytes, is assumed). This may or may not translate into a one cycle penalty for the whole processor. For a tight loop with code spanning over two pages, this cost may be significant, especially if the instruction buffer is empty at the time of the page crossing. For this reason, it is desirable to position short loops within a page (avoid page crossing).

An iTLB miss is handled by software through the use of the TSB, and takes about 32 cycles. Consequently, an iTLB miss may be very costly in terms of idle processor cycles. In order to minimize the frequency of iTLB misses, UltraSPARC IIi provides a large number of entries (64) in the iTLB and allows pages as large as 4Mbytes to be used. Nonetheless, techniques that allocate pages based on profiling are encouraged to further decrease the iTLB miss cost.

21.2.6 Branch Prediction

UltraSPARC IIi predicts the outcome of branches and fetches the next instructions likely to be executed based on that outcome. While this is all done dynamically in hardware, the compiler has an impact on the initialization of the state machine. The static bit provided by BPcc and FBPfcc instructions is used to set the state machine in either the **likely taken** state or the **likely not taken** state (*Figure 21-6*). For branches without prediction (Bicc, FBfcc), UltraSPARC IIi initializes the state machine to **likely not taken**. Notice that a branch initialized to **likely taken** does not produce a correct next field for the immediately following I-cache fetch, since it takes one extra cycle to generate the correct address (branch offset added to the PC). This results in two lost cycles for fetching instructions, which does not necessarily lead to a pipeline stall. This penalty is much less than the mispredicted branch penalty (four cycles) that would occur if the branch prediction bit was always ignored and a static prediction were used (for example, always taken). The state machine representing the algorithm used for branch prediction is represented in *Figure 21-6*. Note that this figure is identical to that shown in *Figure A-14* on page 378.

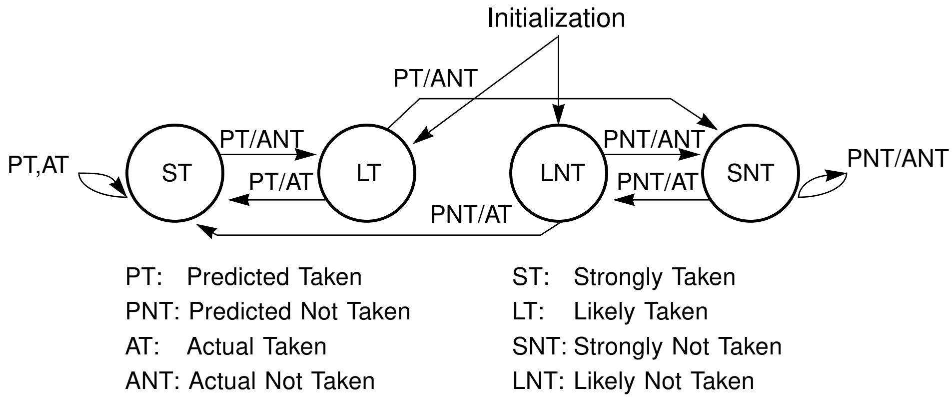


Figure 21-6 Dynamic Branch Prediction State Diagram

For loops in steady state, the algorithm is designed so that it requires two mis-predictions in order for the prediction to be changed from *taken* to *not taken*. Each loop exit will thus cause a single misprediction (versus two for a one-bit dynamic scheme).

21.2.6.1 Impact of the Annulled Slot

Grouping rules in Chapter 22, *Grouping Rules and Stalls*, describe how UltraSPARC IIi handles instructions following an annulling branch. In connection with these instructions, pay regard to the rules:

- Avoid scheduling multicycle instructions in the delay slot (for example, IMUL, IDIV, etc.).
- Avoid scheduling long latency instructions such as FDIV if the branch is predicted to be not-taken for a significant portion of the time (since they affect the timing of the non-taken stream).
- Avoid scheduling an instruction that would stall dispatching owing to a load-use dependency.
- Avoid scheduling WR(PR, ASR), SAVE, SAVED, RESTORE, RESTORED, RETURN, RETRY, and DONE in the delay slot and in the first three groups following an annulling branch.

21.2.6.2 Conditional Moves vs. Conditional Branches

The MOVcc and MOVR instructions provide an alternative to conditional branches for executing short code segments. UltraSPARC IIi differentiates the two as follows:

- Conditional branches: the branches are always resolved in the C stage. Distancing the SETcc from Bicc does not gain any performance. The penalty for a mispredicted branch is always four cycles. SETcc, Bicc, and the delay slot can be grouped together (*Figure 21-7*).

```
setcc G   E   C   N1  N2  N3  W
bicc  G   E   C   N1  N2  N3  W
delay G   E   C   N1  N2  N3  W
```

Figure 21-7 Handling of Conditional Branches

- Conditional moves: MOVcc and MOVR are dispatched as single instruction groups. Consequently, SETcc and MOVcc (or MOVR) cannot be grouped together (*vs*. SETcc and Bicc). Also, a use of the destination register for the MOVcc follows the same rule as a load-use (breaks group plus a bubble). *Figure 21-8* shows a typical example.

```
setcc G   E   C   N1  N2  N3  W
movcc     G   E   C   N1  N2  N3  W
use               G   E   C   N1  N2  N3  W
```

Figure 21-8 Handling of MOVCC

The use of FMOVR is more constrained than MOVcc. Besides having to wait for the load buffer to be empty, FMOVR and any younger IEU instructions must be separated by one group, even if there is no dependency between the IEU instruction and FMOVR.

Assuming that a specific branch can only be predicted with 50% accuracy (basically, it is not predicted), the compiler must balance the two cycle penalty on average for the mispredicted branch case against the ability to schedule other instructions around MOVcc (the SETcc cycle and the two groups after MOVcc, since MOVcc is a single instruction group). The need for multiple MOVcc instructions to guard multiple operations also must be taken into account.

21.2.7 I-cache Utilization

Grouping blocks that are executed frequently can effectively increase the apparent size of the I-cache. Cache studies show that, often, half of the I-cache entries are never executed. Placing rarely executed code out of a line containing a frequently executed block (identified by profiling) achieves a better I-cache utilization.

21.2.8 Handling of CTI couples

UltraSPARC IIi handles CTI couples by taking a “false” trap on the second CTI. It processes the first CTI, executes instructions until the second CTI reaches the N_3 stage, squashes all instructions executed after the first CTI, and executes instructions starting with the second CTI. Nine cycles are lost when CTI couples are encountered, which should discourage their use.

21.2.9 Mispredicted Branches

The dynamic branch prediction mechanism used for UltraSPARC IIi can generally achieve a success rate of 87% for integer programs and around 93% for floating-point programs (SPEC92). Correctly predicted conditional branches allow the processor to group instructions from adjacent basic blocks and continue progress speculatively until the branch is resolved. The capability of executing instructions speculatively is a significant performance boost for UltraSPARC IIi. On the other hand, when a branch is mispredicted, up to 18 instructions can be cancelled; This is the case when two instructions from the current group are cancelled along with four groups of four instructions, as shown in *Figure 21-9*—costly but, fortunately, this one case is very rare.

bicc	F	D	G	E	C	N_1	N_2	N_3	W				
delay	F	D	G	E	C	N_1	N_2	N_3	W				
instr1	F	D	G	E	C	N_1	N_2	N_3	W				
instr2	F	D	G	E	C	N_1	N_2	N_3	W				
grp1		F	D	G	E	C	N_1	N_2	N_3	W			
grp2			F	D	G	E	C	N_1	N_2	N_3	W		
grp3				F	D	G	E	C	N_1	N_2	N_3	W	
grp4					F	D	G	E	C	N_1	N_2	N_3	W
instr1 (correct)						F	D	G	E	C	N_1	N_2	N_3	W
...						...								

Figure 21-9 Cost of a Mispredicted Branch (Shaded Area)

Figure 21-9 shows how expensive badly behaved branches are for UltraSPARC IIi. Special effort should be made to predict branches that follow highly predictable branches based on profiling, and to combining conditions to make branches more predictable. Finally, if two or more branches are found to be correlated, it may be advantageous to duplicate common blocks to obtain separate branch predictions for hard-to-predict branches. For example in *Figure 21-10*, if the outcome of branch A, that is executed before branch B, has an impact on the direction of branch B, then it is preferable to split the code and duplicate the branch.

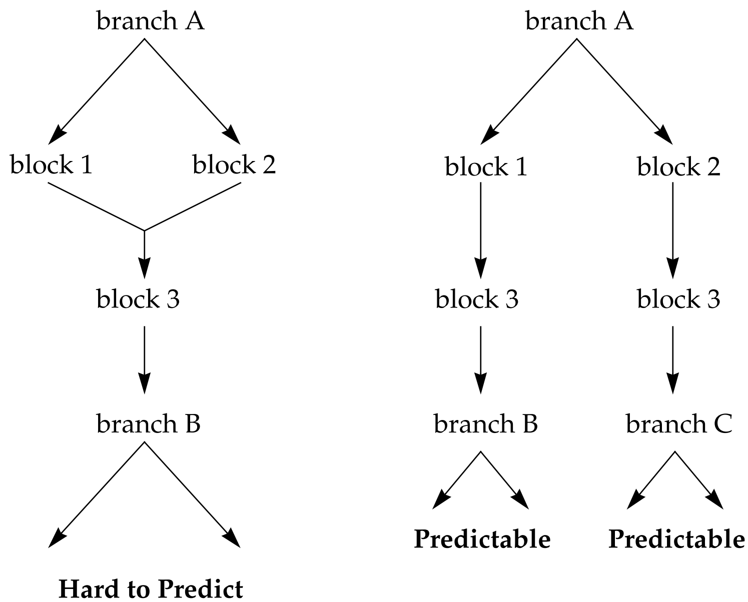


Figure 21-10 Branch Transformation to Reduce Mispredicted Branches

The technique, shown in *Figure 21-10*, can be generalized to *N* levels, where *N* branches are correlated and become more predictable. The above technique may lead to unrolling of loops that were previously identified as bad candidates because of the unpredictable behavior of their conditional branches.

21.2.10 Return Address Stack (RAS)

In order to speed up returns from subroutines invoked through CALL instructions, UltraSPARC IIi dedicates a 4-deep stack to store the return address. Each time a CALL is detected, the return address is pushed onto this RAS (Return Address Stack). Each time a return is encountered, the address is obtained from the top of the stack and the stack is popped. UltraSPARC IIi considers a return to be a JMPL or RETURN with *rs1* equal to `%o7` (normal subroutine) or `%i7` (leaf subroutine). The RAS provides a guess for the target address, so that prefetching can continue even though the address calculation has not yet been performed. JMPL or RETURN instructions using *rs1* values other than `%o7` or `%i7`, and DONE or RETRY instructions also use the value on the top of the RAS for continuing prefetching, but they do not pop the stack. See Section 17.1, *Overview* on page 251 for information about the contents of the RAS during RED_state processing.

21.3 Data Stream Issues

21.3.1 D-cache Organization

The D-cache is a 16K byte, direct mapped, virtually indexed, physically tagged (VIPT), write-through, non-allocating cache. It is logically organized as 512 lines of 32 bytes. Each line contains two 16-byte sub-blocks (see *Figure 21-11*).

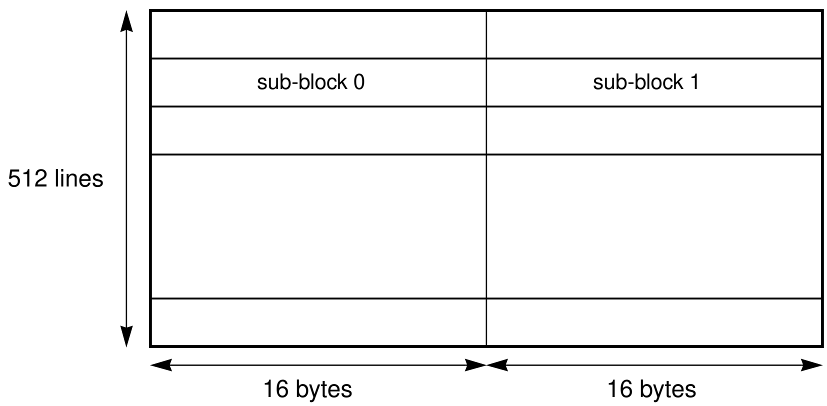


Figure 21-11 Logical Organization of D-cache

21.3.2 D-cache Timing

The latency of a load to the D-cache depends on the opcode. For unsigned loads, data can be used ***two cycles*** after the load. For instance, if the first two instructions in the instruction buffer are a load and an instruction dependent on that load, the grouping logic will break the group after the load and a bubble will be inserted in the pipeline the following cycle. Code compiled for an earlier SPARC processor with a load use penalty of one cycle will show a penalty of about.one CPI just for this rule; thus, it is very important to separate loads from their use.

21.3.2.1 Signed Loads

All signed loads smaller than 64 bits must be separated from their use by three cycles; otherwise, an extra bubble is inserted in the pipeline to force the separation between the load and its use. Floating-point loads are not sign extended, so they have a latency of two cycles.

Once a signed load (smaller than 64 bits) is encountered in the instruction stream, all subsequent consecutive loads (signed or unsigned) also return data in three cycles; otherwise, there would be a collision between two loads returning data. As soon as a cycle without a load appears in the pipeline, the latency of loads is brought back to two cycles.

Note – The SPARC-V8 LD instruction is replaced with LDUW in SPARC-V9; the new instruction does not require sign extension.

21.3.3 Data Alignment

SPARC-V9 requires that all accesses be aligned on an address equal to the size of the access. Otherwise a *mem_address_not_aligned* trap is generated. This is especially important for double precision floating-point loads, which should be aligned on an 8-byte boundary. If misalignment is determined to be possible at compile time, it is better to use two LDF (load floating-point, single precision) instructions and avoid the trap. UltraSPARC IIi supports single-precision loads mixed with double-precision operations, so that the case above can execute without penalty (except for the additional load). If a trap does occur, UltraSPARC IIi dedicates a trap vector for this specific misalignment, which reduces the overall penalty of the trap.

Grouping load data is desirable, since a D-cache sub-block can contain either four properly aligned single-precision operands or two properly aligned double-precision operands (eight and four respectively for a D-cache line). As we shall see later, this is desirable not only for improving the D-cache hit rate (by increasing its utilization density), but also for D-cache misses where, for sequential accesses, one out of two requests to the E-cache can be eliminated. Grouping load data beyond a D-cache sub-block is also desirable, since an E-cache line contains four D-cache sub-blocks (for a total of 64 bytes). Thus, sequential accesses can guarantee that only one E-cache miss will occur for loads that access up to four consecutive D-cache sub-blocks (two D-cache lines). Section 21.3.6 discusses how code scheduled for accessing data directly out of the E-cache can hide the extra latency introduced by D-cache misses.

Data alignment (right justification) for byte, halfword, and word accesses does not add latency to the loads unless superseded by the sign rule described in Section 21.3.2.1, *Signed Loads*. This is true whether the load goes to the register file or to internal pipeline bypasses.

21.3.4 Direct-Mapped Cache Considerations

A direct-mapped cache is more susceptible to collisions than a set-associative cache. It is possible to organize data at compile time so that collisions are minimized, however. For frequently executed loops, the compiler should organize the data so that all accesses within the loop are mapped to different cache lines, unless the access is to a line that is already mapped and the access is to the same *physical* line. For UltraSPARC IIi, this means that accesses should differ in the virtual address bits VA<13:5>. Hot spots can be detected by configuring the on-chip counters to accumulate D-cache accesses and D-cache misses. The counters can be turned on/off before/after the load of interest, or around a series of loads where hot spots are suspected to occur.

21.3.5 D-cache Miss, E-cache Hit Timing

Under normal circumstances (for example, no snoops, no arbitration conflict for the E-cache bus), loads that hit the E-cache are returned *N* cycles later than loads that hit the D-cache, where *N* is determined by the E-cache SRAM mode. *Table 21-1* shows the latency for all supported SRAM Modes. (See Section 1.3.3.1, *E-Cache SRAM Modes* on page 6 for more information.

Table 21-1 D-cache Miss, E-cache Hit Latency Depends on SRAM Mode

	SRAM Modes	
	2-2-2	**2–2**
No. of Cycles	9	7

If such a load (D-cache miss, E-cache hit) is immediately followed by a use, the group is broken and an *(N+1)*-cycle stall occurs; *Pipeline Example 21-1* illustrates this situation. (The figure shows a 8-cycle stall, which is consistent with 2–2 mode; 2–2–2 mode incurs a 10-cycle stall.)

Pipeline Example 21-1 D-cache Miss, E-cache Hit (2–2 mode shown)

load r_1	F	D	G	E	C	N_1	Q	Q	Q	Q	Q	Q					
use r_1	F	D	G	G	E	E	E	E	E	E	E	E	E	C	N_1	N_2	N_3
W																	

Group Break (*N*+1)-Cycle Stall Execution Resumes

Because of the high penalty associated with a load miss for code scheduled based on loads hitting the D-cache, UltraSPARC IIi provides hardware support for non-blocking loads through a load buffer that allows code scheduling based on *External* Cache (E-cache) hits.

21.3.6 Scheduling for the E-cache

Some applications have a working set that is too large to fit within the D-cache (they cause many capacity misses); others use data in patterns that generate many conflict-misses. Compilers c an schedule these applications to "bypass" the D-cache and access the data out of the E-cache.

Loads that miss the D-cache do not necessarily stall the pipeline (non-blocking loads). Instead, they are sent to the load buffer, where they wait for the data to be returned from the E-cache. The pipeline stalls only when an instruction that is dependent on the non-blocking load enters the pipeline before the load data is returned.

21.3.6.1 Mixing D-cache Misses and D-cache Hits

The UltraSPARC IIi "golden rule" is that all load data are returned *in order*. For instance if a load misses the D-cache, enters the load buffer, and is followed by a load that hits the D-cache, the data for the second (younger) load is not accessible. In this case, the younger load also must enter the load buffer; it will access the D-cache array only *after* the older load (D-cache miss) does so. If the load buffer is not empty, the D-cache array access is decoupled from the D-cache tag access; that is, it is performed some cycles after the tag access.

Note – Accessing blocked data in the D-cache while there is a load in the load buffer and scheduling the code so that operations can be performed on the blocked load data is *not* supported on UltraSPARC IIi. Data is always returned and operated upon *in order*.

---

on page 340 clarifies what is *not* supported without stalls on UltraSPARC IIi.

Pipeline Example 21-2 Load Hit Bypassing Load Miss (*Not Supported on UltraSPARC IIi*)

```
ld      [%l1+%g0],%l6 (D-cache miss)
ld      [%l2+%g0],%l7 (D-cache hit)
add     %l7,%g1,%g2   (use of D-cache hit)
add     %l6,%g1,%g3   (use of D-cache miss)
```

In , the first ADD stalls the pipeline until both the load miss and the load hit are handled. If the ADDs are interchanged, the first ADD can proceed as soon as the load miss is handled.

As a rule, if load latencies are expected to be a problem, the compiler should always schedule the use of loads in the same order that the loads appear in the program. While blocking part of an array in the D-cache and operating on the data during a previous D-cache miss may help reduce register pressure (three extra registers could be made available for an inner loop), the added complexity needed to handle conflicts in accessing the D-cache array offsets the potential benefits (for example, adding a port to the D-cache *vs*. adding a bubble on collisions).

21.3.6.2 Loads to the Same D-cache Sub-block

When a load enters the load buffer, the memory location loaded is compared to all other (older) loads in the buffer. If the other loads are to the same 16-byte sub-block, the entering load is marked as a *hit*, since by the time it accesses the D-cache array, the sub-block will be present (*Pipeline Example 21-3*). The detection of a hit eliminates a transaction to the E-cache, which results in making more slots available for other clients of the E-cache bus (I-cache, store buffer, snoops). Thus, it helps to organize the code so that data is accessed sequentially. This may involve interchanging loops so that array subscripts are incremented by one between each load access.

Pipeline Example 21-3 Interleaved D-cache Hits and Misses to Same Sub-block

```
.align start 16 bytes
ld       [start],%f0      (D-cache miss)
ld       [start + 8],%f2  (D-cache hit)
ld       [start + 16],%f4 (D-cache miss)
ld       [start + 24],%f6 (D-cache hit)
```

UltraSPARC IIi can access the E-cache only every other cycle. This still provides an average of 8 bytes per cycle, but only in 16-byte chunks.

21.3.6.3 Mixing Independent Loads and Stores

Note – The bus turnaround penalty is two cycles for systems running in 2-2-2 mode only; systems running in 2–2 mode incur *no* turnaround penalty.

Mixing reads and writes from and to the E-cache results in a penalty, caused by the difference in timing between reads and writes and also the bus turnaround time. UltraSPARC IIi automatically tends to separate loads and stores through the use of the load buffer and store buffer. The loads are given access to the E-cache, even if older stores have been waiting to access it. Only when the number of stores passes the “high-water mark” (5 stores) does the store buffer have priority. The code can be organized to further minimize the number of bus turnaround cycles. shows how loads and stores can be grouped so that only one turn-around penalty occurs (for a given state of the load buffer and store buffer). This can be accomplished with the help of a memory reference analyzer (Section 21.3.9, *Non-Faulting Loads*” covers this in more detail).

Code Example 21-1 Avoiding Bus Turnaround Penalties (1–1–1 mode only)

```
ld    [addr1],%l1        ld[addr1],%l1
st    [addr2],%l2        ld[addr3],%l3
ld    [addr3],%l3        st[addr2],%l2
st    [addr4],%l4        st[addr4],%l4
```

2 Penalties   1 Penalty

21.3.6.4 Using LDDF to Load Two Single-Precision Operands/Cycle

UltraSPARC IIi supports single cycle 8-byte data transfers into the floating-point register file for LDDF. Wherever possible, applications that use single-precision floating-point arithmetic heavily should organize their code and data to replace two LDFs with one LDDF. This reduces the load frequency by approximately one half, and cuts execution time considerably.

21.3.7 Store Buffer Considerations

The store buffer on UltraSPARC IIi is designed so that stores can be issued even when the data is not ready. More specifically, a store can be issued in the same group as the instruction producing the result. The address of a store is buffered until the data is eventually available. Once in the store buffer, the store data is buffered until it can be sent "quietly" (that is, without interfering with other instructions) to the D-cache, the E-cache, I/0 devices, or the frame buffer (for noncacheable stores).

To increase the throughput to the E-cache, which results in decreasing the frequency of the *store buffer full* condition, UltraSPARC IIi collapses two stores to the same 16 bytes of memory into one store. Since compression only occurs among two adjacent entries in the store buffer, the code should be organized so that multiple stores to the same "region" in memory are issued sequentially (increasing or decreasing order).

21.3.8 Read-After-Write and Write-After-Read Hazards

A Read-After-Write (RAW) hazard occurs when a load to the same address as an older outstanding store is issued. UltraSPARC IIi does not provide direct by-passing from intermediate stages of the store buffer to the various pipes that may result in pipeline stalls.

Most RAW hazards can be eliminated by proper register allocation and by eliminating spurious loads. Disassembled traces of various programs showed that most RAWs were "false" RAWs, and can be eliminated. However, some RAWs were "true" RAWs; they occur because two data structures point to the same memory location (through array indexes or pointers) without having knowledge that there could be a match between them. In order to simplify the hardware, the full 40 physical address bits are not used when comparing the address of the memory location requested by the load with the addresses associated with the stores in the store buffer. The rules are:

- The physical tag of the address is ignored
- If the load hits the D-cache, bits <13:0> of the address are used for comparison (byte granularity)

- If the load misses the D-cache, bits <13:4> of the address are used for comparison (sub-block granularity)

In order to cover both cache hits and cache misses, one should try to avoid RAWs based on a 16-byte boundary (using bits <13:4>). Even if a RAW occurs, the pipeline is not stalled until a use of the load data enters the pipeline (similar to the way loads are handled during D-cache misses). *Code Example 21-2* shows an example of back-to-back instructions causing a RAW hazard and a load-use. In the best scenario (that is, when the store buffer and load buffer are empty) the RAW hazard stalls the pipe for 8 cycles (versus one cycle for the normal load-use stall). This is mainly due to the fact that the store data enters the store buffer late in the pipe and that the load buffer must wait until the data is in the D-cache before it can access it.

Code Example 21-2 RAW Hazard Penalty

```
st      %l1,[addr1]      RAW Hazard
ld      [addr1],%l2
add     %l2,%l3,%l4
```

Under the Relaxed Memory Order (RMO) mode, stores can pass younger loads if a MEMBAR instruction has not been issued to prevent it. UltraSPARC IIi provides hardware detection of Write-After-Read (WAR) hazards so that a store to the same memory address as an older outstanding load does not pass that load. If a WAR hazard is detected, the store waits in the store buffer until the older load completes. The CPI penalties resulting from this only have a second-order effect on performance. The store buffer may fill up (rare), or an extra RAW hazard could be generated because stores stay in the store buffer longer.

21.3.9 Non-Faulting Loads

The ability to move instructions "up" in the instruction stream beyond conditional branches can effectively hide the latencies of long operations. This also increases the number of candidate instructions that the compiler can schedule without conflicts. SPARC-V9 provides *non-faulting* loads (equivalent to *silent loads* used for Multiflow TRACE and Cydrome Cydra-5), so that loads can be moved ahead of conditional control structures that guard their use. Non-faulting loads execute as any other loads, except that catastrophic errors, such as segmentation fault conditions, do not cause the program to terminate. The hardware and software (trap handler) cooperate so that the load appears to complete normally with a zero result. In order to minimize page faults when a speculative load references a NULL pointer (address zero), system software should map low addresses (especially address zero) to a page of all zeros and use the Non-Faulting Only (NFO) page attribute bit.

Simulations of general code percolation for UltraSPARC IIi have shown that there is much to be gained by using non-faulting loads. For integer programs the average group size (AGS) sent down the pipeline is 33% larger when code motion is allowed across one branch (using speculative loads) and 50% larger when instructions can be moved ahead of two branches.

CHAPTER **22**

Grouping Rules and Stalls

22.1 Introduction

This chapter explains in detail how to group instructions to obtain maximum throughput in UltraSPARC IIi. The following subsections explain the formatting conventions that make it easier to understand this information.

22.1.1 Textual Conventions

Rules are presented that consider instructions in three different ways:

Instructions: Actual SPARC-V9 and UltraSPARC IIi machine instructions are always written in Mixed Case BODY FONT. Examples are:

- FdMULq (Floating-point multiply double to quad—SPARC-V9)
- LDDF (Load Double Floating-Point Register—SPARC-V9)
- SHUTDOWN (Power Down Support—UltraSPARC IIi)

Instruction Families:

These are Groups of related SPARC-V9 instructions, introduced (but not described) in *The SPARC Architecture Manual, Version 9*. Instruction families are always written in **Mixed Case Bold Face Body Font**. Examples are:

- **BPcc** (Branch on Integer Condition Codes with Prediction) consists of the following instructions: BPA, BPCC, BPCS, BPE, BPG, BPGE, BPGU, BPL, BPLE, BPLEU, BPN, BPNE, BPNEG, BPPOS, BPVC, and BPVS.

- **FMOVcc** (Move Floating-Point Register on Condition) consists of the following instructions: FMOV{s,d,q}A, FMOV{s,d,q}CC, FMOV{s,d,q}CS, FMOV{s,d,q}E, FMOV{s,d,q}G, FMOV{s,d,q}GE, FMOV{s,d,q}GU, FMOV{s,d,q}L, FMOV{s,d,q}LE, FMOV{s,d,q}LEU, FMOV{s,d,q}N, FMOV{s,d,q}NE, FMOV{s,d,q}NEG, FMOV{s,d,q}POS, FMOV{s,d,q}VC, and FMOV{s,d,q}VS.

Instruction Classes: These are groups of SPARC-V9 and UltraSPARC IIi instructions that have similar effects. Instruction classes are always written in lower case italic body font. Examples are:

- *setcc* (any instruction that sets the condition codes)
- *alu* (any instruction processed in the Arithmetic and Logic Unit)

22.1.2 Example Conventions

Instructions are shown with offsets between their stages, to indicate the amount of latency that normally occurs between the instructions. The following instruction pair—*Pipeline Example 22-4*—has one cycle of latency:

Pipeline Example 22-4 Instruction with one cycle of latency

ADD	i1, i2, i6	G	E	C	N_1	N_2	N_3	W	
SLL	i6, 2, i8		G	E	C	N_1	N_2	N_3	W

This instruction pair shown in *Pipeline Example 22-5* has no latency.

Pipeline Example 22-5 Instruction with no latency

alu	→ r6	G	E	C	N_1	N_2	N_3	W
store	→ r6	G	E	C	N_1	N_2	N_3	W

22.2 General Grouping Rules

Up to four instructions can be dispatched in one cycle, subject to availability from the instruction buffer, execution resources, and instruction dependencies. UltraSPARC IIi has input (read-after-write) and output (write- after-write) dependency constraints, but no anti-dependency (write-after-read) constraints on instruction grouping.

Instructions belong to one or more of the following categories:

- Single group

- IEU
- Control transfer
- Load/store
- Floating-point/graphics

Note – CALL, RETURN, JMPL, **BPr**, PST and FCMP{LE,NE,GT,EQ}{16,32} belong to multiple categories.

22.3 Instruction Availability

Instruction dispatch is limited to the number of instructions available in the instruction buffer. Several factors limit instruction availability. UltraSPARC IIi fetches up to four instructions per clock from an aligned group of eight instructions. When the fetch address (**modulo** 32) is equal to 20, 24, or 28, then three, two, or one instruction(s) respectively are added to the instruction buffer. The next cache line and set are predicted using a next field and set predictor for each aligned four instructions in the instruction cache. When a set or next field mispredict occurs, instructions are not added to the instruction buffer for two clocks.

When an I-cache miss occurs, instructions are added to the instruction buffer as data is returned from the E-cache.

22.4 Single Group Instructions

Certain instructions are always dispatched by themselves to simplify the hardware. These instructions are: LDD(A), STD(A), block load instructions (LDDF{A} with an ASI of 70_{16}, 71_{16}, 78_{16}, 79_{16}, $F0_{16}$, $F1_{16}$, $F8_{16}$, $F9_{16}$), **ADDC{cc}**, **SUBC{cc}**, **{F}MOVcc**, {F}MOVr, SAVE, RESTORE, {U,S}MUL{cc}, MULX, MULScc, {U,S}DIV{X}, {U,S}DIVcc, LDSTUB{A}, SWAP{A}, CAS{X}A, LD{X}FSR, ST{X}FSR, SAVED, RESTORED, FLUSH{W}, ALIGNADDR, RETURN, DONE, RETRY, WR{PR}, RD{PR}, **Tcc**, SHUTDOWN, and the second control transfer instruction of a DCTI couple.

22.5 Integer Execution Unit (IEU) Instructions

IEU instructions can be dispatched only if they are in the first three instruction slots. A maximum of two IEU instructions can be executed in one cycle. There are two IEU pipelines: IEU_0 and IEU_1. The two data paths are slightly different, and some IEU instructions can be dispatched only to a particular pipeline. The following instructions can dispatched to either IEU pipeline: ADD, AND, ANDN, OR, ORN, SUB, XOR, XNOR and SETHI. These instructions can be grouped together or with older IEU_0 or IEU_1 specific instructions.

The IEU_0 data path has dedicated hardware for shift instructions: SLL{X}, SRL{X}. SRA{X}. Two shift instructions cannot be grouped together. Shift instructions can be grouped with older IEU_1 specific instructions, but they cannot be grouped with older non-specific IEU instructions. See *Pipeline Example 22-6*.

Pipeline Example 22-6 Showing allowable grouping of shift instructions

ADD	i1, i2, i6	G	E	C	N_1	N_2	N_3	W	
SLL	i6, 2, i8		G	E	C	N_1	N_2	N_3	W

The IEU_1 datapath has dedicated hardware for the condition-code-setting instructions: (TADDcc{TV}, TSUBcc{TV}, ADDcc, ANDcc, ANDNcc, ORcc, ORNcc, SUBcc, XORcc, XNORcc), **EDGE** and **ARRAY**. CALL, JMPL, **BPr**, **PST** and FCMP{LE,NE,GT,EQ}{16,32} also require the IEU_1 data path (besides counting as CTI, store, or floating-point instructions respectively), since they must access the integer register file. Two instructions requiring the use of IEU_1 cannot be grouped together; for example, only one instruction that sets the condition codes can be dispatched per cycle. An IEU_1 instruction can be grouped with older shift instructions and non-specific IEU instructions.

Note – For UltraSPARC IIi, a valid control transfer instruction (CTI) that was fetched from the end of a cache line is not dispatched until its delay slot also has been fetched.

22.5.1 Multi-Cycle IEU Instructions

Some integer instructions execute for several cycles and sometimes prevent the dispatch of subsequent instructions until they complete.

MULScc inserts one bubble after it is dispatched.

SDIV{cc} inserts 36 bubbles, UDIV{cc} inserts 37 bubbles, and {U,S}DIVX inserts 68 bubbles after they are dispatched.

MULX, and {U,S}MUL{cc} delay dispatching subsequent instructions for a variable number of clocks, depending on the value of the *rs1* operand. Four bubbles are inserted when the upper 60 bits of *rs1* are zero, or for signed multiplies when the upper 60 bits of *rs1* are one. Otherwise, an additional bubble is inserted each time the upper 60 bits of *rs1* are not all zeros (or all ones for signed multiplies) after arithmetic right shifting *rs1* by two bits. This implies a maximum of 18 bubbles for SMUL{cc}, 19 bubbles for UMUL{cc}, and 34 bubbles for MULX.

WR{PR} inserts four bubbles after it is dispatched. RDPR from the CANSAVE, CANRESTORE, CLEANWIN, OTHERWIN, FPRS, and WSTATE registers, and RD from *any* register are not dispatchable until four clocks after the instruction reaches the first slot of the instruction buffer.

Writes to the TICK, PSTATE, and TL registers and FLUSH{W} instructions cause a pipeline flush when they reach the W Stage, effectively inserting nine bubbles.

22.5.2 IEU Dependencies

Instructions that have the same destination register (in the same register file) cannot be grouped together, unless the destination register is `%g0`. *Pipeline Example 22-7* gives such an instance.

Pipeline Example 22-7 Instructions with the same destination cannot be grouped together

alu	→ i6	G	E	C	N_1	N_2	N_3	W	
load	→ i6		G	E	C	N_1	N_2	N_3	W

Instructions that reference the result of an IEU instruction cannot be grouped with that IEU instruction, unless the result is being stored in `%g0` See *Pipeline Example 22-8*.

Pipeline Example 22-8 Instructions cannot be grouped with the IEU instruction whose result they reference, unless stored in %g0

alu	→ i6	G	E	C	N_1	N_2	N_3	W	
LDX	[i6+i1], i8		G	E	C	N_1	N_2	N_3	W

There are two exceptions to this rule: Integer stores can store the result of an IEU instruction other than FCMP{LE,NE,GT,EQ}{16,32} and be in the same group—see *Pipeline Example 22-9*:

Pipeline Example 22-9 Exception to rule of *Pipeline Example 22-8*

alu	→ r6	G	E	C	N_1	N_2	N_3	W
store	→ r6	G	E	C	N_1	N_2	N_3	W

Also, **BPicc** or **Bicc** can be grouped with an older instruction that sets the condition codes as in *Pipeline Example 22-10*.

Pipeline Example 22-10 Grouping **BPicc** or **Bicc** instructions

Group1	*seticc*	G	E	C	N_1	N_2	N_3	W
	BPicc	G	E	C	N_1	N_2	N_3	W

Instructions that read the result of a **MOVcc** or **MOVr** cannot be in the same group or the following group; see *Pipeline Example 22-11*.

Pipeline Example 22-11 Grouping for instructions that read results of **MOVcc** or **MOVr**

MOVcc	%xcc, 0, i6	G	E	C	N_1	N_2	N_3	W		
LDX	[i6+i1], i8			G	E	C	N_1	N_2	N_3	W

Instructions that read the result of an FCMP{LE,NE,GT,EQ}{16,32} (including stores) cannot be in the same group or in the two following groups. STD is treated as dependent on earlier **FCMP** instructions, regardless of the actual registers referenced—*Pipeline Example 22-12*.

Pipeline Example 22-12 Rule for instructions that read the result of an FCMP{LE,NE,GT,EQ}{16,32}

FCMPLE32	f2, f4, i6	G	E	C	N_1	N_2	N_3	W			
LDX	[i6+i1], i8				G	E	C	N_1	N_2	N_3	W

In some cases, UltraSPARC IIi prematurely dispatches an instruction that uses the result of an FCMP{LE,NE,GT,EQ}{16,32}; it then cancels the instruction in the W Stage and refetches it. This effectively inserts nine bubbles into the pipe. To avoid this, software should explicitly force the use instruction to be in the *third* group or later after the FCMP{LE,NE,GT,EQ}{16,32}.

MULX, {U,S}MUL{cc}, MULScc, {U,S}DIV{X}, {U,S}DIVcc, and STD cannot be in the two groups following an FCMP{LE,NE,GT,EQ}{16,32}—see *Pipeline Example 22-13*.

Pipeline Example 22-13 MULX cannot be in the two groups following FCMP{LE,NE,GT,EQ}{16,32}

FCMPLE32	f2, f4, i6	G	E	C	N_1	N_2	N_3	W			
MUL	i8,i7,i9				G	E	C	N_1	N_2	N_3	W

FMOVr cannot be in the same group or in the group following an IEU instruction, even if it does not reference the result of the IEU instruction. It cannot be in the same group (*Pipeline Example 22-14*) or the next two groups (*Pipeline Example 22-15*) following an FCMP{LE,NE,GT,EQ}{16,32}.

Pipeline Example 22-14 FMOVr i5,i7 must be at least two groups ahead of an IEU instruction

ADD	i1, i2, i6	G	E	C	N_1	N_2	N_3	W		
FMOVr	i5,i7			G	E	C	N_1	N_2	N_3	W

Pipeline Example 22-15 **FMOVr** cannot be in the next two groups following an FCMP{LE,NE,GT,EQ}{16,32}

FCMPLE16 → i6	G	E	C	N_1	N_2	N_3	W			
FMOVr i5				G	E	C	N_1	N_2	N_3	W

22.6 Control Transfer Instructions

One Control Transfer Instruction (CTI) can be dispatched per group. The following control transfer instructions are not single group instructions: CALL, **BPcc**, **Bicc**, **FB(P)fcc**, **BPr**, and JMPL. CALL and JMPL are always dispatched as the oldest instruction in the group; that is, a group break is forced before dispatching these instructions.

DONE**,** RETRY, and the second instruction of a delayed control transfer instruction (DCTI) couple flush the pipe when they reach the W Stage, effectively inserting nine bubbles into the pipe. The pipeline is flushed even if the second DCTI is annulled.

22.6.1 Control Transfer Dependencies

UltraSPARC IIi can group instructions following a control transfer with the control transfer instruction. Instructions following the delay slot come from the predicted instruction stream. Examples for a branch predicted taken and a branch predicted not taken are shown in *Pipeline Example 22-16* and *Pipeline Example 22-17* respectively.

Pipeline Example 22-16 Branch predicted taken

Group 1	*setcc*	G	E	C	N_1	N_2	N_3	W
	BPcc	G	E	C	N_1	N_2	N_3	W
	FADD (delay slot)	G	E	C	N_1	N_2	N_3	W
	FMUL (branch target)	G	E	C	N_1	N_2	N_3	W

Pipeline Example 22-17 Branch predicted not taken

Group 1	*setcc*	G	E	C	N_1	N_2	N_3	W
	BPcc	G	E	C	N_1	N_2	N_3	W
	FADD (delay slot)	G	E	C	N_1	N_2	N_3	W
	FDIV (sequential)	G	E	C	N_1	N_2	N_3	W

If the delay slot of a DCTI is aligned on a 32-byte address boundary (that is, the DCTI is the last instruction in a cache line and the delay slot contains the first instruction in the *next* cache line), then the DCTI *cannot* be grouped with instructions pcfrom the predicted stream.—*Pipeline Example 22-18*.

Pipeline Example 22-18 Case when DCTI *cannot* be grouped with instructions from the predicted stream

Group 1	*setcc*	G	E	C	N_1	N_2	N_3	W	
	BPcc	G	E	C	N_1	N_2	N_3	W	
	FADD (32-byte aligned)	G	E	C	N_1	N_2	N_3	W	
Group 2	**FMUL** (branch target)		G	E	C	N_1	N_2	N_3	W

If the second instruction of the predicted stream is aligned on a 32-byte address boundary, then the DCTI *cannot* be grouped with that instruction—*Pipeline Example 22-19*

Pipeline Example 22-19 Cannot group DCTI with second instruction of predicted stream if it is on a 32-byte boundary

Group 1	**BPcc**	G	E	C	N_1	N_2	N_3	W	
	ADD (delay slot)	G	E	C	N_1	N_2	N_3	W	
	FADD	G	E	C	N_1	N_2	N_3	W	
Group 2	**FMUL** (32-byte aligned)		G	E	C	N_1	N_2	N_3	

The delay slot of a DCTI cannot be grouped with instructions from the predicted stream of another DCTI following the delay slot—*Pipeline Example 22-20*.

Pipeline Example 22-20 Cannot group DCTI delay slot with instructions from predicted stream of following DCTI

Group 1	**FADD** (delay slot 1)	G	E	C	N_1	N_2	N_3	W	
	BPcc	G	E	C	N_1	N_2	N_3	W	
	ADD (delay slot 2)	G	E	C	N_1	N_2	N_3	W	
Group 2	**FMUL** (branch target)		G	E	C	N_1	N_2	N_3	W

When a control transfer is mispredicted, the instruction buffer and instructions younger than the delay slot in the pipe are flushed, effectively inserting four bubbles in the pipe. An **FDIV** or **FSQRT** in the mispredicted stream causes dependent instructions in the correct branch stream to stall until the **FDIV** or **FSQRT** reaches the W_1 Stage[1]. *Pipeline Example 22-21* shows the case If the branch in the previous example was predicted not taken but actually were taken.

Pipeline Example 22-21 Stall after mispredicted control transfer

Group 1	*setcc*	G	E	C	N_1	N_2	N_3	W		
	BPcc (mispredicted)	G	E	C	N_1	N_2	N_3	W		
	FADD (delay slot)	G	E	C	N_1	N_2	N_3	W		
	FMUL → f0 (sequential)	G	E	C	N_1	N_2	N_3	W	W_1	
Group 2	**FMUL** f0,f0,f0 (branch target)								G	E

If an annulling branch is predicted not taken, the delay slot is still dispatched.

Multicycle instructions (except load instructions) run to completion, even if the delay slot instruction is annulled—*Pipeline Example 22-22*.

Pipeline Example 22-22 Multicycle instructions complete when delay-slot instruction is annulled

BPcc, a (not taken)	G	E	C	N_1	N_2	N_3	W		
imul (delay slot)		G	E	E	E	E	E	E	. . .

The *imul* unit is busy for the duration of the multiply.

An annulled delay slot, other than a load, affects subsequent dependency checking until the delay slot reaches the W_1 Stage—*Pipeline Example 22-23*.

Pipeline Example 22-23 Annulled delay-slot affects subsequent dependency checking

Group 1	**BPcc, a** (not taken)	G	E	C	N_1	N_2	N_3	W		
	FDIV → f0 (delay slot)		G	E	C	N_1	N_2	N_3	W	W_1

1. The W_1 Stage is a virtual stage that is normally not visible to the programmer.

Pipeline Example 22-23 Annulled delay-slot affects subsequent dependency checking

Group 2	**FADD** f0,f0,f1 (sequential)									G

In the example above, the **FADD** instruction is stalled in issue until the **FDIV** instruction completes.

A predicted annulled load does not affect dependency checking after it is dispatched—*Pipeline Example 22-24*.

Pipeline Example 22-24 Predicted annulled load does not affect dependency checking after dispatch

Group 1	**BPcc, a** (predicted not taken)	G	E	C	N_1	N_2	N_3	W	
	fld → f0 (delay slot)	G	E	C	N_1	N_2	N_3	W	
Group 2	**FADD** f0,f0,f1 (sequential)		G	E	C	N_1	N_2	N_3	W

An annulled load use or floating-point use is treated as a dependent instruction until the N_2 Stage of the branch—*Pipeline Example 22-25*.

Pipeline Example 22-25 Use treated as a dependent instruction

Group 1	**FADD** f7,f7,f6	G	E	C	N_1	N_2	N_3	W		
	Bcc, a (not taken)	G	E	C	N_1	N_2	N_3	W		
	bubble(2)									
Group 2	**FADD** f6,f7,f8				G	*flushed*				
Group 3	**FADD** f6,f7,f8					G	E	C	N_1	N_2

If the annulling branch is grouped with a delay slot containing a load use, the group will pay the full load use penalty even if the load use is annulled. This is because the branch is not resolved until the use stall is released.

WR{PR}, SAVE, SAVED, RESTORE, RESTORED, RETURN, RETRY, and DONE are stalled in the G-stage until earlier annulling branches are resolved, even if they are not in the delay slot. This means that they cannot be dispatched in the same group or the first three groups following an annulling branch instruction; see *Pipeline Example 22-26*.

Pipeline Example 22-26 Some instructions cannot be dispatched within three groups of an annulling branch instruction

Bicc, a	G	E	C	N_1	N_2	N_3	W		
SAVE					G	E	C	N_1	N_2

LDD{A}, LDSTUB{A}, SWAP{A} and CAS{X}A are stalled in the G-stage if there is a delayed control transfer instruction in the E Stage or C Stage; see *Pipeline Example 22-27*.

Pipeline Example 22-27 Instructions that stall for delayed control transfer instruction

Bicc	G	E	C	N_1	N_2	N_3	W	
Bubble(2)								
LDD				G	E	C	N_1	N_2

22.7 Load / Store Instructions

Load / store instructions can be dispatched only if they are in the first three instruction slots. One load/store instruction can be dispatched per group. Load / store instructions other than single group are: LD{SB,SH,SW,UB,UH,UW,X}{A}, LD{D}F{A}, ST{B,H,W,X}{A}, STF{A}, STDF{A}, JMPL, MEMBAR, STBAR, PREFETCH{A}.

LDD{A}, STD{A}, LDSTUB{A}, SWAP{A} will not dispatch younger instructions for one clock after they are dispatched. CAS{X}A will not dispatch younger instructions for two clocks after they are dispatched.

Loads are not stalled on a cache miss, instead they are enqueued in the load buffer until data can be returned. Load data is returned in the order that loads are issued, so a cache miss forces subsequent load hits to be enqueued until the older load miss data is available.

Stores are not stalled on a cache miss. Stores are enqueued in the store buffer until data can be written to the E-cache SRAM for cacheable accesses, to PCI or UPA64S for noncacheable accesses, or to the internal register for internal ASIs. Store data is written in the order that stores are issued, so a cache miss forces subsequent store hits to remain enqueued until the older store miss data is written out.

22.7.1 Load Dependencies and Interaction with Cache Hierarchy

Instructions that reference the result of a load instruction cannot be grouped with the load instruction or in the following group unless the register is `%g0; see` *Pipeline Example 22-28*.

Pipeline Example 22-28 Grouping instructions that reference the result of a load instruction

LDDF	[r1], f6 (not enqueued)	G	E	C	N_1	N_2	N_3	W	
Bubble(1)									
FMULd	f4, f6, f8			G	E	C	N_1	N_2	N_3

Single-precision floating-point loads lock the double register containing the single precision *rd* for data dependency checking—*Pipeline Example 22-29*.

Pipeline Example 22-29 Single-precision floating-point loads

LDF	[r1], f6 (not enqueued)	G	E	C	N_1	N_2	N_3	W	
Bubble(1)									
FMULs	f7, f7, f8			G	E	C	N_1	N_2	N_3

Instructions other than floating-point loads that have the same destination register as an outstanding load are treated the same as a source register dependency—*Pipeline Example 22-30*.

Pipeline Example 22-30 Instructions other than floating-point loads

load	i6 (not enqueued)	G	E	C	N_1	N_2	N_3	W	
ADD	i2, i1, i6			G	E	C	N_1	N_2	N_3

When an instruction referencing a load result enters the E Stage and the data is not yet returned, all instructions in the E Stage and earlier will be stalled. If there are multiple load uses, then all E-Stage and earlier instructions will be stalled until loads that have dependencies return data. E-Stage stalls can occur when referencing the result of a signed integer load, a load that misses the D-cache or a D-cache load hit whose data is delayed following one of the two previous cases.

22.7.1.1 Delayed Return Mode

Signed integer loads that hit the D-cache cause UltraSPARC IIi to enter delayed return mode. In delayed return mode, an extra clock of delay is added to all returning load data. UltraSPARC IIi remains in delayed return mode until some load other than a signed integer D-cache hit can return data in the normal time without colliding with a delayed return mode load.

22.7.1.2 Cache Timing

The following example illustrates D-cache hit timing. The first load causes UltraSPARC IIi to enter delayed return mode, returning data in the N_1 Stage. The second load is also in delayed return mode returning data in its N_1 Stage, otherwise it would collide with the first load data. The group containing the third load and the first ADD (which references the first load data) is stalled in the E Stage for one clock until both load uses by the first ADD have returned data. Since the third load is stalled in E, its normal C Stage data return will not collide with a previous delayed return mode load. This allows the last ADD to avoid an E Stage stall. If the third load were not grouped with the first ADD, it would not be stalled in the E Stage, and the last ADD would be dispatched one clock earlier. The third load causes the pipeline to exit delayed return mode.

Pipeline Example 22-31 Illustrating D-cache hit timing

Group 1	LDSB	[i1], i6 (D-cache hit)	G	E	C	N_1	N_2	N_3	W		
Group 2	LDB	[i3], i7 (D-cache hit)		G	E	C	N_1	N_2	N_3	W	
	Bubble(1)										
Group 3	LDB	[i7], i4 (D-cache hit)				G	E	E	C	N_1	
Group 4	ADD	i6,i7,i8				G	E	E	C	N_1	N_2
	Stall										
	Bubble(2))										
Group 5	ADD	i4,i5,i91ta							G	E	C

22.7.1.3 Block Memory Accesses

Unlike other loads, block loads do not lock all of their destination registers. If there are two block loads outstanding, any instruction except a block store is held in the G-stage until the first block load leaves the load buffer. A block load leaves the load buffer when its first word of data has returned.

22.7.1.4 Read-After-Write and Interaction with Store Buffer

If a load hits the D-cache and overlaps a store in the store buffer, the load does not return data until two clocks after the store updates the D-cache. The overlap check is pessimistic, because only the lower 14 bits of the effective memory address are checked. If a store is issued one clock earlier than an overlapping load that hits the D-cache, the load data is returned seven clocks later than normal. If a load misses the D-cache and if bits 13..4 of the load's effective memory address are the same as a store in the store buffer, the load data is not returned until six clocks after the store leaves the store buffer. If a store is issued one clock earlier than a D-cache miss load and bits 13..4 of the address are the same, the load data is returned six clocks later than a normal D-cache miss load.

MEMBAR `#StoreLoad` or `#MemIssue` blocks younger loads from returning data until three clocks after no older stores are outstanding (see Section 22.7.2, *Store Dependencies* on page 359). In the best case, a load use is stalled in the E Stage until 15 clocks after the previous store is dispatched.

22.7.1.5 Other Timing Issues

LD{X}FSR blocks dispatch of younger floating-point / graphics instructions that reference floating-point registers, **FB{P}fcc**, **MOVfcc**, ST{X}FSR, and LD{X}FSR instructions until four clocks after the data is returned in delayed return mode, and five clocks after the load data is returned otherwise. For example, if there are no outstanding load misses from the D-cache:

Pipeline Example 22-32 LD{X}FSR blocks FP instruction issue.

LDFSR (D-cache hit)	G	E	C	N_1	N_2	N_3	W	W_1	W_2
FMULS f7,f7,f8									G

LDD{A} instructions are held in the G-stage until three clocks after the N_3 Stage, or until older loads have returned data. If LDD{A} is dispatched and a miss occurs on an N_2 Stage or earlier load, the instruction will be canceled in the W Stage and fetched again. It will then be held in the G-stage until three clocks after older loads have returned data.

FLUSH{W}, **{F}MOVr**, **MOVcc**, RDFPRS, STD{A}, loads and stores from an internal ASI (4x-6x, 76, 77), SAVE, RESTORE, RETURN, DONE, RETRY, WRPR, and MEMBAR `#Sync` instructions cannot be dispatched until three clocks after older loads have returned data. The instruction is stalled in the G-stage until the N_3 Stage of the earliest outstanding load, if the load is not enqueued. For example:

Pipeline Example 22-33 Some instructions must wait three clocks from data return of prior loads

load (not enqueued)	G	E	C	N_1	N_2	N_3	W		
SAVE						G	E	C	N_1

LD{SB,SH,SW,UB,UH,UW,X}{A}, LD{D}F{A}, LDD{A}, LDSTUB{A}, SWAP{A}, CAS{X}A, LD{X}FSR, MEMBAR `#MemIssue` and MEMBAR `#StoreLoad` are held in the G-stage if there are already nine outstanding loads. A load is considered outstanding from the clock that it enters the E Stage through the clock that it returns data.

22.7.2 Store Dependencies

A store is considered outstanding from the clock that it enters the E-stage until two clocks after the data leaves the store buffer. Data leaves the store buffer when the write is issued to the E-cache SRAM for cacheable accesses, to PCI or UPA64S for noncacheable accesses, and to internal register for internal ASI. If there is no extra delay, a noncacheable store or cacheable store that misses the D-cache is outstanding for ten clocks after it is dispatched. An internal ASI or cacheable store that hits the D-cache is outstanding for eleven clocks after it is dispatched. If the last two stores in the store buffer are writing to the same 8-byte block and both are ready to go to the E-cache, the store buffer compresses the two entries into one. This reduces the number of outstanding stores by one. Compression is repeated as long as the last two entries are ready to go and are compressible. There is additional compression of sequential 8-byte stores tp UPA64S.

ST{B,H,W,X}{A}, STF{A}, STDF{A}, STD{A}, LDSTUB{A}, SWAP{A}, CAS{X}A, FLUSH, STBAR, MEMBAR `#StoreStore`, and MEMBAR `#LoadStore` are not dispatched if there are already eight outstanding stores. A block store counts as eight outstanding stores when it is dispatched.

If bits 13..4 of a store's effective memory address are the same as an older load in the load buffer, the store remains outstanding until four clocks after the load is not outstanding.

See *Event Ordering on UltraSPARC IIi* on page 429 for other details of event ordering.

LDSTUB, SWAP, CAS{X}A, store to internal ASI, block store, FLUSH, and MEMBAR `#Sync` instructions are not dispatched until no older stores are outstanding. The maximum rate of internal ASI stores or atomics is one every 12 clocks.

ST{X}FSR cannot be dispatched in the two groups following another ST{X}FSR.

PDIST cannot be dispatched in the group after a floating-point store or when a block store is outstanding.

22.8 Floating-Point and Graphic Instructions

Floating-point and graphics instructions that reference floating-point registers are divided into two classes: A and M. Two of these instructions can be dispatched together only if they are in different classes.

A Class:

F{i,x}TO{s,d}, F{s,d}TO{d,s}, F{s,d}TO{i,x}, FABS{s,d}, FADD{s,d}, FALIGNDATA, FAND{s}, FANDNOT1{s}, FANDNOT2{s}, FCMP{E}{s,d}, FEXPAND, FMOVr{s,d}, FMOV{s,d}cc, FNAND{s}, FNEG{s,d}, FNOR{s}, FNOT1{s}, FNOT2{s}, FONE{s}, FOR{s}, FORNOT1{s}, FORNOT2{s}, FPADD{16,32}{s}, FPMERGE, FPSUB{16,32}{s}, FSRC1{s}, FSRC2{s}, FSUB{s,d}, FXNOR{s}, FXOR{s}, and FZERO{s}.

M Class:

FCMP{LE,NE,GT,EQ}{16,32}, FDIST, FDIV{s,d}, FMUL{d}8SUx16, FMUL{d}8ULx16, FMUL{s,d}, FMUL8x16{AL,AU}, FPACK{16,32,FIX}, FsMULd, and FSQRT{s,d}.

FDIV{s,d}, FSQRT{s,d}, and FCMP{LE,NE,GT,EQ}{16,32} instructions break the group; that is, no earlier instructions are dispatched with these instructions.

22.8.1 Floating-Point and Graphics Instruction Dependencies

Instructions that have the same destination register (in the same register file) cannot be grouped together. For example:

Pipeline Example 22-34 Instructions with the same destination register cannot be grouped

FADD	f 2, f2, f6	G	E	C	N_1	N_2	N_3	W	
LDF [r0+r1], f6			G	E	C	N_1	N_2	N_3	W

FBfcc cannot be grouped with an older FCMP{E}{s,d}, even if they reference different floating-point condition codes. For example:

Pipeline Example 22-35 These two instructions cannot be grouped

FCMP	fcc0, f2, f4	G	E	C	N_1	N_2	N_3	W	
FBfcc	fcc1, target		G	E	C	N_1	N_2	N_3	W

It is possible, however, for an FCMP{E}{s,d} to be grouped with an older **FBfcc** in the same group. For example:

Pipeline Example 22-36 FCMP{E}{s,d} can be grouped with an older **FBfcc**

FBfcc	G	E	C	N_1	N_2	N_3	W
FCMP	G	E	C	N_1	N_2	N_3	W

An **FMOVcc** that references the same condition code set by a FCMP{E}{s,d} cannot be in the same or the following group. For example:

Pipeline Example 22-37 Grouping for **FMOVcc** that references the same condition code set by a FCMP{E}{s,d}

FCMP	fcc0, f2, f4	G	E	C	N_1	N_2	N_3	W		
FMOVcc	fcc0, f6, f8			G	E	C	N_1	N_2	N_3	W

FMOVcc cannot be in the same group as FCMP{E}{s,d}, because they are both A-Class floating-point instructions.

MOVcc based on a floating-point condition code can be in the same group as an FCMP{E}{s,d}, however, if they reference different condition codes. For example:

Pipeline Example 22-38 MOVcc can be grouped with an FCMP{E}{s,d} if FP condition codes are different

FCMP	fcc0, f2, f4	G	E	C	N_1	N_2	N_3	W
MOVcc	fcc1, f6, f8	G	E	C	N_1	N_2	N_3	W

Latencies between dependent floating-point and graphics instructions are shown in *Table 22-2* on page 365. Latencies depend on the instruction generating the result (use the left column of the table to select a row) and the operation using the result (use the top row of the table to select a column). For example, *Pipeline Example 22-39*:

Pipeline Example 22-39 Groupings also depend upon latency of the instruction producing a result for a subsequent operation

FADDs	f2, f3, f0	G	E	C	N_1	N_2	N_3	W		
FMULs	f6, f1, f2				G	E	C	N_1	N_2	N_3
FADDs	f2, f3, f0	G	E	C	N_1	N_2	N_3	W		
FMOVs	f6,f1,f2					G	E	C	N_1	N_2

FDIV{s,d}, FSQRT{s,d}, block load, block store, ST{X}FSR, and LD{X}FSR instructions wait in the G-stage for the remaining latency of the previous divide or square root, even if there is no data dependency. An FGA or FGM instruction (see *Table 22-2*) that first enters the G-stage one cycle before an **FDIV** or **FSQRT** dependent instruction would be released will be held for one clock, regardless of data dependency.

FDIV and **FSQRT** use the floating-point multiplier for final rounding, so an M-Class operation cannot be dispatched in the third clock before the divide is finished. A load use stall that occurs in the third or fourth clock before normal divide completion will delay completion by a corresponding amount.

FDIV and **FSQRT** stall earlier instructions with the same *rd* (including floating-point loads) for the same time as a source register dependency.

Graphics instructions, FdTOi, FxTOs, FdTOs, FDIVs, and FSQRTs lock the double-precision register containing the single-precision result for data dependency checking. For example:

Pipeline Example 22-40 Group separation because of dependency checking of prior result

FORs	f2, f4, f0	G	E	C	N_1	N_2	N_3	W	
FANDs	f1, f1, f1		G	E	C	N_1	N_2	N_3	W

Floating-point stores other than ST{X}FSR can store the result of a floating-point or graphics instruction other than **FDIV** or **FSQRT** and be in the same group. For example:

Pipeline Example 22-41 Most FP stores can be in the same group

FADDs	f2, f5, f6	G	E	C	N_1	N_2	N_3	W
STF	f6, [address]	G	E	C	N_1	N_2	N_3	W

Floating-point stores of the result of an **FDIV** or **FSQRT** are treated the same as a dependent floating-point instruction.

ST(X)FSR cannot be dispatched in the two groups following a floating-point or graphics instruction that references the floating-point registers. For example:

Pipeline Example 22-42 ST(X)FSR cannot be in two groups following a reference to the FP registers

FMULd	G	E	C	N_1	N_2	N_3	W		
STFSR				G	E	C	N_1	N_2	N_3

To simplify critical timing paths, floating-point operations are usually stalled in the G-stage until earlier floating-point operations with a different precision complete, regardless of data dependency. This behavior is described more precisely in the following two rules. Floating-point loads and stores are independent of these mixed precision rules.

A floating-point or graphics instruction that follows an **FMOV**, **FABS**, **FNEG** of different precision breaks the group, even if there is no data dependency. For example:

Pipeline Example 22-43 Group separation for instructions following FMOV, FABS, FNEG, of differing precision

FMOVs	G	E	C	N_1	N_2	N_3	W	
FMULd		G	E	C	N_1	N_2	N_3	W

A floating-point or graphics instruction following an operation other than **FMOV**, **FABS**, **FNEG**, **FDIV**, **FSQRT** of different precision is stalled until the N_2 Stage of the earlier operation, even if there is no data dependency. For example:

Pipeline Example 22-44 Stall for instructions following other instructions of differing precision

FADDs	f2, f5, f0	G	E	C	N_1	N_2	N_3	W			
FMULd	f2, f2, f2					G	E	C	N_1	N_2	

As an exception to the previous rule, **FDIV** or **FSQRT** can be grouped with an older operation of different precision, but are stalled until the N_2 Stage of the earlier operation otherwise.

For the preceding two rules, all graphics instructions, FDIVs, FSQRTs, FdTOi, FsTOx, FiTOd, FxTOs, FsTOd, FdTOs, and FsMULd are considered to be double, even though a single-precision register is referenced. For example, the following instructions can be grouped together:

Pipeline Example 22-45 Instructions grouped because graphics instruction is considered as double

FORs	f2, f4, f0	G	E	C	N_1	N_2	N_3	W
FANDs	f2, f2, f2	G	E	C	N_1	N_2	N_3	W

22.8.2 Floating-Point and Graphics Instruction Latencies

Table 22-2 on page 365 documents the latencies for floating-point and graphics instructions. For table entries containing two numbers, premature dispatching occurs when the destination and source precision are different, but both are treated as double because of a graphics or mixed-precision floating-point instruction. To avoid the pipe flush overhead, software should explicitly force the use instruction to be at least the latency number of groups after the source instruction. Mixed precision bypassing is unlikely to occur with floating-point data. Software scheduling is only needed for initializing the PDIST *rd* register and for graphics instructions single results used as part of a double-precision graphics source operand, or vice versa.

The table uses the following abbreviations:

Table 22-1 Abbreviations Used in *Table 22-2*

Abbrev.	Meaning
FGA	Graphics A-Class instruction
FGM	Graphics M-Class instruction
FPA	Floating-point A-Class instruction
FPM	Floating-point M-Class instruction

Table 22-2 Latencies for Floating-Point and Graphics Instructions

Result used by →		**FPA or FPM**	**FGA**	**FGM**	
Result generated by: ↓		FADD{s,d} FSUB{s,d} F{s,d}TO{i,x} F{i,x}TO{d,s} F{s,d}TO{d,s} FCMP{s,d} FCMPE{s,d} FMUL{s,d} FsMULd FDIV{s,d} FSQRT{s,d}	FMOVr{s,d} FMOVcc{s,d} FMOV{s,d} FABS{s,d} FNEG{s,d} FPADD{16,32}{s} FPSUB{16,32}{s} FALIGNDATA FPMERGE FEXPAND	FPACK{16,32,FIX} FMUL8x16{AL,A U} FMUL{d}8ULx16 FMUL{d}8SUx16 PDIST{rs1, rs2} FCMPLE{16,32} FCMPNE{16,32} FCMPGT{16,32} FCMPEQ{16,32}	PDIST {rd}
FPA or FPM	FADD{s,d} FSUB{s,d} F{s,d}TO{i,x} F{i,x}TO{d,s} F{s,d}TO{d,s} FMUL{s,d} FsMULd	3[4][1]	4	4	[2][1]
	FDIVs, FSQRTs	12[13][1]	13	13	13
	FDIVd, FSQRTd	22[23][1]	23	23	23
FGA	FMOV{s,d} FABS{s,d} FNEG{s,d}	1	1	1	[2][1]
	FMOVr{s,d} FMOVcc{s,d}	2	2	2	[2][1]
	FPADD{16,32}{s} FPSUB{16,32}{s} FALIGNDATA FPMERGE FEXPAND	2	1	1[2][1]	[2][1]
FGM	FPACK{16,32,FIX}	4	3	1[4][1]	[2][1]
	FMUL8x16{AL,A U} FMUL{d}8ULx16 FMUL{d}8SUx16 PDIST	4	3	3[4][1]	1

1. Latency numbers enclosed in square brackets ([ ]) indicate cases where the hardware may prematurely dispatch a dependent instruction from the G-stage, cancel it in the W Stage, and then refetch it. This effectively inserts nine bubbles into the pipe.

APPENDIX **A**

Debug and Diagnostics Support

A.1 Overview

All debug and diagnostics accesses are double-word aligned, 64-bit accesses. Non-aligned accesses cause a *mem_address_not_aligned* trap. Accesses must use LDXA/STXA/LDFA/STDFA instructions, except for the instruction cache ASIs which must use LDDA/STDA/STDFA. Using another type of load or store causes a *data_access_exception* trap (with SFSR.FT=8, Illegal ASI size). An Attempt to access these registers in non-privileged mode causes a *data_access_exception* trap (with SFSR.FT=1, privilege violation). User accesses can be made through system calls to these facilities. See Section 15.9.4, *I-/D-MMU Synchronous Fault Status Registers (SFSR)* on page 216 for SFSR details.

Caution – A STXA to any internal debug or diagnostic register requires a MEMBAR `#Sync` before another load instruction is executed. The MEMBAR `#Sync` must also be done on or before the delay slot of a delayed control transfer instruction of any type. This condition is not only to guarantee that the result of the STXA is seen; the STXA may corrupt the load data if there is not an intervening MEMBAR `#Sync`.

A.2 Diagnostics Control and Accesses

The UltraSPARC IIi diagnostics control and data registers are accessed through RDASR/WRASR or through load/store alternate instructions.

A.3 Dispatch Control Register

ASR 0x12: The Dispatch Control Register, ASR 0x12, enables performance features related to instruction dispatch, and also controls the output of internal signals to UltraSPARC IIi SYSADR[14:0] pins to help in chip debug and instrumentation.

For a more detailed description, see Section I.1.2, *Dispatch Control Register* on page 434.

A.4 Floating-Point Control

Two state bits (PSTATE.PEF and FPRS.FEF) in the SPARC-V9 architecture provide the means to disable direct floating-point execution. If either field is cleared, an *fp_disabled* trap is taken when a floating-point instruction is encountered.

Note – Graphics instructions that use the floating-point register file and instructions that read or update the Graphic Status Register (GSR) are treated as floating-point instructions. They cause an *fp_disabled* trap if either PSTATE.PEF or FPRS.FEF is cleared. See Section 13.4, *Graphics Instructions* on page 134 for more information.

A.5 Watchpoint Support

UltraSPARC IIi implements "break before" *watchpoint* traps; instruction execution is stopped immediately before the watchpoint memory location is accessed. *TABLE A-1* on page 369 lists ASIs that are affected by the two watchpoint traps. For 128-bit atomic load and 64-byte block load and store, a *watchpoint* trap is generated only if the watchpoint overlaps the lowest addressed 8 bytes of the access.

Note – In order to avoid trapping indefinitely, software should emulate the instruction at the watched address and execute a DONE instruction or turn off the watchpoint before exiting a *watchpoint* trap handler.

TABLE A-1 ASIs Affected by Watchpoint Traps

ASI Type	ASI Range	D-MMU	Watchpoint if Matching VA	Watchpoint if Matching PA
Translating ASIs	$04_{16}..11_{16}$, $18_{16}..19_{16}$, $24_{16}..2C_{16}$, $70_{16}..71_{16}$, $78_{16}..79_{16}$, $80_{16}..FF_{16}$	On Off	Y N	Y Y
Bypass ASIs	$14_{16}..15_{16}$, $1C_{16}..1D_{16}$	—	N	Y
Nontranslating ASIs	$45_{16}..6F_{16}$, $76_{16}..77_{16}$, $7E_{16}..7F_{16}$	—	N	N

A.5.1 Instruction Breakpoint

There is no hardware support for instruction breakpoint in UltraSPARC IIi. The TA (Trap Always) instruction can be used to set program breakpoints.

A.5.2 Data Watchpoint

Two 64-bit data watchpoint registers provide the means to monitor data accesses during program execution. When virtual/physical data watchpoint is enabled, the virtual/physical addresses of all data references are compared against the content of the corresponding watchpoint register. If a match occurs, a *VA_/PA_watchpoint* trap is signalled before the data reference instruction is completed. The virtual address watchpoint trap has higher priority than the physical address watchpoint trap.

Separate 8-bit byte masks allow watchpoints to be set for a range of addresses. Zero bits in the byte mask causes the comparison to ignore the corresponding bytes in the address. These watchpoint byte masks and the watchpoint enable bits reside in the LSU_Control_Register. See Section A.6, *LSU_Control_Register* on page 370 for a complete description.

A.5.3 Virtual Address (VA) Data Watchpoint Register

63 … 44	43 … 3	2 … 0
	DB_VA	—

Figure A-1 VA Data Watchpoint Register Format (ASI 58_{16}, VA=38_{16})

DB_VA: The 64-bit virtual data watchpoint address

Note – UltraSPARC-I and UltraSPARC-II support a 44-bit virtual address space. Software must write a sign-extended 64-bit address into the VA watchpoint register. The watchpoint address is sign-extended to 64 bits from bit 43 when read.

A.5.4 Physical Address Data Watchpoint Register

63 … 41	40 … 3	2 … 0
	DB_PA	—

Figure A-2 PA Data Watchpoint Register Format (ASI 58_{16}, VA=40_{16})

DB_PA: The 41-bit physical data watchpoint address

Note – UltraSPARC-I and UltraSPARC-II support a 41-bit physical address space. Software must write a zero-extended 64-bit address into the watch point register.

A.6 LSU_Control_Register

ASI 45_{16}, VA=00_{16}

Name: ASI_LSU_CONTROL_REGISTER

The LSU_Control_Register contains fields that control several memory-related hardware functions in UltraSPARC IIi. These include I-cache, D-cache, MMUs, bad parity generation, and watchpoint setting. See also *Table 17-5* on page 261 for the state of this register after reset or RED_state trap.

—	—	—	—	PM	VM	PR	PW	VR	VW	—	FM	DM	IM	DC	IC
63 44	43	42	41	40 33	32 25	24	23	22	21	20	19 4	3	2	1	0

Figure A-3 LSU_Control_Register Access Data Format (ASI 45_{16})

A.6.1 Cache Control

IC:L SU.I-cache_enable; if cleared, misses are forced on I-cache accesses with no cache fill.

DC:L SU.D-cache_enable; if cleared, misses are forced on D-cache accesses with no cache fill. A FLUSH, DONE, or RETRY instruction is needed after software changes this bit to ensure the new information is used.

A.6.2 MMU Control

IM: LSU.enable_I-MMU; if cleared, the I-MMU is disabled (pass-through mode).

DM: LSU.enable_D-MMU; if cleared, the D-MMU is disabled (pass-through mode).

Note – When the MMU/TLB is disabled, a VA is passed through to a PA. Accesses are assumed to be non-cacheable with side-effects.

A.6.3 Parity Control

FM<7:0>: LSU.parity_mask; if set, UltraSPARC IIi writes generate incorrect parity on the E-cache data bus for bytes corresponding to this mask. The parity_mask corresponds to the eight bytes of the E-cache data bus.

Note – The parity mask is endian-neutral.

TABLE A-2 LSU Control Register: Parity Mask Examples

Parity Mask	Addr of Bytes Affected	
	7654	3210
00_{16}	0000	0000
01_{16}	0000	0000
22_{16}	0010	0010
FF_{16}	1111	1111

A.6.4 Watchpoint Control

Watchpoint control is further discussed in Section A.5, *Watchpoint Support* on page 368.

A.6.4.1 Virtual Address Data Watchpoint Enable

VR, **VW**: LSU.virtual_address_data_watchpoint_enable; if VR/VW is set, a data read/write that matches the (range of) addresses in the virtual watchpoint register causes a watchpoint trap. Both VR and VW may be set to place a watchpoint for either a read or write access.

A.6.4.2 Virtual Address Data Watchpoint Byte Mask

VM<7:0> LSU.virtual_address_data_watchpoint_mask; the virtual_address_data_watch_point_register contains the virtual address of a 64-bit word to be watched. The 8-bit virtual_address_data_watch_point_mask controls which bytes within the 64-bit word should be watched. If all eight bits are cleared, the virtual watchpoint is disabled. If watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, a virtual watchpoint trap is generated.

TABLE A-3 LSU Control Register: VA/PA Data Watchpoint Byte Mask Examples

Watchpoint Mask	Addr of Bytes Watched	
	7654	**3210**
00_{16}	**Watchpoint disabled**	

TABLE A-3 LSU Control Register: VA/PA Data Watchpoint Byte Mask Examples

Watchpoint Mask	Addr of Bytes Watched	
	7654	3210
01_{16}	0000	0001
32_{16}	0011	0010
FF_{16}	1111	1111

A.6.4.3 Physical Address Data Watchpoint Enable

PR, **PW**: LSU.physical_address_data_watchpoint_enable; if PR/PW is set, a data read/write that matches the (range of) addresses in the physical watchpoint register causes a watchpoint trap. Both PR and PW may be set to place a watchpoint on either a read or write access.

A.6.4.4 Physical Address Data Watchpoint Byte Mask

PM<7:0>: LSU.physical_address_data_watchpoint_mask; the physical_address_data_watch_point_register contains the physical address of a 64-bit word to be watched. The 8-bit physical_address_data_watch_point_mask controls which bytes within the 64-bit word should be watched. If all eight bits are cleared, the physical watchpoint is disabled. If the watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, a physical watchpoint trap is generated.

A.7 I-cache Diagnostic Accesses

The instruction cache (I-cache) utilizes the Dynamic Set Prediction technique to realize a set-associative cache with a direct-mapped physical RAM design. The direct-mapped RAM core is logically divided into two sets. Rather than using the tag to determine which set contains the requested instructions, a set prediction from the last access to the I-cache is used to access the instructions for the current fetch.

Figure A-4 Simplified I-cache Organization (Only 1 Set Shown)

Each set of the I-cache is divided into four fields per entry:

- The instruction field contains eight 32-bit instructions.
- The tag field contains a 28-bit physical tag and a valid bit.
- The pre-decode field contains eight 4-bit information packets about the instructions stored.
- The next field contains the LRU bit, next address, branch and set predictions. There is one physical LRU bit per I-cache line (that is, 16 instructions) but it is logically replicated for each set. There are four 2-bit dynamic branch prediction (BRPD) fields, one for each two adjacent instructions. Two sets of set prediction and next address fields, one for each four instructions.

Note – To simplify the implementation, read access to the instruction cache fields (ASIs 60_{16}..$6F_{16}$) must use the LDDA instruction instead of LDXA or LDDFA. Using another type of load causes a *data_access_exception* trap (with SFSR.FT=8, Illegal ASI size). LDDA updates two registers. The useful data is in the odd register, the contents of the even register are undefined.

A.7.1 I-cache Instruction Fields

ASI 66_{16}, VA<63:14>=0, VA<13>=IC_set, VA<12:3>=IC_addr, VA<2:0>=0

Name: ASI_ICACHE_INSTR

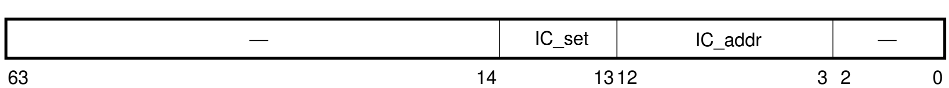


Figure A-5 I-cache Instruction Access Address Format (ASI 66_{16})

IC_set: This 1-bit field selects a set (2-way associative).

IC_addr: This 10-bit index <12:3> selects an aligned pair of 32-bit instructions.

IC_instr 0		IC_instr 1	
63	33	32	0

Figure A-6 I-cache Instruction Access Data Format (ASI 66_{16})

IC_instr: two 32-bit instruction fields

A.7.2 I-cache Tag/Valid Fields

ASI 67_{16}, VA<63:14>=0, VA<13>=IC_set, VA<12:5>=IC_addr, VA<4:0>=0

Name: ASI_ICACHE_TAG

—		IC_set	IC_addr		—	
63	14	13	12	5	4	0

Figure A-7 I-cache Tag/Valid Access Address Format (ASI 67_{16})

IC_set: This 1-bit field selects a set (2-way associative).

IC_addr: This 8-bit index (VA<12:5>) selects a cache tag.

Undefined		IC_valid	IC_tag		*Undefined*	
63	37	36	35	8	7	0

Figure A-8 I-cache Tag/Valid Field Data Format (ASI 67_{16})

Undefined: The values of these bits are undefined on reads and must be masked off by software.

IC_valid: The 1-bit valid field

IC_tag: The 28-bit physical tag field (PA<40:13> of the associated instructions)

A.7.3 I-cache Predecode Field

ASI $6E_{16}$, VA<63:14>=0, VA<13>=IC_set, VA<12:5>=IC_addr, VA<4:3>=IC_line, VA<2:0>=0

Name: ASI_ICACHE_PRE_DECODE

—	IC_set	IC_addr	IC_line	—
63 14	13	12 5	4 3	2 0

Figure A-9 I-cache Predecode Field Access Address Format (ASI $6E_{16}$)

IC_set: This 1-bit field selects a set (2-ways).

IC_addr: This 8-bit index (i.e. addr <12:5>) selects an IC_Line.

IC_line: For LDDA accesses, this 2-bit field selects a pair of pre-decode fields in a 64-bit-aligned instruction pair. For STXA accesses, the least significant bit is ignored. The most significant bit selects four pre-decode fields in a 128-bit-aligned instruction quad.

Undefined	IC_pdec 0	IC_pdec 1
63 8	7 4	3 0

Figure A-10 I-cache Predecode Field LDDA Access Data Format (ASI $6E_{16}$)

Undefined	IC_pdec 0	IC_pdec 1	IC_pdec 2	IC_pdec 3
63 16	15 12	11 8	7 4	3 0

Figure A-11 I-cache Predecode Field STXA Access Data Format (ASI $6E_{16}$)

Undefined: The value of these bits are undefined on reads and must be masked off by software.

IC_pdec: The two 4-bit pre-decode fields. The encodings are:

- Bits<3:2> = 00 CALL, BPA, FBA, FBPA or BA
- Bits<3:2> = 01 Not a CALL, JMPL, BPA, FBA, FBPA or BA
- Bits<3:2> = 10 Normal JMPL (do not use return stack)
- Bits<3:2> = 11 Return JMPL (use return stack)
- Bit<1>If clear, indicates a PC-relative CTI.
- Bit<0>If set, indicates a STORE.

Note – The predecode bits are not updated when instructions are loaded into the cache with ASI_ICACHE_INSTR. They are only accurate for instructions loaded by instruction cache miss processing.

A.7.4 I-cache LRU/BRPD/SP/NFA Fields

ASI $6F_{16}$, VA<63:14>=0, VA<13>=IC_set, VA<12:3>=IC_addr, VA<2:0>=0

Name: ASI_ICACHE_PRE_NEXT_FIELD

—	IC_set	IC_addr	IC_line	—
63 14	13	12 5	4	3 0

Figure A-12 I-cache LRU/BRPD/SP/NFA Field Access Address Format (ASI $6F_{16}$)

Stores to ASI_ICACHE_PRE_NEXT_FIELD are undefined unless the instruction cache is disabled via the IC bit of the LSU control register (see *LSU_Control_Register* on page 370).

IC_set: This 1-bit field selects a set (2-way associative).

IC_addr: This 8-bit index (addr <12:5>) selects an IC_Line.

IC_line: This 1-bit field selects two BRPD and one NFA fields for four 128-bit aligned instructions.

Undefined	IC_lru	IC_sp	IC_nfa	IC_brpd 0	IC_brpd 1	und.
63 25	24	23	22 12	11 10	9 8	7 0

Figure A-13 I-cache LRU/BRPD/SP/NFA Field LDDA Access Data Format (ASI $6F_{16}$)

Undefined, **und**: The value of these bits are undefined on reads and must be masked by software.

IC_lru: selects the least recently accessed set of the line corresponding to IC_addr. There is only one physical LRU bit per IC_addr value (i.e. cache line). The IC_lru field can be read for each value of IC_set and IC_line, but can only be written when IC_set is zero.

Note – The LRU bit is not updated when instructions are accessed with ASI_ICACHE_INSTR.

IC_brpd<1:0>: Two 2-bit dynamic branch prediction fields. The encodings are

- IC_brpd<1>If set, strong prediction
- IC_brpd<0>If set, taken prediction

During I-cache miss processing, IC_brpd is initialized to likely-taken if either of the corresponding instructions is a branch with static prediction bit set; otherwise, IC_brpd is set to likely-not-taken. The prediction bits are subsequently updated according to the dynamic branch history of the corresponding instructions, as shown in *Figure A-14*. (**Note**: This figure is identical to *Figure 21-6*.)

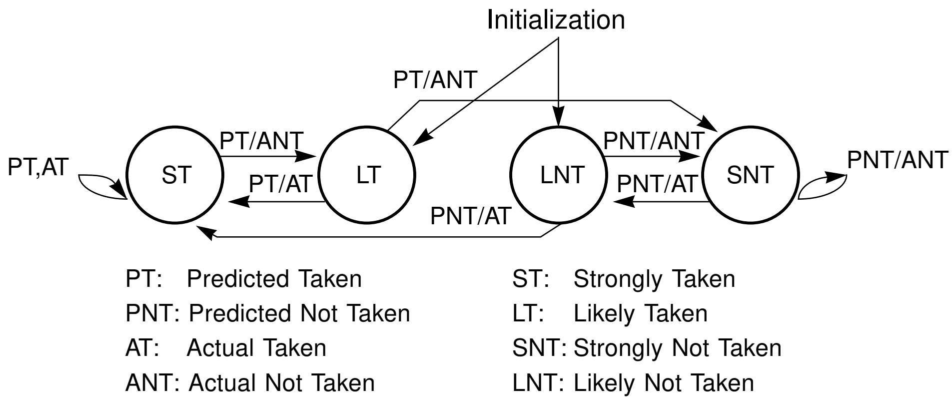


Figure A-14 Dynamic Branch Prediction State Diagram

IC_sp 1-bit Set-Prediction (SP) field; selects the next set from which to fetch

IC_nfa1 1-bit Next-Field-Address field (NFA<10:0> = VA<13:3>); selects the next line from which to fetch and the instruction offset within that line

---

Note – The branch prediction, set prediction and next field address fields are not updated when instructions are loaded into the cache with ASI_ICACHE_INSTR.

When a cache line is brought into the I-cache, the corresponding IC_sp fields are initialized to the same set as the currently missed line. The corresponding IC_nfa fields are initialized to the next sequential sub-block.

A.8 D-cache Diagnostic Accesses

Two D-cache ASI accesses are supported: data (ASI 46_{16}) and tag/valid (ASI 47_{16}).

A.8.1 D-cache Data Field

ASI 46_{16}, VA<63:14>=0, VA<13:3>=DC_addr, VA<2:0>=0

Name: ASI_DCACHE_DATA

—	DC_addr	—
63 … 14	13 … 3	2 … 0

Figure A-15 D-cache Data Access Address Format (ASI 46_{16})

DC_addr: This 11-bit index <13:3> selects a 64-bit data field (16KB).

DC_data
63 … 0

Figure A-16 D-cache Data Access Data Format (ASI 46_{16})

DC_data: 64-bit data

A.8.2 D-cache Tag/Valid Fields

ASI 47_{16}, VA<63:14>=0, VA<13:5>=DC_addr, VA<4:0>=0

Name: ASI_DCACHE_TAG

—	DC_addr	—
63 … 14	13 … 5	4 … 0

Figure A-17 D-cache Tag/Valid Access Address Format (ASI 47_{16})

DC_addr: This 9-bit index <13:5> selects a tag/valid field (512 tags).

—	DC_tag	DC_valid
63 … 30	29 … 2	1 … 0

Figure A-18 D-cache Tag/Valid Access Data Format (ASI 47_{16})

DC_tag: The 28-bit physical tag (PA<40:13> of the associated data).

DC_valid: The 2-bit valid field, one for each sub-block (32b block, 16b sub-block). Bit<1> corresponds to the highest addressed 16 bytes, bit<0> to the lowest addressed 16 bytes.

A.9 E-cache Diagnostics Accesses

Compatibility Note – Because of the smaller external cache data and tag, some adjustments are made to these diagnostic accesses.

Separate ASIs are provided for reading (0x7E) and writing (0x76) the E-cache tags and data.

Note – During E-cache diagnostics accesses, the VA is passed through to the PA without page mapping. To avoid undesired modifications of the E-cache state, Take care when using ldxa/stxa instructions with these ASIs to prevent cacheable instruction prefetch PA<17:6> that matches the PA<17:6> of the E-cache diagnostic access. It is permissible, however, for the E-cache state to change; there is no hardware conflict involved.

Caution – Using ASI 0x76/77/7E/7F with VA[40:39]==00 and a VA[15:0] matching any of the PA[15:0] listed for the CSR addresses in noncacheable space, other than 0x00, 0x18, 0x20, 0x38, 0x40, 0x50, 0x60, or 0x70, can cause a load to return data, and a store to modify, the corresponding CSR. The list of addresses is in Section 19.4.3, *DMA Error Registers* on page 316. These ASIs are protected by privilege bit/trap so as not to provide an unprotected back-door access.

A.9.1 E-cache Data Fields

- ASI 0x76 (WRITING) or 0x7E (READING), VA<63:41>==0, VA<40:39>==1,
- VA<38:18>==0, VA<17:3>==EC_addr, VA<2:0>==0 (0.25MB)
- VA<38:19>==0, VA<18:3>==EC_addr, VA<2:0>==0 (0.5MB)
- VA<38:20>==0, VA<19:3>==EC_addr, VA<2:0>==0 (1 MB)
- VA<38:21>==0, VA<20:3>==EC_addr, VA<2:0>==0 (2 MB)

Name: ASI_ECACHE_W (0x76), ASI_ECACHE_R (0x7E

—	01	—	EC_addr	—
63 41	40 39	38 21	20 3	2 0

Figure A-19 E-cache Data Access Address Format

EC_addr: A 15-bit index <17:3> selects a 64-bit data field from a 0.25 MB E-cache. A 16-bit index <18:3> selects a 64-bit data field from a 0.5 MB E-cache. A 17-bit index <19:3> selects a 64-bit data field from a 1 MB E-cache. An 18-bit index <20:3> selects a 64-bit data field from a 2 MB E-cache.

EC_data
63 0

Figure A-20 E-cache Data Access Data Format

EC_data: 64-bit data

A.9.2 E-cache Tag/State/Parity Field Diagnostic Accesses

- ASI 0x76 (WRITING) or 0x7E (READING), VA<63:41>==0, VA<40:39>==2,
- VA<38:18>==0, VA<17:6>==EC_addr, VA<5:0>==0 (0.25MB)
- VA<38:19>==0, VA<18:6>==EC_addr, VA<5:0>==0 (0.5MB)
- VA<38:20>==0, VA<19:6>==EC_addr, VA<5:0>==0 (1 MB)
- VA<38:21>==0, VA<20:6>==EC_addr, VA<5:0>==0 (2 MB)
- Name: ASI_ECACHE_W (0x76), ASI_ECACHE_R (0x7E)

—	10	—	EC_addr	—
63 41	40 39	38 22	21 6	5 0

Figure A-21 E-cache Tag Access Address Format

If read, the contents of the E-cache tag/state/parity fields in the selected E-cache line are stored in the E-cache_tag_data_register. This register can be read by an LDA with ASI_ECACHE_TAG_DATA; its contents are written to the destination register.

If written, the content of the E-cache_tag_data_register is written to the selected E-cache tag/state/parity fields. The content of the **E-cache_tag_data_register** are previously updated with STA at ASI_ECACHE_TAG_DATA.

Note – Software must ensure that the two-step operations are done atomically; e.g., LDXA ASI_ECACHE (TAG) and LDXA ASI_ECACHE_TAG_DATA, STXA ASI_ECACHE_TAG_DATA and STXA ASI_ECACHE (TAG).

Note – The destination register of a LDXA ASI_ECACHE (TAG) is undefined. It is recommended to use `%g0` as the destination for this ASI access. Similarly, the contents of the destination register in STXA ASI_ECACHE (TAG) is ignored, but the contents of the E-cache_tag_data_register are written to the selected E-cache line.

A.9.3 E-cache Tag/State/Parity Data Accesses

ASI 0x4E, VA<63:0>==0

Name: ASI_ECACHE_TAG_DATA

—	EC_parity	EC_state	00	EC_tag
63 29	17 16	15 14	13 12	11 0

Figure A-22 E-cache Tag Access Data Format

EC_tag:14-bit physical tag field

- EC_tag<13:0>==00, PA<29:18> of associated data. Note EC_tag<13:12> always read as 0's. (The actual SRAM contents are returned, but UltraSPARC IIi always forces 0's on all tag writes)

EC_state: 2-bit E-cache state field. Encodings are

- EC_state<1:0> == 00 Invalid
- EC_state<1:0> == 01 Not Used
- EC_state<1:0> == 10 Exclusive
- EC_state<1:0> == 11 Modified

EC_parity: 2-bit E-cache tag (odd) parity field

- EC_parity<1>Parity of EC_state<1:0], EC_tag<13:8>

Tag parity on normal operation is computed using the actual PA<31:30>. If that PA<31:30> ==01 or 10 (greater than the supported DRAM) a tag parity error is created.

- EC_parity<0>Parity of EC_tag<7:0>

A.10 Memory Probing and Initialization

The Memory Controller in the UltraSPARC IIi processor is changed between the SME1040 and SME1430 CPUs. See Chapter 18, *MCU Control and Status Registers* for detailed information.

A.10.1 Initialization

The following steps must be performed before *any* access can be made to memory.

1. Determine the operating frequency of the system, then initialize the Mem_Control1 register with the appropriate values for the given operating frequency. See Section 18.3, *Mem_Control1 Register* on page 270.
2. Enable refresh by setting the RefEnable bit in the Mem_Control0 register. See Section 18.2, *Mem_Control0 Register* on page 267. This action supplies the DRAMs with their required minimum of eight RAS cycles to initialize their internal circuitry before they can be accessed. Refresh is turned on by setting the RefEnable bit in the Mem_Control0 register. ()RefInterval should be set to a value assuming a full memory system (see RefInterval table). Also, the DIMMPairPresent bits should all be set to 1.

After the probing step, RefInterval and DIMMPairPresent can be set to the proper values (must first turn off RefEnable). After setting the RefEnable, wait at least

```
(8 DIMMs)*(8 refreshes)*(RefInterval)*(32 clocks)*(clock period) seconds
```

before beginning the probing step.

A.10.2 Memory Probing

The only way to determine the number and size of DIMMs in the system is by probing. That is, writing to certain memory locations, and reading back to determine the effects of those writes.

This section describes an algorithm for DIMM probing that is based upon the behavior of the hardware and the supported DIMM configurations. The algorithm employs the fact that writes to non-existent addresses can "wrap around" and overwrite data in a valid location (assuming that a DIMM is present). The algorithm described in the following sections specifies these addresses. The data pattern that is written to each location should contain a unique bit-signature, rather than consisting of all 0's or all 1's.

All addresses for block write/read within a DIMM slot are specified below as PA[26:0]. PA[29:27] are varied for probing different DIMM slots/banks.

Perform the two steps below for PA[29:27] = 000, 001, 010, 011, in 10-bit column address mode. This covers a single bank in all four DIMM-pair slots/banks.

A.10.3 Detection of DIMM presence

To check whether a DIMM-pair is present or not, perform a write to a block of memory beginning at 0x000_0000, then read back from this location. If incorrect data is returned and/or an ECC error is generated, then there is no DIMM-pair at this location. Skip to the next DIMM-pair.

The data pattern written to each location should contain a unique bit-signature, rather than consisting of all 0s or all 1s.

A.10.4 Determination of DIMM pair Size

To determine the base size of the existing DIMMs, write to 0x100_0000, then read from 0x000_0000. If the read does not return the data initially written to 0x000_0000, DIMM size is 8 MB. This is because an 8 MB DIMM only has 24 address bits and the write to 0x100_0000 wrapped to overwrite the contents of 0x000_0000.

Perform a write to 0x200_0000, then read from 0x000_0000. If the read does not return the data written to 0x000_0000, the DIMM is of 16-MB capacity. This is because 16 MB DIMM only has 25 valid address bits, so the write to 0x200_0000 wrapped and overwrote the contents of 0x000_0000.

If the correct data is returned, write to 0x400_0000 and read back from 0x000_0000. If the read does not return the data originally written into 0x000_0000, this indicates a 32 MB DIMM. The 32 MB DIMM has 26 valid address bits so the write to 0x400_0000 wrapped and overwrote the contents of 0x000_0000.

If the correct data is returned in 10-bit column address mode, this indicates a 64 MB DIMM—The largest possible using 10-bit column address mode.

If in 11-bit column address mode, and the correct data is returned, write to 0x800_0000. Read back from 0x000_0000. If the read fails to return the data originally written into 0x000_0000, this indicates a 64 MB DIMM. A 64 M-byte DIMM has 27 bits of valid address, so the write to 0x800_0000 wrapped around and overwrote the contents of 0x000_0000.

Return of correct data indicates a 128 MB DIMM—the largest possible in 11-bit column address mode.

Repeat with PA[29]==1 to check for a second bank on each DIMM.

A.10.5 Determination of DIMM pair size equivalence

For each DIMM pair, the above process should be repeated with PA[4]==1. The size of the other DIMM in the pair should be the same. If not, the smaller result must be used.

A.10.6 11-bit Column Address Mode

The DIMMs may have 11-bit column addresses, in which case they may be twice as large as previously indicated. 11-bit column addresses are supported with a mode bit in the Mem_Control0 CSR. It should only be enabled if all DIMMs have 11-bit column addresses.

Only DIMM pairs 0 and 2 are used in 11-bit column address mode.

After determining which DIMMs are present, the boot PROM should determine if DIMM pairs 0 and 2 have 11-bit column addresses, and, if so, enable that mode.

Since column address bit [10] is always PA[14], 11-bit column addresses can be detected by the same algorithm used above to detect DIMM presence,. Instead of toggling high order PA bits, PA[14] is toggled while all other bits are kept constant (the PA to use depends on the DIMM pair being tested).

If toggling PA[14] causes overwrite while the 11-bit column address mode is enabled, then the DRAMs in that DIMM should be assumed to be 10-bit column address DIMMs, and the mode not used.

Ideally, the PA[14] test should be used on every DIMM (2 in each pair) by toggling PA[4] also, to guarantee that matching DIMMs have been inserted before 11-bit column address mode is allowed.

If enabled, the sizes of DIMM pair 0 and 2 are doubled if they exist, and pair 1 and 3 should be ignored because they should not exist.

A.10.7 Banked DIMMs

The probing algorithm should also toggle PA[29] to determine if banked DIMMs are present, as above.

A.10.8 Completion of probing

Write RefInterval and DIMMPairPresent with the appropriate values after the probing is finished. After the probing step is performed, then the physical memory space available in the machine is known. The boot processor can then initialize data and ECC in the entire memory space with known values using block writes. After this step is performed, the memory system is ready for operation.

APPENDIX **B**

Performance Instrumentation

B.1 Overview

Two performance events can be measured simultaneously in UltraSPARC IIi. The Performance Control Register (PCR) controls event selection and filtering (that is, counting user and/or system level events) for a pair of 32-bit Performance Instrumentation Counters (PICs).

B.2 Performance Control and Counters

The 64-bit PCR and PIC are accessed through read/write Ancillary State Register instructions (RDASR/WRASR). PCR and PIC are located at ASRs 16 (10_{16}) and 17 (11_{16}) respectively. Access to the PCR is privileged. Non privileged accesses cause a *privileged_opcode* trap. Non-privileged access to PICs may be restricted by setting the PCR.PRIV field while in privileged mode. When PCR.PRIV=1, an attempt by non-privileged software to access the PICs causes a *privileged_action* trap. Event measurements in non-privileged and/or privileged modes can be controlled by setting the PCR.UT and PCR.ST fields.

Two 32-bit PICs each accumulate over 4 billion events before wrapping around. There is no special handling or notification when the counters wrap. Extended event logging may be accomplished by periodically reading the contents of the PICs before each overflows. Additional statistics can be collected using the two PICs over multiple passes of program execution.

Two events can be measured simultaneously by setting the PCR.select fields together with the PCR.UT and PCR.ST fields. The selected statistics are reflected during subsequent accesses to the PICs. The difference between the values read from the PIC on two subsequent reads reflects the number of events that occurred between them. Software may only rely on read-to-read counts of the PIC for accurate timing and not on write-to-read counts. See also *Table 17-5* on page 261 for the state of these registers after reset.

—	S1	—	S0	—	UT	ST	PRIV
63 15	14 11	10 8	7 4	3	2	1	0

Figure B-1 Performance Control Register (PCR)

S1|S0: Two four-bit fields; each selects a performance instrumentation event from the list in Section B.4.5, *PCR.S0 and PCR.S1 Encoding* on page 392. The event selected by S0 is counted in PIC.D0; the event selected by S1 is counted in PIC.D1.

UT: User_trace; if set, events in non-privileged (user) mode are counted. This may be set along with PCR.ST to count all selected events.

ST: System_trace; if set, events in privileged (system) mode are counted. This may be set along with PCR.UT to count all selected events.

PRIV: Privileged; if set, non-privileged access to the PIC will cause a *privileged_action* trap.

D1	D0
63 32	31 0

Figure B-2 Performance Instrumentation Counters (PIC)

D1|D0: A pair of 32-bit counters; D0 counts the events selected by PCR.S0; D1 counts the events selected by PCR.S1.

B.3 PCR/PIC Accesses

An example of the operational flow in using the performance instrumentation is shown in *Figure B-3*.

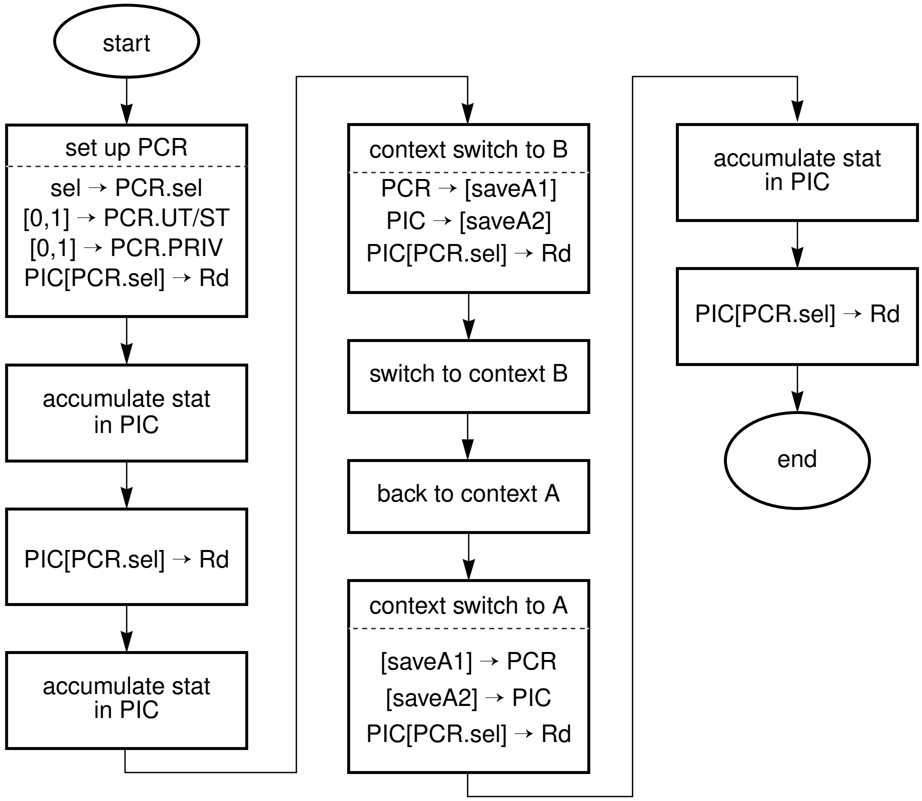


Figure B-3 PCR/PIC Operational Flow

B.4 Performance Instrumentation Counter Events

B.4.1 Instruction Execution Rates

Cycle_cnt [PIC0,PIC1]: accumulated cycles; this counter is similar to the SPARC-V9 TICK register, except that cycle counting is controlled by the PCR.UT and PCR.ST fields.

Instr_cnt [PIC0,PIC1]: the number of instructions completed; annulled, mispredicted or trapped instructions are not counted.

Using the two counters to measure instruction completion and cycles allows calculation of the average number of instructions completed per cycle.

B.4.2 Grouping (G) Stage Stall Counts

These are the major cause of pipeline stalls (bubbles) from the G Stage of the pipeline. Stalls are counted for each clock for which the associated condition is true.

Dispatch0_IC_miss [PIC0]: I-buffer is empty from I-cache miss. This includes E-cache miss processing if an E-cache miss also occurs.

Dispatch0_mispred [PIC1]: I-buffer is empty from Branch misprediction. Branch misprediction kills instructions after the dispatch point, so the total number of pipeline bubbles is approximately twice as big as measured from this count.

Dispatch0_storeBuf [PIC0]: Store buffer can not hold additional stores, and a store instruction is the first instruction in the group.

Dispatch0_FP_use [PIC1]: The first instruction in the group depends on an earlier floating point result that is not yet available, but only while the earlier instruction is not stalled for a Load_use (see Section B.4.3). Thus, Dispatch0_FP_use and Load_use are mutually exclusive counts.

Some less common stalls (see Chapter 22, *Grouping Rules and Stalls*) are not counted by any performance counter. This situation includes one cycle stalls for an FGA/FGM instruction entering the G stage following an FDIV or FSQRT.

B.4.3 Load Use Stall Counts

Stalls are counted for each clock that the associated condition is true.

Load_use [PIC0]: An instruction in the execute stage depends on an earlier load result that is not yet available. This stalls all instructions in the execute and grouping stages.

Load_use also counts cycles when no instructions are dispatched due to a one cycle load-load dependency on the first instruction presented to the grouping logic.

There are also overcounts due to, for example, mispredicted CTIs and dispatched instructions that are invalidated by traps.

Load_use_RAW [PIC1]: There is a load use in the execute stage and there is a read-after-write hazard on the oldest outstanding load. This indicates that load data is being delayed by completion of an earlier store.

Some less common stalls (see Chapter 22, *Grouping Rules and Stalls*) are not counted by any performance counter, including:

- Stalls associated with WRPR/RDPR and internal ASI loads
- MEMBAR stalls
- One cycle stalls due to bad prediction around a change to the Current Window Pointer (CWP)

B.4.4 Cache Access Statistics

I-, D-, and E-cache access statistics can be collected. Counts are updated by each cache access, regardless of whether the access will be used.

IC_ref [PIC0]: I-cache references; I-cache references are fetches of up to four instructions from an aligned block of eight instructions. I-cache references are generally prefetches and do not correspond exactly to the instructions executed.

IC_hit [PIC1]: I-cache hits

DC_rd [PIC0]: D-cache read references (including accesses that subsequently trap); non d-cacheable accesses are not counted. Atomic, block load, "internal," and "external" bad ASIs, quad precision LDD, and MEMBARs also fall into this class.

Atomic instructions, block loads, "internal" and "external" bad ASIs, quad LDD, and MEMBARs also fall into this class.

DC_rd_hit [PIC1]: D-cache read hits are counted in one of two places:

- When they access the D-cache tags and do not enter the load buffer (because it is already empty)
- When they exit the load buffer (due to a D-cache miss or a non-empty load buffer)

Loads that hit the D-cache may be placed in the load buffer for a number of reasons — because of a non-empty load buffer, for example. Such loads may be turned into misses if a snoop occurs during their stay in the load buffer (due to an external request or to an E-cache miss). In this case they do not count as D-cache read hits. See Section 21.3, *Data Stream Issues* on page 336.

DC_wr [PIC0]: D-cache write references (including accesses that subsequently trap); non D-cacheable accesses are not counted.

DC_wr_hit [PIC1]: D-cache write hits

EC_ref [PIC0]: total E-cache references; non-cacheable accesses are not counted.

EC_hit [PIC1]: total E-cache hits.

EC_write_hit_RDO [PIC0]: E-cache hits that do a read for ownership of a UPA transaction.

EC_wb [PIC1]: E-cache misses that do writebacks

EC_snoop_inv [PIC0]: E-cache invalidates from the following UPA transactions: S_INV_REQ, S_CPI_REQS_INV_REQ, S_CPI_REQS_INV_REQ, S_CPI_REQ

EC_snoop_cb [PIC1]: E-cache snoop copy-backs from the following UPA transactions: S_CPB_REQ, S_CPI_REQ, S_CPD_REQ, S_CPB_MSI_REQ

EC_rd_hit [PIC0]: E-cache read hits from D-cache misses

EC_ic_hit [PIC1]: E-cache read hits from I-cache misses

The E-cache write hit count is determined by subtracting the read hit and the instruction hit count from the total E-cache hit count. The E-cache write reference count is determined by subtracting the D-cache read miss (D-cache read references minus D-cache read hits) and I-cache misses (I-cache references minus I-cache hits) from the total E-cache references. Because of store buffer compression, this value is not the same as D-cache write misses.

Note – A block memory access is counted as a single reference. Atomics count the read and write individually.

B.4.5 PCR.S0 and PCR.S1 Encoding

TABLE B-1 PiC.S0 Selection Bit Field Encoding

S0 Value	PIC0 Selection
0000	Cycle_cnt
0001	Instr_cnt
0010	Dispatch0_IC_miss
0011	Dispatch0_storeBuf
1000	IC_ref
1001	DC_rd
1010	DC_wr
1011	Load_use
1100	EC_ref
1101	EC_write_hit_RDO
1110	EC_snoop_inv
1111	EC_rd_hit

TABLE B-2 PIC.S1 Selection Bit Field Encoding

S1 Value	PIC1 Selection
0000	Cycle_cnt
0001	Instr_cnt
0010	Dispatch0_mispred
0011	Dispatch0_FP_use
1000	IC_hit
1001	DC_rd_hit
1010	DC_wr_hit
1011	Load_use_RAW
1100	EC_hit
1101	EC_wb
1110	EC_snoop_cb
1111	EC_ic_hit

APPENDIX **C**

IEEE 1149.1 Scan Interface

C.1 Introduction

UltraSPARC IIi provides an *IEEE Std 1149.1-1990*-compliant test access port (TAP) and boundary scan architecture. The primary use of 1149.1 scan interface is for board-level interconnect testing and diagnosis.

The IEEE 1149.1 test access port and boundary scan architecture consists of three major parts:

- Test access port controller
- Instruction register
- Test data registers (numerous; public and private)

For information about how to obtain a copy of *IEEE Std 1149.1-1990*, see *Bibliography.*

C.2 Interface

The *IEEE Std 1149.1-1990* serial scan interface is composed of a set of pins and a TAP controller state machine that responds to the pins. The five wire IEEE 1149.1 interface is used in UltraSPARC IIi. *TABLE C-1* describes the five pins.

TABLE C-1 IEEE 1149.1 Signals

Signal	I/O	Description
TDO	O	Test data out. This is the scan shift output signal from either the instruction register or one of the test data registers.
TDI	I	Test data input. This forms the scan shift in signal for the instruction and various test data registers.
TMS	I	This signal is used to sequence the TAP state machine through the appropriate sequences. Holding this signal high for at least five clock cycles will force the TAP to the TEST-LOGIC-RESET state.
TCK	I	Test clock. The inputs TDI and TMS are sampled on the rising edge of TCK and the TDO output becomes valid after the falling edge of TCK.
TRST_L	I	The IEEE 1149.1 logic is asynchronously reset when TRST_L goes low.

C.3 Test Access Port Controller

The Test Access Port (TAP) controller is a 16-state synchronous finite state machine. Transitions between states occur only at the rising edge of TCK in response to the TMS signal, or when TRST_L is asserted

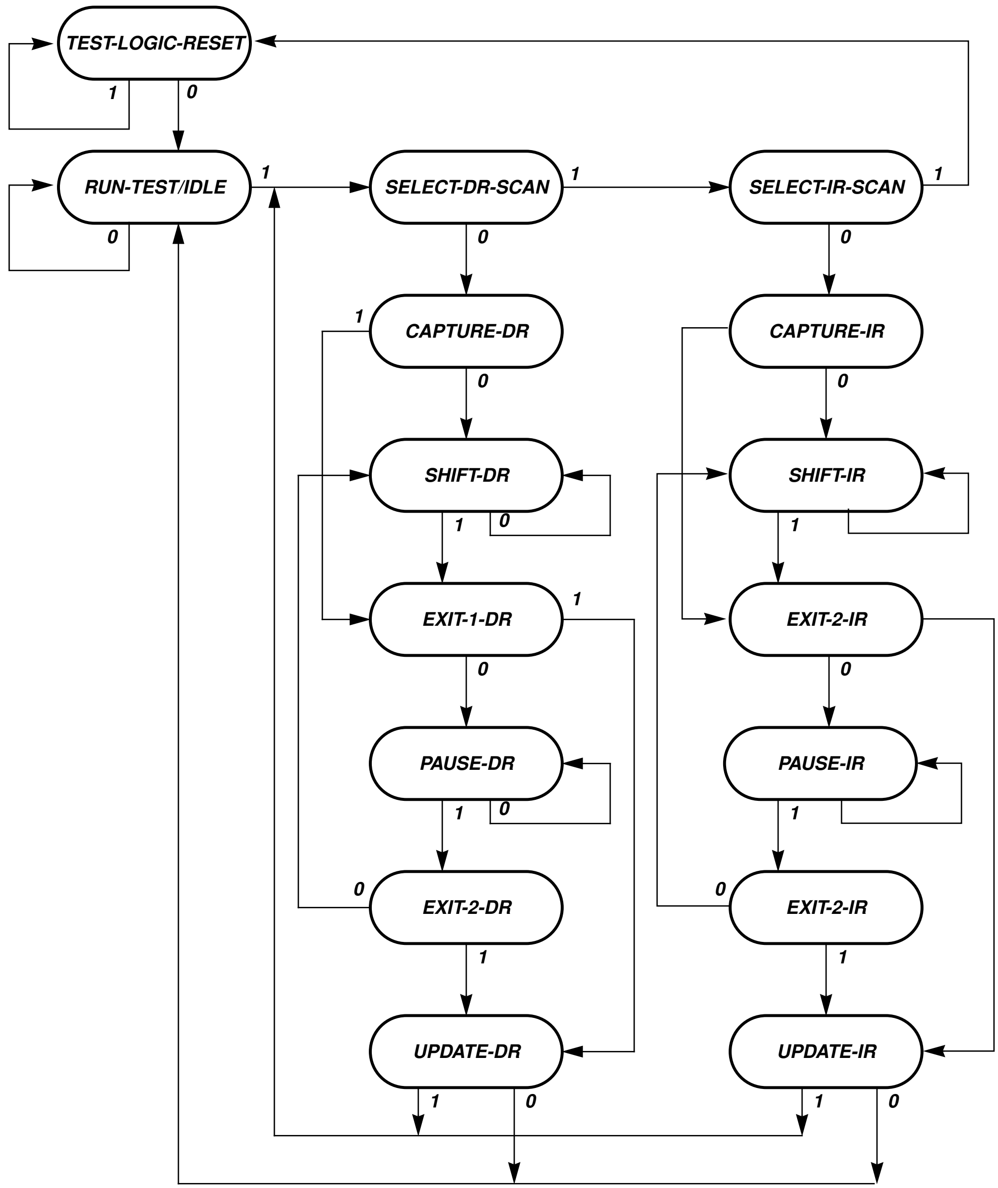


Figure C-1 TAP Controller State Diagram

Figure C-1 shows the state machine diagram. The values shown adjacent to state transitions represents the value of TMS required at the time of a rising edge of TCK for the transition to occur. Note that the IR states select the instruction register and DR states refer to states that may select a test data register, depending on the active instruction.

C.3.1 TEST-LOGIC-RESET

The TAP controller enters the TEST-LOGIC-RESET state when the TRST_L pin is asserted or when the TMS signal is held high for at least five clock cycles, regardless of the original state of the controller. It remains in this state while TMS is held high. In this state the test logic is disabled and the instruction register is initialized to select the Device ID register.

C.3.2 RUN-TEST/IDLE

RUN-TEST/IDLE is an intermediate controller state between scan operations. If no instruction is selected, all test data registers retain their current states.

Once the state machine enters this state, it remains there for as long as TMS is held low.

C.3.3 SELECT-DR-SCAN

SELECT-DR-SCAN is a temporary state in which all test data registers retain their previous states.

C.3.4 SELECT-IR-SCAN

SELECT-IR-SCAN is another temporary state in which all test data registers retain their previous states.

C.3.5 CAPTURE IR/DR

In this state, the selected register, which can be either an instruction register or a data register, loads data into its parallel input.

For the instruction register, this corresponds to sampling the eight bits of status information and loading the constant ‘01’ pattern into the two least significant bit locations.

C.3.6 SHIFT IR/DR

In this state, the IR/DR shift towards their serial output during each rising edge of TCK.

C.3.7 EXIT-1 IR/DR

This state is a temporary controller state in which the IR/DR retain their previous states.

C.3.8 PAUSE IR/DR

This state is a temporary controller state in which the IR/DR retain their previous states. It is provided to temporarily halt data-shifting through the instruction register or the test data register—without having to stop TCK.

C.3.9 EXIT-2 IR/DR

This state is a temporary controller state in which the IR/DR retain their previous states.

C.3.10 UPDATE IR/DR

Data is latched on to the parallel output of the IR/DR from the shift-register path during this controller state.

The data held at the previous outputs of the instruction register or test data register only changes in this controller state.

C.4 Instruction Register

The instruction register is used to select the test to be performed and the test data register to be accessed.

This register is 8-bits wide and consists of a serial-input/serial-output shift-register that has parallel inputs and a parallel output stage. The parallel outputs are loaded during the UPDATE-IR state with the instruction shifted into the shift register stage. This method ensures that the instruction only changes synchronously at the end of an instruction register shift or on entry to the TEST-LOGIC-RESET state. The behavior of the instruction register in each controller state is shown in *TABLE C-2*.

TABLE C-2 Instruction Register Behavior

Controller State	Shift Register	Parallel Output
TEST-LOGIC-RESET	Undefined	Set to 00_{16} (select Device ID register for shift)
CAPTURE IR	Load 01 into IR <1:0>	Retain last state
SHIFT IR	Shift towards serial output	Retain last state
UPDATE IR	Retain last state	Load from shift-register stage
All other states	Retain last state	Retain last state

At the start of an instruction register shift, that is, during the CAPTURE-IR state, a constant '01' pattern loads into the least-significant two bits to aid fault isolation in the board-level serial test data path.

C.5 Instructions

The UltraSPARC IIi 8-bit instruction register (IR) implements public and private instructions. Out of the 256 encodings possible, there are 75 valid instructions. All invalid encodings default to the BYPASS instruction as defined in IEEE Std 1149.1-1990. The public instructions implemented are: BYPASS, IDCODE, EXTEST, SAMPLE and INTEST. Private instructions are used in manufacturing and *should not* be used before consulting your SPARC sales representative. The instruction encodings and the test data register selected is presented in *TABLE C-3*.

TABLE C-3 IEEE 1149.1 Instruction Encodings

Instruction	IR encoding	Scan Chain
BYPASS	FF_{16}	bypass
IDCODE	FE_{16}	id register
EXTEST	00_{16}	boundary
SAMPLE	07_{16}	boundary
INTEST	01_{16}	boundary
PLLMODE	$9F_{16}$	pll mode
CLKCTRL	$9D_{16}$	clock control
RAMWCP	BD_{16}	ram control
POWERCUT	$8E_{16}$	N/A
HIGHZ	FD_{16}	bypass
INTEST2	$8F_{16}$	boundary
FULLSCAN	$40_{16}..7F_{16}$	internal

C.5.1 Public Instructions

C.5.1.1 BYPASS

The BYPASS instruction selects the BYPASS register as the active test data register.

C.5.1.2 SAMPLE/PRELOAD

SAMPLE/PRELOAD selects the active test data register to be the boundary scan register. Without disturbing normal processor operation, this instruction enables the I/O pin states to be observed or a value to be shifted in to the boundary scan chain.

C.5.1.3 EXTEST

EXTEST selects the boundary scan register to be the active test data register and is used to perform board level interconnect testing. In this condition the boundary scan chain drives the processor pins and UltraSPARC IIi cannot function normally.

C.5.1.4 INTEST

This instruction selects the boundary scan register to be the active test data register. allowing it to be used as a virtual low-speed functional tester. The on-chip clock is derived from TCK and is issued in the Run-Test/Idle state of the TAP controller.

C.5.1.5 IDCODE

IDCODE selects the ID register for shifting.

C.5.2 Private Instructions

All private instructions: PLLMODE, CLKCTRL, RAMWCP, POWERCUT, HIGHZ, INTEST2, and all versions of FULLSCAN should not be used before consulting your SPARC sales representative. Improper use of any private instructions can permanently damage UltraSPARC IIi and render it inoperative.

C.6 Public Test Data Registers

C.6.1 Device ID Register

The 32-bit Device ID register is loaded with the UltraSPARC IIi ID upon entering the CAPTURE-DR TAP state when the ID instruction is active or during the TEST-LOGIC-RESET state. *Figure C-2* shows the structure of the Device ID Register.

Version	0100 0110 0110 1000	000 0100 0101	1
31 28	27 12	11 1	0

Figure C-2 Device ID Register

The device ID is loaded into the register on the rising edge of TCK in the Capture-DR state. The value of ID<27:0> is fixed at $4668045F_{16}$ and the version number, ID<31:28>, changes as specified in IEEE Std 1149.1-1990.

C.6.2 Bypass Register

This register provides a single bit delay between TDI and TDO. During the CAPTURE-DR controller state, and if it is selected by the current instruction, the bypass register loads a logical zero.

C.6.3 Boundary Scan Register

The Boundary Scan Register allows for testing circuitry external to the device; for example:

- testing the interconnect by setting defined values at the device periphery – using the EXTEST instruction
- sampling and examination of pin states without disturbing the system – using the SAMPLE/PRELOAD instruction
- testing device function itself – using the INTEST instruction

The boundary scan register for UltraSPARC IIi is 770 bits long. The mapping between register bits and the pin signals is described in a Boundary Scan Description Language (BSDL) file available from your SPARC sales representative.

Note – It is recommended that transitions from the Capture-DR TAP controller state to the Shift-DR controller state progress through the Exit1-DR, Pause-DR, and Exit2-DR states. A direct progression from Capture-DR to Shift-DR is not recommended when the boundary scan register is selected.

C.6.4 Private Data Registers

Private data registers should not be accessed before consulting your SPARC sales representative.

APPENDIX **D**

ECC Specification

D.1 ECC Code

The 64-bit ECC code specification can be found in Shigeo Kaneda's correspondence note: *"A Class of Odd-Weight-Column SEC-DED-SbED Codes for Memory System Applications", IEEE Transactions on Computers, August 1984.*

TABLE D-1 shows the syndrome table for the ECC code, followed by the Verilog code for error detection, correction, and syndrome generation.

TABLE D-1 Syndrome table for ECC SEC/S4ED code .

SYND bits																
7	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
6	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
5	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
4	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
0 1 2 3																
0 0 0 0	*	C4	C5	D	C6	D	D	T	C7	D	D	T	D	T	T	Q
0 0 0 1	C0	D	D	00	D	25	M	D	D	05	17	D	08	D	D	12
0 0 1 0	C1	D	D	01	D	29	36	D	D	M	21	D	13	D	D	09
0 0 1 1	D	32	33	D	42	D	D	M	47	D	D	M	D	T	T	D
0 1 0 0	C2	D	D	10	D	27	07	D	D	M	19	D	02	D	D	14
0 1 0 1	D	57	61	D	59	Q	D	M	63	D	Q	M	D	M	M	D
0 1 1 0	D	M	04	D	39	D	D	22	M	D	D	30	D	16	24	D
0 1 1 1	T	D	D	M	D	M	54	D	D	50	M	D	T	D	D	M
1 0 0 0	C3	D	D	15	D	31	M	D	D	38	23	D	03	D	D	11

TABLE D-1 Syndrome table for ECC SEC/S4ED code *(Continued)*.

SYND bits 7 6 5 4 / 0 1 2 3	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
1 0 0 1	D	37	M	D	M	D	D	18	06	D	D	26	D	20	28	D
1 0 1 0	D	49	53	D	51	Q	D	M	55	D	Q	M	D	M	M	D
1 0 1 1	T	D	D	M	D	M	62	D	D	58	M	D	T	D	D	M
1 1 0 0	D	40	45	D	34	D	D	T	35	D	D	T	D	M	M	D
1 1 0 1	T	D	D	T	D	M	48	D	D	52	M	D	M	D	D	M
1 1 1 0	T	D	D	T	D	M	56	D	D	60	M	D	M	D	D	M
1 1 1 1	Q	44	41	D	46	D	D	M	43	D	D	M	D	M	M	Q

Code Example D-1 describes the check bit generation equations in the most concise way

Code Example D-1 Description of ECC checkbit Generation Equations

```
function [7:0] get_ecc8;
input [63:0] data;
begin
        get_ecc8[7:0] = {
                     ^(64'h9494884855bb7b6c & data[63:0]),
                     ^(64'h49494494bb557b8c & data[63:0]),
                     ^(64'h6161221255eede93 & data[63:0]),
                     ^(64'h16161161ee55de23 & data[63:0]),
                     ^(64'h55bb7b6c94948848 & data[63:0]),
                     ^(64'hbb557b8c49494494 & data[63:0]),
                     ^(64'h55eede9361612212 & data[63:0]),
                     ^(64'hee55de2316161161 & data[63:0]) };
end
endfunction
```

APPENDIX **E**

UPA64S interface

E.1 UPA64S Bus

The UPA64S bus transfers data in a packetized mode between UltraSPARC IIi and system DRAM. In addition it is used to transfer data to a connected UPA64S device, for example, a Fast Frame Buffer (FFB).

E.1.1 Data Bus (MEMDATA)

MEMDATA is a 72-bit bidirectional bus between UltraSPARC IIi and the memory transceivers. Bits[63:0] are also used to connect to a UPA64S device.

The transaction set supports block transfers of 64 bytes; and quadword noncached transfers of 1 to 16 bytes, qualified with a 16-bit bytemask. Data transfers are 8 bytes per UPA clock cycle on MEMDATA[63:0].

Figure E-1 illustrates how data and ECC bytes are arranged and addressed within a doubleword.

	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
Dword Bytes	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7

Figure E-1 Data Byte Addresses Within a Dword

E.1.2 SYSADDR Bus

UltraSPARC IIi directly sends a request to the UPA64S slave, using SYSADDR and ADR_VLD, which are always driven.

E.2 UPA64S Transaction Overview

- P_REQ transaction request from UltraSPARC IIi to the UPA64S device on the SYSADDR bus; these transactions initiate activity.
- P_REPLY by UPA64S device is generated in response to a previous P_REQ transaction; indicates read data available, or write data consumed.
- S_REPLY by the UltraSPARC IIi CPU initiates transfer of data.

E.2.1 NonCachedRead (P_NCRD_REQ)

Noncached Read; generated by UltraSPARC IIi for a load or instruction fetch to noncached UPA64S address.

1, 2, 4, 8, and 16 bytes are read with this transaction, and the byte location is specified with a bytemask. The address is aligned on a 16-byte boundary. The bytemask is aligned on a natural boundary that matches the total data size.

One P_NCRD_REQ may be outstanding to UPA64S device at a time. The next P_NCRD_REQ request can be sent on the cycle after the P_RASB reply.

Data is transferred with S_SRS reply.

E.2.2 NonCachedBlockRead (P_NCBRD_REQ)

Noncached Block Read Request; 64 bytes of non-cached data is read with this transaction generated by UltraSPARC IIi for block read of a non-cached UPA64S address space.

Similar to P_NCRD_REQ except that there is no bytemask; the data is aligned on a 64-byte boundary (PA<5:4> = 016).

Data is delivered with S_SRB reply.

E.2.3 NonCachedWrite (P_NCWR_REQ)

Noncached Write; generated by UltraSPARC IIi to write a non-cached address UPA64S space.

The address is aligned on 16-byte boundary. An arbitrary number of 0-16 bytes can be written as specified by a 16-bit bytemask to slave devices that support writes with arbitrary byte masks (mainly graphics devices). A bytemask of all zeros indicates a no-op at the slave.

S_SWS is used to transfer the data. When UltraSPARC IIi drives the S_REPLY, it considers the transaction completed and decrements the count of outstanding requests for flow control.

E.2.4 NonCachedBlockWrite (P_NCBWR_REQ)

Noncached Block Write Request; 64 bytes of noncached data is written by UltraSPARC IIi with this transaction; generated for block store to a non-cached UPA64S address.

Similar to P_NCWR_REQ except that there is no bytemask; the data is aligned on a 64-byte boundary (PA<5:4> = 016).

Data is transferred with S_SWB reply.

E.3 P_REPLY and S_REPLY

E.3.1 P_REPLY

The UPA64S device drives P_REPLY<1:0> to UltraSPARC IIi. All P_REPLYs are generated as an acknowledgment by the UPA64S device in response to a request previously sent by the UltraSPARC IIi CPU.

TABLE E-1 P_REPLY Type Definitions

Type	Definition
P_IDLE	*Idle*. The default state of the wires when there is no reply to be given.
P_RASB	Read Ack single and Block. **16** or **64** bytes of data are ready in its output data queue for the **P_NCRD_REQ** | **P_NCBRD_REQ** request sent to it, and there is room in its input request queue for another **P_REQ**. The UltraSPARC IIi CPU knows, from programmable registers, the depth of the queues on the UPA64S device, and does not cause the queues to be overflowed, or underflowed.
P_WAS	Write Ack Single; reply to **P_NCWR_REQ** request for single writes The UPA64S port acknowledges that the **16** bytes of data placed in its input data queue has been absorbed, and there is room for writing another **16** bytes of data into the input data queue, and there is room in its input request queue for another slave **P_REQ** for data.
P_WAB	Write Ack Block; reply to **P_NCBWR_REQ** for block write; the UPA64S slave port acknowledges that the **64** bytes of data placed in its input data queue has been absorbed, and there is room for writing another **64** bytes of data into the input data queue, and there is room in its input request queue for another slave **P_REQ** for data.

TABLE E-2 shows the encodings for the transactions defined in *TABLE E-1*.

TABLE E-2 P_REPLY<1:0> Encoding

P_REPLY	Name	Reply to Transaction	
P_IDLE	Idle	Default State	00
P_WAB	Write ACK Block	P_NCBWR_REQ	01
P_WAS	Write ACK Single	P_NCWR_REQ	10
P_RASB	Read ACK Single/Block	P_NCRD_REQ, P_NCBRD_REQ	11

E.3.2 S_REPLY

S_REPLY is a 3-bit signal between UltraSPARC IIi and the UPA64S device. *TABLE E-4* specifies the S_REPLY encoding.

S_REPLY takes a single UPA clock cycle, and initiates data transfer on MEMDATA. The encoding for S_IDLE is 00. (also driven during reset).

TABLE E-3 specifies the S_REPLY types. The following rules apply to S_REPLY generation:

- The S_REPLY is strongly ordered with respect to requests.

- The S_REPLY timing to the source and sink of data is shown in *Figure E-2* and *Figure E-3*.The UPA64S device drives the data 2 UPA clock cycles after receiving S_SRS | S_SRB. UPA64S receives data 1 UPA clock cycle after S_SWS | S_SWB
- The S_REPLY read data timing after receiving a P_REPLY from is shown in *Figure E-4*. The ***minimum*** number of clock cycles between the P_REPLY and the S_REPLY is two; that is, this number represents the earliest time after receiving P_REPLY that S_REPLY can be sent to get the data.
- S_REPLY can be pipelined such that the MEMDATA bus can be kept continually busy without any dead cycles on the MEMDATA bus, as long as the same source is driving the data
- If sources are switched, one dead cycle is required on the MEMDATA bus; this allows the first source to switch off before the next source can drive the data. The earliest that the next source can drive the data is in the cycle following the dead cycle; thus, the pipelining of data accompanying S_REPLY types is adjusted accordingly with one extra bubble for the dead cycle.
- The ordering of S_REPLY for delivering data to a UPA64S device is shown in *Figure E-5*.

TABLE E-3 S_REPLY Type Definitions

Type	Definition
S_IDLE	*Idle*. The default state; indicates no reply.
S_SRS	Read Single Ack; the output data queue of the UPA64S device drives **16** bytes of read data in response to **P_RAS** reply.
S_SRB	Read Block Ack; the output data queue of the UPA64S device drives **64** bytes of read data in response to **P_RAB** reply from it.
S_SWB	Write Block Ack; the input data queue of the UPA64S device accepts a **64** bytes of data.
S_SWS	Write Single Ack; the input data queue of the UPA64S device accepts **16** bytes of data.

TABLE E-4 S_REPLY Encoding

S_REPLY	Name	Reply to Transaction	
S_IDLE	Idle	Default State	000
S_SWS	Slave Write Single	P_NCWR_REQ	100
S_SWB	Slave Write Block	P_NCBWR_REQ	101
S_SRS	Slave Read Single	P_NCRD_REQ	110
S_SRB	Slave Read Block	P_NCBRD_REQ	111

E.3.3 P_REPLY and S_REPLY Timing

The following figures show the control of data flow on the MEMDATA bus due to S_REPLY and P_REPLY.

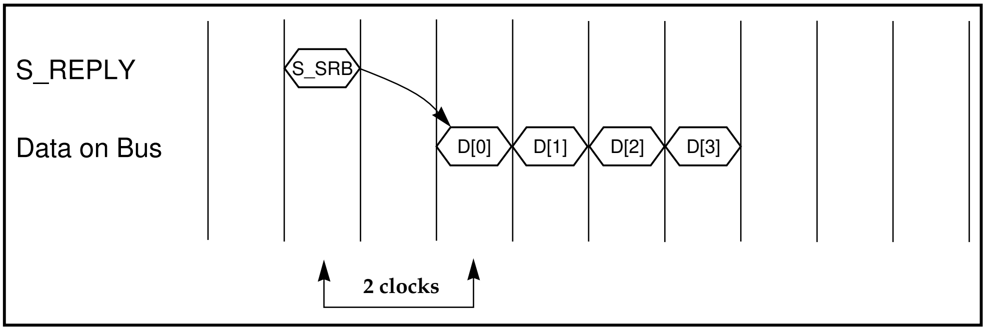


Figure E-2 S_REPLY Timing: UPA64S device Sourcing Block

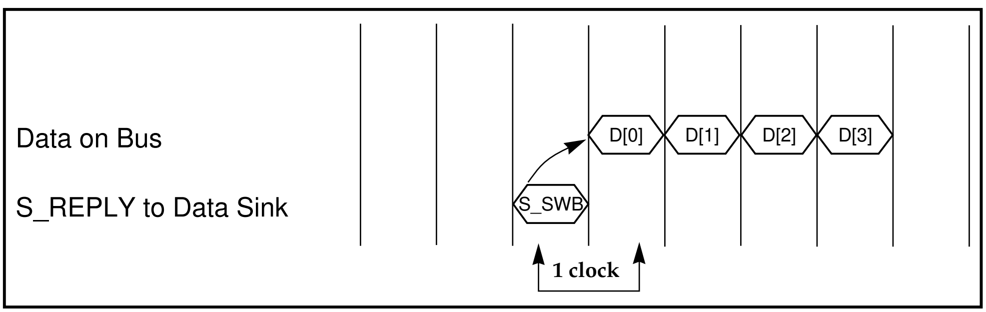


Figure E-3 S_REPLY Timing: UPA64S device Sinking Block

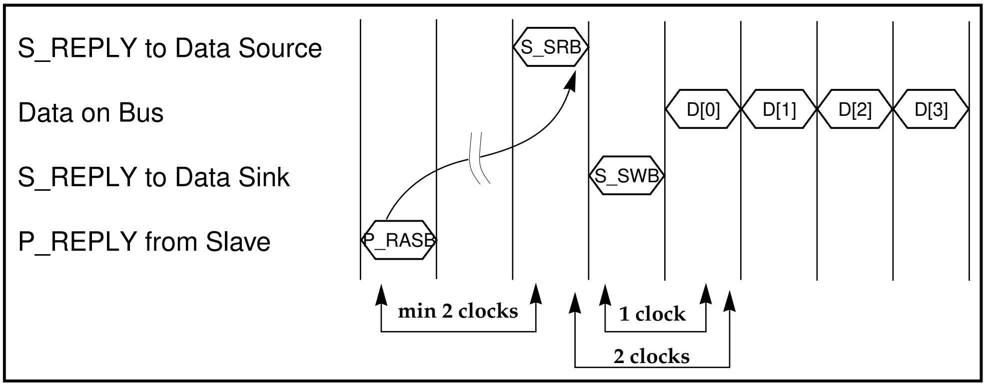


Figure E-4 P_REPLY to S_REPLY Timing

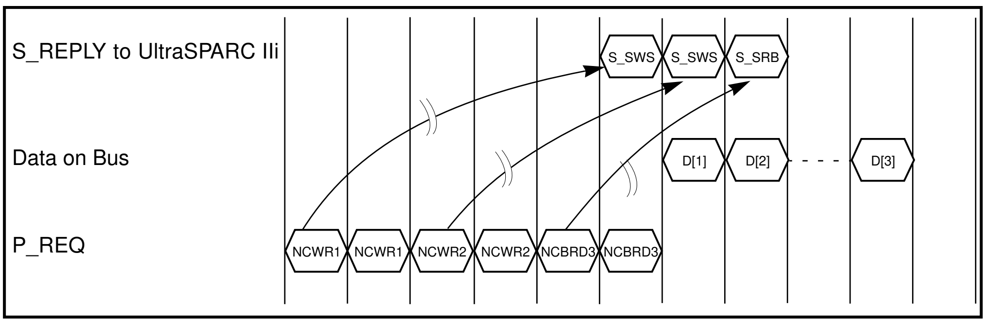


Figure E-5 S_REPLY Pipelining

E.4 Issues with Multiple Outstanding Transactions

E.4.1 Strong Ordering

All prior 16-byte noncacheable stores (P_NCWR_REQ) must complete before completing a P_NCRD_REQ. This condition is necessary to meet a software requirement that all noncacheable operations can be strongly ordered. The E-bit feature of UltraSPARC IIi does not wait for prior noncacheable operations to complete (as do MEMBARs).

While a 16-byte noncacheable load is outstanding (P_NCRD_REQ), UltraSPARC IIi will not issue any more transactions, so the reverse case—completing noncacheable loads before noncacheable stores—does not occur.

E.4.2 Limiting the Number of Transactions

UltraSPARC IIi can limit the total number of outstanding transactions, and additionally, can limit the amount of outstanding data creating by outstanding stores.

E.4.3 S_REPLY assertion

The assertion of S_REPLYs must guarantee that there is at least one dead cycle between different drivers (for example, port and memory). No dead cycle is required for multiple packets from the same driver.

E.5 UPA64S Packet Formats

E.5.1 Request Packets

The SYSADDR bus is a 29-bit transaction request bus. The request packet comprises 58 bits and is carried on the SYSADDR bus in *two* successive UPA64S clock cycles.

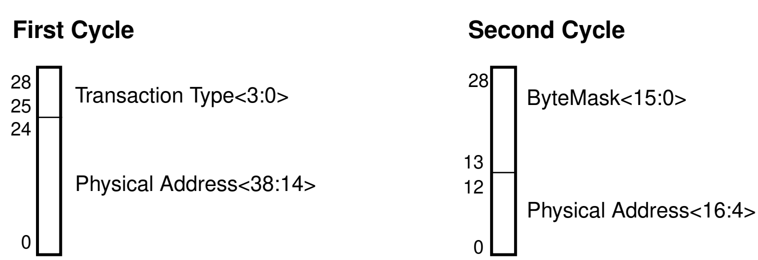


Figure E-6 Packet Format: Noncached P_REQ Transactions

E.5.2 Packet Description

E.5.2.1 Transaction Type

This 4-bit field encodes the transaction type, as shown in *TABLE E-5*.

TABLE E-5 Transaction Type Encoding

Transaction Type	Name	Type<3:0>
P_NCRD_REQ	NonCachedRead	0101
P_NCBRD_REQ	NonCachedBlockRead	0110
P_NCBWR_REQ	NonCachedBlockWrite	0111
P_NCWR_REQ	NonCachedWrite	1110

E.5.2.2 Physical Address PA<38:4>

Bits PA<38:4> of the 39-bit physical address space accessible to UltraSPARC IIi.

The low order 4 bits PA<3:0> of the physical address are implied in the bytemask in P_NCRD_REQ and P_NCWR_REQ transactions. All other transactions transfer 64-byte blocks and do not need PA<3:0>, since it is 0_{16}.

E.5.2.3 Bytemask<15:0>

Bytemask is only available for P_NCRD_REQ and P_NCWR_REQ. This 16-bit field indicates valid bytes on MEMDATA. The bytemask can be 1-, 2-, 4-, 8- and 16-byte for non-cached read requests; *arbitrary bytemasks are allowed for slave writes*. An all-zero bytemask indicates a no-op at the slave.

Bytemask<0> corresponds to byte 0 (bits <63:56> in cycle 0 on the 64-bit data bus.

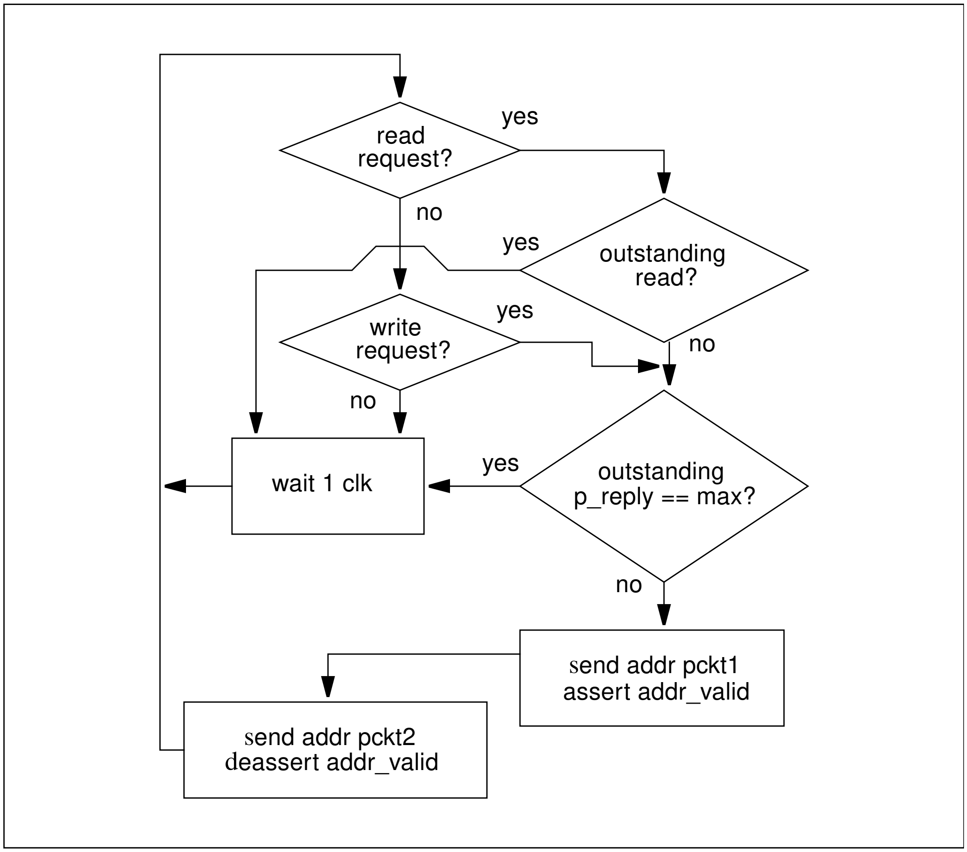


Figure E-7 UPA64s Transactions Flowchart—Address Bus

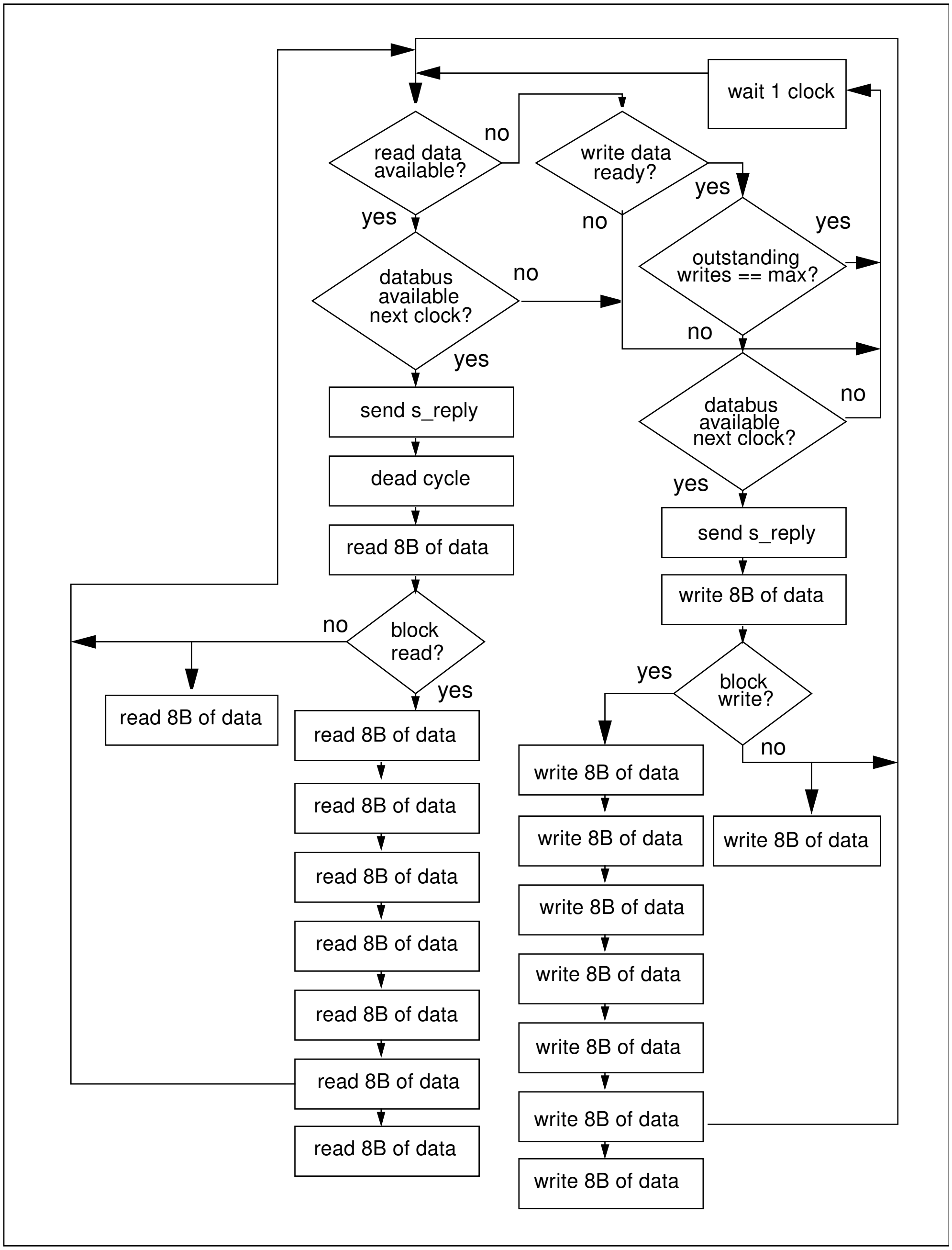


Figure E-8 UPA64s Transactions Flowchart—Data Bus

APPENDIX **F**

Pin and Signal Descriptions

Consult the relevant data sheets for detailed information about the electrical and mechanical characteristics of the processor, including pin and pad assignments. *Bibliography* on page 465 describes the available data sheets and how to obtain them.

APPENDIX **G**

ASI Names

G.1 Introduction

This Appendix lists the names and suggested macro syntax for all supported Address Space Identifiers.

Table G-1 ASI registers and ASI Names—listed alphabetically

ASI Name or Macro Syntax	Description	Value
ASI_AFAR	Asynchronous fault address register	$4D_{16}$
ASI_AFSR	Asynchronous fault status register	$4C_{16}$
ASI_AIUP	Primary address space, user privilege	10_{16}
ASI_AIUPL	Primary address space, user privilege, little endian	18_{16}
ASI_AIUS	Secondary address space, user privilege	11_{16}
ASI_AIUSL	Secondary address space, user privilege, little endian	19_{16}
ASI_AS_IF_USER_PRIMARY	Primary address space, user privilege	10_{16}
ASI_AS_IF_USER_PRIMARY_LITTLE	Primary address space, user privilege, little endian	18_{16}
ASI_AS_IF_USER_SECONDARY	Secondary address space, user privilege	11_{16}
ASI_AS_IF_USER_SECONDARY_LITTLE	Secondary address space, user privilege, little endian	19_{16}
ASI_BLK_AIUP	Primary address space, block load/store, user privilege	70_{16}
ASI_BLK_AIUPL	Primary address space, block load/store, user privilege, little endian	78_{16}
ASI_BLK_AIUS	Secondary address space, block load/store, user privilege	71_{16}

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_BLK_AIUSL	Secondary address space, block load/store, user privilege, little endian	79_{16}
ASI_BLK_COMMIT_P	Primary address space, block store commit operation	$E0_{16}$
ASI_BLK_COMMIT_PRIMARY	Primary address space, block store commit operation	$E0_{16}$
ASI_BLK_COMMIT_S	Secondary address space, block store commit operation	$E1_{16}$
ASI_BLK_COMMIT_SECONDARY	Secondary address space, block store commit operation	$E1_{16}$
ASI_BLK_P	Primary address space, block load/store	$F0_{16}$
ASI_BLK_PL	Primary address space, block load/store, little endian	$F8_{16}$
ASI_BLK_S	Secondary address space, block load/store	$F1_{16}$
ASI_BLK_SL	Secondary address space, block load/store, little endian	$F9_{16}$
ASI_BLOCK_AS_IF_USER_PRIMAR Y	Primary address space, block load/store, user privilege	70_{16}
ASI_BLOCK_AS_IF_USER_PRIMARY_LI TTLE	Primary address space, block load/store, user privilege, little endian	78_{16}
ASI_BLOCK_AS_IF_USER_SECONDAR Y	Secondary address space, block load/store, user privilege	71_{16}
ASI_BLOCK_AS_IF_USER_SECONDAR Y_LITTLE	Secondary address space, block load/store, user privilege, little endian	79_{16}
ASI_BLOCK_PRIMARY	Primary address space, block load/store	$F0_{16}$
ASI_BLOCK_PRIMARY_LITTLE	Primary address space, block load/store, little endian	$F8_{16}$
ASI_BLOCK_SECONDARY	Secondary address space, block load/store	$F1_{16}$
ASI_BLOCK_SECONDARY_LITTLE	Secondary address space, block load/store, little endian	$F9_{16}$
ASI_D-MMU	D-MMU Tag Target Register	58_{16}
ASI_DCACHE_DAT A	D-cache data RAM diagnostics access	46_{16}
ASI_DCACHE_DATA	D-cache data RAM diagnostics access	46_{16}
ASI_DCACHE_TAG	D-cache tag/valid RAM diagnostics access	47_{16}
ASI_DMMU	D-MMU PA Data Watchpoint Register	58_{16}
ASI_DMMU	D-MMU Secondary Context Register	58_{16}
ASI_DMMU	D-MMU Synch. Fault Address Register	58_{16}

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_DMMU	D-MMU Synch. Fault Status Register	58_{16}
ASI_DMMU	D-MMU Tag Target Register	58_{16}
ASI_DMMU	D-MMU TLB Tag Access Register	58_{16}
ASI_DMMU	D-MMU TSB Register	58_{16}
ASI_DMMU	D-MMU VA Data Watchpoint Register	58_{16}
ASI_DMMU	I/D MMU Primary Context Register	58_{16}
ASI_DMMU_DEMAP	DMMU TLB demap	$5F_{16}$
ASI_DMMU_TSB_64KB_PTR_RE G	D-MMU TSB 64K Pointer Register	$5A_{16}$
ASI_DMMU_TSB_64KB_PTR_REG	D-MMU TSB 64K Pointer Register	$5A_{16}$
ASI_DMMU_TSB_8KB_PTR_REG	D-MMU TSB 8K Pointer Register	59_{16}
ASI_DMMU_TSB_DIRECT_PTR_REG	D-MMU TSB Direct Pointer Register	$5B_{16}$
ASI_DTLB_DATA_ACCESS_REG	D-MMU TLB Data Access Register	$5D_{16}$
ASI_DTLB_DATA_IN_REG	D-MMU TLB Data In Register	$5C_{16}$
ASI_DTLB_TAG_READ_REG	D-MMU TLB Tag Read Register	$5E_{16}$
ASI_ECACHE_R	E-cache data RAM diagnostic read access	$7E_{16}$
ASI_ECACHE_R	E-cache tag/valid RAM diagnostic read access	$7E_{16}$
ASI_ECACHE_TAG_DATA	E-cache tag/valid RAM data diagnostic access	$4E_{16}$
ASI_ECACHE_W	E-cache data RAM diagnostic write access	76_{16}
ASI_ECACHE_W	E-cache tag/valid RAM diagnostic write access	76_{16}
ASI_EC_R	E-cache data RAM diagnostic read access	$7E_{16}$
ASI_EC_R	E-cache tag/valid RAM diagnostic read access	$7E_{16}$
ASI_EC_TAG_DATA	E-cache tag/valid RAM data diagnostic access	$4E_{16}$
ASI_EC_W	E-cache data RAM diagnostic write access	76_{16}
ASI_EC_W	E-cache tag/valid RAM diagnostic write access	76_{16}
ASI_ESTATE_ERROR_EN_REG	E-cache error enable register	$4B_{16}$
ASI_Fl16_P	Primary address space, one 16-bit floating-point load/store	$D2_{16}$
ASI_FL16_PL	Primary address space, one 16-bit floating-point load/store, little endian	DA_{16}
ASI_FL16_PRIMARY	Primary address space, one 16-bit floating-point load/store	$D2_{16}$
ASI_FL16_PRIMARY_LITTLE	Primary address space, one 16-bit floating-point load/store, little endian	DA_{16}

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_FL16_S	Secondary address space, one 16- bit floating-point load/store	$D3_{16}$
ASI_FL16_SECONDARY	Secondary address space, one 16- bit floating-point load/store	$D3_{16}$
ASI_FL16_SECONDARY_LITTLE	Secondary address space, one 16- bit floating-point load/store, little endian	DB_{16}
ASI_FL16_SL	Secondary address space, one 16- bit floating-point load/store, little endian	DB_{16}
ASI_FL8_P	Primary address space, one 8-bit floating-point load/store	$D0_{16}$
ASI_FL8_PL	Primary address space, one 8-bit floating-point load/store, little endian	$D8_{16}$
ASI_FL8_PRIMARY	Primary address space, one 8-bit floating-point load/store	$D0_{16}$
ASI_FL8_PRIMARY_LITTLE	Primary address space, one 8-bit floating-point load/store, little endian	$D8_{16}$
ASI_FL8_S	Secondary address space, one 8-bit floating-point load/store	$D1_{16}$
ASI_FL8_SECONDARY	Secondary address space, one 8-bit floating-point load/store	$D1_{16}$
ASI_FL8_SECONDARY_LITTLE	Secondary address space, one 8-bit floating-point load/store, little endian	$D9_{16}$
ASI_FL8_SL	Secondary address space, one 8-bit floating-point load/store, little endian	$D9_{16}$
ASI_ICACHE_INSTR	I-cache instruction RAM diagnostic access	66_{16}
ASI_ICACHE_NEXT_FIELD	I-cache next-field RAM diagnostics access	$6F_{16}$
ASI_ICACHE_PRE_DECODE	I-cache pre-decode RAM diagnostics access	$6E_{16}$
ASI_ICACHE_TAG	I-cache tag/valid RAM diagnostic access	67_{16}
ASI_IC_INSTR	I-cache instruction RAM diagnostic access	66_{16}
ASI_IC_NEXT_FIELD	I-cache next-field RAM diagnostics access	$6F_{16}$
ASI_IC_PRE_DECODE	I-cache pre-decode RAM diagnostics access	$6E_{16}$
ASI_IC_TAG	I-cache tag/valid RAM diagnostic access	67_{16}
ASI_IMMU	I-MMU Synchronous Fault Status Register	50_{16}
ASI_IMMU	I-MMU Tag Target Register	50_{16}
ASI_IMMU	I-MMU TLB Tag Access Register	50_{16}
ASI_IMMU	I-MMU TSB Register	50_{16}

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_IMMU_DEMAP	I-MMU TLB demap	57_{16}
ASI_IMMU_TSB_64KB_PTR_REG	I-MMU TSB 64KB Pointer Register	52_{16}
ASI_IMMU_TSB_8KB_PTR_REG	I-MMU TSB 8KB Pointer Register	51_{16}
ASI_INTR_DISPATCH_STATUS	Interrupt vector dispatch status	48_{16}
ASI_INTR_RECEIVE	Interrupt vector receive status	49_{16}
ASI_ITLB_DATA_ACCESS_REG	I-MMU TLB Data Access Register	55_{16}
ASI_ITLB_DATA_IN_REG	I-MMU TLB Data In Register	54_{16}
ASI_ITLB_TAG_READ_RE G	I-MMU TLB Tag Read Register	56_{16}
ASI_ITLB_TAG_READ_REG	I-MMU TLB Tag Read Register	56_{16}
ASI_LSU_CONTROL_REG	Load/store unit control register	45_{16}
ASI_N	Implicit address space, nucleus privilege, TL > 0,	04_{16}
ASI_NL	Implicit address space, nucleus privilege, TL > 0, little endian	$0C_{16}$
ASI_NUCLEUS	Implicit address space, nucleus privilege, TL > 0,	04_{16}
ASI_NUCLEUS_LITTLE	Implicit address space, nucleus privilege, TL > 0, little endian	$0C_{16}$
ASI_NUCLEUS_QUAD_LDD	Cacheable, 128-bit atomic LDDA	24_{16}
ASI_NUCLEUS_QUAD_LDD_L	Cacheable, 128-bit atomic LDDA, little endian	$2C_{16}$
ASI_NUCLEUS_QUAD_LDD_LITTLE	Cacheable, 128-bit atomic LDDA, little endian	$2C_{16}$
ASI_P	Implicit primary address space	80_{16}
ASI_PHYS_BYPASS_EC_WITH_EBIT	Physical address, noncacheable, with side-effect	15_{16}
ASI_PHYS_BYPASS_EC_WITH_EBIT_L	Physical address, noncacheable, with side-effect, little endian	$1D_{16}$
ASI_PHYS_BYPASS_EC_WITH_EBIT_LITTLE	Physical address, noncacheable, with side-effect, little endian	$1D_{16}$
ASI_PHYS_USE_EC	Physical address, external cacheable only	14_{16}
ASI_PHYS_USE_EC_L	Physical address, external cacheable only, little endian	$1C_{16}$
ASI_PHYS_USE_EC_LITTLE	Physical address, external cacheable only, little endian	$1C_{16}$
ASI_PL	Implicit primary address space, little endian	88_{16}
ASI_PNF	Primary address space, no fault	82_{16}
ASI_PNFL	Primary address space, no fault, little endian	$8A_{16}$
ASI_PRIMARY	Implicit primary address space	80_{16}

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_PRIMARY_LITTLE	Implicit primary address space, little endian	88_{16}
ASI_PRIMARY_NO_FAULT	Primary address space, no fault	82_{16}
ASI_PRIMARY_NO_FAULT_LITTLE	Primary address space, no fault, little endian	$8A_{16}$
ASI_PST16_PL	Primary address space,4 16-bit partial store, little endian	CA_{16}
ASI_PST16_PRIMARY	Primary address space,4 16-bit partial store	$C2_{16}$
ASI_PST16_PRIMARY_LITTLE	Primary address space,4 16-bit partial store, little endian	CA_{16}
ASI_PST16_S	Secondary address space,4 16-bit partial store	$C3_{16}$
ASI_PST16_SECONDARY	Secondary address space,4 16-bit partial store	$C3_{16}$
ASI_PST16_SECONDARY_LITTLE	Secondary address space,4 16-bit partial store, little endian	CB_{16}
ASI_PST16_SL	Secondary address space,4 16-bit partial store, little endian	CB_{16}
ASI_PST32_P	Primary address space, 2 32-bit partial store	$C4_{16}$
ASI_PST32_PL	Primary address space, 2 32-bit partial store, little endian	CC_{16}
ASI_PST32_PRIMARY	Primary address space, 2 32-bit partial store	$C4_{16}$
ASI_PST32_PRIMARY_LITTLE	Primary address space, 2 32-bit partial store, little endian	CC_{16}
ASI_PST32_S	Secondary address space, 2 32-bit partial store	$C5_{16}$
ASI_PST32_SECONDARY	Secondary address space, 2 32-bit partial store	$C5_{16}$
ASI_PST32_SECONDARY_LITTLE	Secondary address space, 2 32-bit partial store, little endian	CD_{16}
ASI_PST32_SL	Secondary address space, 2 32-bit partial store, little endian	CD_{16}
ASI_PST8_P	Primary address space, 8 8-bit partial store	$C0_{16}$
ASI_PST8_PL	Primary address space, 8 8-bit partial store, little endian	$C8_{16}$
ASI_PST8_PRIMARY	Primary address space, 8 8-bit partial store	$C0_{16}$
ASI_PST8_PRIMARY_LITTLE	Primary address space, 8 8-bit partial store, little endian	$C8_{16}$
ASI_PST8_S	Secondary address space, 8 8-bit partial store	$C1_{16}$
ASI_PST8_SECONDARY	Secondary address space, 8 8-bit partial store	$C1_{16}$

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_PST8_SECONDARY_LITTLE	Secondary address space, 8 8-bit partial store, little endian	$C9_{16}$
ASI_PST8_SL	Secondary address space, 8 8-bit partial store, little endian	$C9_{16}$
ASI_PSY16_P	Primary address space,4 16-bit partial store	$C2_{16}$
ASI_S	Implicit secondary address space	81_{16}
ASI_SECONDARY	Implicit secondary address space	81_{16}
ASI_SECONDARY_LITTLE	Implicit secondary address space, little endian	89_{16}
ASI_SECONDARY_NO_FAULT	Secondary address space, no fault	83_{16}
ASI_SECONDARY_NO_FAULT_LITTLE	Secondary address space, no fault, little endian	$8B_{16}$
ASI_SL	Implicit secondary address space, little endian	89_{16}
ASI_SNF	Secondary address space, no fault	83_{16}
ASI_SNFL	Secondary address space, no fault, little endian	$8B_{16}$
ASI_UDB L_CONTROL_R	External UDB Control Register, read low	$7F_{16}$
ASI_UDBH_CONTROL_R	External UDB Control Register, read high	$7F_{16}$
ASI_UDBH_CONTROL_REG_READ	External UDB Control Register, read high	$7F_{16}$
ASI_UDBH_CONTROL_REG_WRITE	External UDB Control Register, write high	77_{16}
ASI_UDBH_ERROR_R	External UDB Error Register, read high	$7F_{16}$
ASI_UDBH_ERROR_REG_READ	External UDB Error Register, read high	$7F_{16}$
ASI_UDBH_ERROR_REG_WRITE	External UDB Error Register, write high	77_{16}
ASI_UDBL_CONTROL_REG_READ	External UDB Control Register, read low	$7F_{16}$
ASI_UDBL_CONTROL_REG_WRITE	External UDB Control Register, write low	77_{16}
ASI_UDBL_ERROR_R	External UDB Error Register, read low	$7F_{16}$
ASI_UDBL_ERROR_REG_READ	External UDB Error Register, read low	$7F_{16}$
ASI_UDBL_ERROR_REG_WRITE	External UDB Error Register, write low	77_{16}
ASI_UDB_CONTROL_W	External UDB Control Register, write high	77_{16}
ASI_UDB_CONTROL_W	External UDB Control Register, write low	77_{16}
ASI_UDB_ERROR_W	External UDB Error Register, write high	77_{16}
ASI_UDB_ERROR_W	External UDB Error Register, write low	77_{16}
ASI_UDB_INTR_R	Incoming interrupt vector data register 0	$7F_{16}$
ASI_UDB_INTR_R	Incoming interrupt vector data register 1	$7F_{16}$
ASI_UDB_INTR_R	Incoming interrupt vector data register 2	$7F_{16}$

Table G-1 ASI registers and ASI Names—listed alphabetically *(Continued)*

ASI Name or Macro Syntax	Description	Value
ASI_UDB_INTR_W	Interrupt vector dispatch	77_{16}
ASI_UDB_INTR_W	Outgoing interrupt vector data register 0	77_{16}
ASI_UDB_INTR_W	Outgoing interrupt vector data register 1	77_{16}
ASI_UDB_INTR_W	Outgoing interrupt vector data register 2	77_{16}
ASI_UPA_CONFIG_REG	UPA configuration register	$4A_{16}$

APPENDIX **H**

Event Ordering on UltraSPARC IIi

H.1 Highlight of US-II*i* specific issues

The UltraSPARC IIi CPU meets the requirements of the SPARC V-9 and SUN4U memory models.

Some important points that may not be obvious:

- The membar instruction cannot be used to guarantee that a noncacheable store has completed to a device.

  However, a feature of the UltraSPARC IIi CPU is that explicit membar instructions can be used to guarantee that PCI activity has progressed to the primary PCI buses. However progress to the UPA64S interface cannot be guaranteed with membars.
- A single cacheable mutex semaphore should not be used to control shared access to a PCI device when shared access involves the processor and a PCI DMA master. A robust solution might use a passed token instead in a a single reader and single-writer lock exchange. This solution meets the PCI producer/consumer model.

  There is a lack of SMP-like ordering because a PCI DMA master can short-circuit the global ordering mechanism by direct peer-to-peer access to the device on its local bus.

  This could allow the PCI DMA master to issue stores to the device that jump ahead of uncompleted activity from the processor. This issue exists because of the hierarchy of buses in the PCI domain, and also because of the fact that the membar instruction cannot guarantee the completion of a noncacheable store.

- A single cacheable mutex semaphore is ideal for controlling similarly shared access to cacheable memory or the UPA64S interface, since the PCI DMA master cannot jump ahead of any globally ordered CPU activity, and SMP-like global ordering is enforced with the ordering point inside UltraSPARC IIi.
- The SUN4U architecture has no mechanism for ordering PCI PIO and DMA activity. DMA event completion is ordered with interrupts, or possibly with a cacheable semaphore as noted above.

H.2 Review of SPARC V9 load/store ordering

The SPARC V9 Architecture began with a straightforward set of "sequencing" memory barrier instructions (membars) to be used by software to guarantee that prior program order loads and/or stores would be globally ordered after future program order loads and/or stores, for a single processor.

This global order could be considered "created" when the system could guarantee that the loads and stores would eventually complete at their final destination with effects consistent with this global order.

This known global ordering of events is necessary in multi-processor systems when processors share access to common resources. The formal definition of order is more abstract than this description but this language follows the behavior of typical hardware implementations.

Complicating the issue for performance reasons, implementations typically introduce additional queues for noncacheable operations that can operate in parallel to the ordering mechanisms for cacheable operations. Requiring the membars to order both cacheable and noncacheable events was believed to create a performance problem, since some membars exist implicitly for certain memory models.

Consequently, V9 organized that the sequencing membars apply separately within the cacheable and noncacheable domains.

To order between domains, without the additional overhead of Membar #Sync, a Membar #MemIssue instruction was created.

Membar #Sync is additionally constrained to guarantee that the effects of any exceptions have been ordered.

According to V9:

"All memory reference operations appearing prior to the MEMBAR #MemIssue must have been performed before any memory operation after the MEMBAR #MemIssue may be initiated."

The word "performed" may have been purposely chosen to be nebulous!

This instruction is known as a "completion" membar, and the apparent implication was that subsequent load/stores would be stalled until prior loads were completed, and prior stores were completed to the destination (device). However, the SUN4U architecture recognized store "completion" as a possible performance problem. and relaxed the definition to mean that load/store issue would be stalled until all prior loads and stores had been globally ordered.

This global order would be preserved out to the device, which was responsible for completing them in that order. No side-effects between devices were allowed, so this model meets the overall goals.

If knowledge of store completion to the device were really necessary for some reason, perhaps because of side-effects, SUN4U requires software to issue a load to that device (into some implementation-specific address) and wait for its completion. The device is responsible for completing the effects of all prior load/stores before completing that load.

In short, the SUN4U requirement for a Membar #MemIssue is the same as that for a sequencing Membar with #StoreStore, #StoreLoad, #LoadLoad, #LoadStore all set, but with the effects applied across both cacheable and noncacheable domains.

The UltraSPARC I and II CPUs actually implement a more conservative approach to the explicitly coded sequencing Membars. The sequencing effect applies equally against cacheable and noncacheable loads and stores. (This is not true for the implicit sequencing members in the memory models).

With PSTATE.MM==TSO, UltraSPARC I and II CPUs guarantee that all stores, both cacheable and noncacheable, are ordered globally so as to complete in program order. This is described as an implicit Membar #MemIssue in the appropriate User's Manual.

With PSTATE.MM==PSO or RMO store ordering is not necessarily preserved, notably between cacheable and noncacheable stores, and between cacheable block store commits and other cacheable stores.

Note that global ordering may also be important in all memory models if noncacheable loads have side-effects.

For the noncacheable domain, the DMMU supports a bit per page mapping called the E-bit, that has the same architecturally specified effect as having a membar with all the sequencing bits set, between loads and stores. That is, a strong sequential order is created and preserved out to devices. However, the E-bit only orders load/store within the noncacheable domain.

H.2.1 Ordering load/store Activity Out To The Primary PCI bus

This activity is not a requirement of the software model, but it is a design feature that might be minimally useful in debug situations.

UltraSPARC I and II membars only guarantee that PIO stores have completed as far as the processor data bus system, not to the SBUS or any PCI bus. As noted the global order created is preserved from that point on.

Since the software model has no ordering between DMA and PIO on the PCI bus, there should not be any case of software using a membar #sync for guaranteeing some ordering of events on the PCI bus.

The SUN4U software model description states:

"There are times that it is desirable to know if an I/O access has completed...."

"Any store queue must have an address associated with it that can be read by a processor to see if previously issued stores have completed, this may be the address of a safe-to-read status or control register..."

"Code that wishes to see if the path from the processor to a device has been cleared can do so by reading the synchronization address associated with the buffer closest to the target device."

UltraSPARC IIi also does not guarantee that writes to UPA64S have completed all the way to the UPA64S interface with a membar #sync. Since UPA64S is a single master interface, no multi-master order issues exist. The software model instead uses loads to determine store completion all the way to the UPA64S internals.

APPENDIX I

Observability Bus

The UltraSPARC IIi CPU implements an observability bus to assist in bringing up the processor and its associated systems. The bus can also be used for performance monitoring and instrumentation.

I.1 Theory of Operation

I.1.1 Muxing

At any one time, one group of 15 signals out of five possible groups—75 total signals—is selected for output to the SYSADR[14:0] pins of UltraSPARC IIi. This selection is controlled by an ASR register.

Since SYSADR is used for UPA64S addresses, the observability information is not available for the two UPA clocks—eight processor clocks for the SME1430 CPU, six for the SME1040 CPU—of an UPA64S address packet, and for one more UPA clock after that—four processor clocks for the SME1430 CPU, three for the SME1040 CPU. This period is indicated by the assertion of ADR_VLD for the first four processor clocks of the period for the SME1430 CPU (the first three for the SME1040 CPU). After the twelve processor clocks have expired in the SME1430 CPU's case—nine for the SME1040 CPU's, SYSADR[14:0] can again change state every processor clock instead of being aligned to UPA clocks.

To avoid sending CPU-frequency signals to UPA64S during normal operation, program the select to choose all 1's. This selection also limits EMI by disabling the test L5CLK outputs (CPU and PCI) on UltraSPARC IIi.

The first group (group 0) is chosen to be the most useful debug group, since this is the default group selected upon POR. There is no overlap of signals between groups.

I.1.2 Dispatch Control Register

The Dispatch Control Register, ASR 0x12, enables some performance features related to instruction dispatch, and controls the output of internal signals to UltraSPARC IIi SYSADR[14:0] pins for help in chip debug and instrumentation.

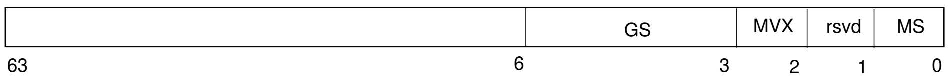


Figure I-1 Dispatch Control Register (ASR 0x12)

GS<2:0>: Group select bits. Selects the group of signals driven out on SYSADR<14:0> during cycles not used by UPA64S address packets. All unused encodings cause undefined results; zero after POR.

TABLE I-1 Group Select Bits

GS<2:0>	Group
000	0
001	1
010	2
011	3
100	4
111	ALL1

MVX: IEU.movx_enable—Controls a performance enhancement (compared to US-I) for movx instructions. If set, stops movx instruction dispatch if there is a valid load instruction in the E-stage. (performance enhancement); zero after POR.

MS: IEU.multi_scalar—Multi-Scalar Dispatch Control. If cleared, instruction dispatch is forced to a single instruction per group; zero after POR.

Recommended initialization for normal system operation is 0x3D.

I.1.3 Timing

All signals appear on the pins three stages after they are valid within UltraSPARC IIi. Each signal is buffered with a rising-edge-triggered D-flip-flop.

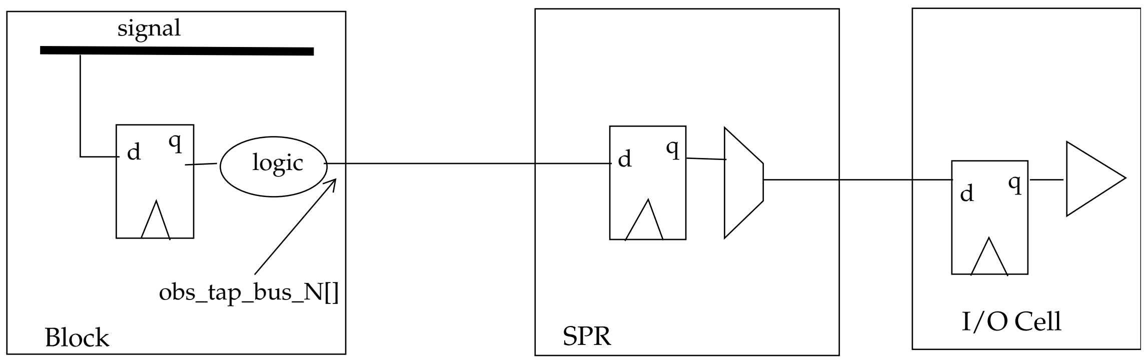


Figure I-2 Diagram of Observability Bus Logic.

I.1.4 Signal List

Groups are divided roughly into:

- Group 0: Primary pipe pins
- Group 1: Program counter
- Group 2: Prefetch unit.
- Group 3: Load-store unit, E-cache unit.
- Group 4: Special Purpose Register block signals
- ALL1: Bus is driven high at all times

I.1.4.1 Group 0

Primary pipeline signals (default group)

- obs_tap_bus_0[2:0]= num_complete = f(tr.trctrl.trpc.trap_*_ins_comp_w).

  The number of instructions completed in W, from zero through four inclusive. Help instructions are counted only once, but they differ in the exact cycle that gets counted because of the way the valid bits behave for different instructions. For example, CASA is counted on W1 of the help==00 cycle, while MULX is counted on W1 of the help==11 cycle.

- obs_tap_bus_0[4:3] = ieu_dispatched_g[3:0] compressed to 2 bits

  The number of instructions dispatched into the pipeline by G-logic.

0==no instructions dispatched
0x1 == one instruction dispatched
0x2 == two instructions dispatched
0x3 == three or four instructions dispatched.

- obs_tap_bus_0[5]= lsu_stall_v4_e

  Stall the e-stage of the pipe when an instruction requires data from an earlier load operation that is not yet available. Can happen due to D$ miss, read-after-write hazard, sign extension on a D$ hit, load buffer not empty, etc.

- obs_tap_bus_0[6]= flop(tr_microtrap_n3 | ieu_flush_n3)

  Indicates a flush or microtrap is being taken.
  obs_tap_bus_0[6] and obs_tap_bus_0[8] should not be active together and should always be followed by bit 7 going active two to many cycles later before either go active again. Both should be single cycle pulses.

- obs_tap_bus_0[7]= flop(ieu_done || ieu_retry)

  Indicates that trap logic is delivering a PC (and NPC for retries) from which to begin fetching after POR, traps, DONE/RETRY inst flushes, microtraps, etc.

- obs_tap_bus_0[8]= flop(ieu_traptaken_n3)

  The trap unit has determined that an N3 instruction should trap, and signals the pipeline to take the trap.
  obs_tap_bus_0[6] and obs_tap_bus_0[8] should not be active together and should always be followed by bit 7 going active 2 to many cycles later before either go active again. Both should be single cycle pulses.

- obs_tap_bus_0[9]= finish_fpop

  A floating point operation has come off the queue.

```
(`FGC.c_f1_write[0] | fdiv_finish)
```

- obs_tap_bus_0[10]= finish_load (NEEDS FIX IN RTL--LOGIC IN EX)

  A floating point operation has come off the queue

- obs_tap_bus_0[11]= pdu_bad_pred_c

  This C-stage signal is asserted when the direction of a conditional branch has been mispredicted or the target address of a register-indirect jump (JMPL or RETURN) has been mispredicted.

  Note: obs_tap_bus_2[5] (pdu_br_resol_c) should be asserted at the same time.

- obs_tap_bus_0[14:12]= E$ arbitration

  // ecache fills or ownership etag/edata writes

  ((dxfsm_ecache_req & ~dxfsm_ecache_busy) ? 3'd1 : 3'd0) |

  // copybacks or invalidates

```
((snp_ecache_req & ~snp_ecache_busy) ? 3'd2 : 3'd0) |
// writebacks or block stores
((trfsm_ecache_req & ~trfsm_ecache_busy) ? 3'd3 : 3'd0) |
// data back for noncacheable loads or the sdb data transfer nc stores
((nc_ecache_req & ~nc_ecache_busy) ? 3'd4 : 3'd0) |
// noncacheable or cacheable loads/bloads, asi stores to sdb/ecache
(ldb_win ? 3'd5 : 3'd0) |
// noncacheable or cacheable stores/bstores, asi loads to sdb/ecache
(stb_win ? 3'd6 : 3'd0) |
// tag checks for stb
(sttag_win ? 3'd7 : 3'd0);
```

I.1.4.2 Group 1

Program counter

- obs_tap_bus_1[11:0]= pc[13:2].

  These are bits [13:2] (the word address) of the D-stage “fetch PC”. (LSB of the virtual page number + page offset).

  RTL use: In the D-stage, this PC (bits [43:13]) is being translated by the ITLB. It is also the PC that will be enqueued in the GPCQ (G-stage PC Queue) in the next cycle (when the associated instructions are enqueued in the IBuffer), if this fetch starts a new PC segment.
- obs_tap_bus_1[12]= pfc_utlb_miss
- This D-stage signal is asserted when the fetch PC crosses a page boundary (e.g. by jumping to a different page), the prefetcher stalls 1 cycle to wait for the ITLB translation.
- obs_tap_bus_1[13]= function of (pfc_va_valid, pfc_cancel_itlb)
- When this signal is asserted in the D2 stage, the results (hit/miss/exception and the physical address) of the ITLB translation performed the previous cycle (D stage) are valid and used.
- obs_tap_bus_1[14]= function of (pfc_imu_exc, pfc_imu_miss)

  This signal is asserted in the D2 stage (when a uTLB miss has occurred in D, forcing the prefetcher to stall for the ITLB translation) if the VA translation has caused an exception (caused an ITLB miss or an ITLB access exception, or the

VA is illegal--in the “hole”). This signal is already qualified by the “cancel” signal, pdu_cancel_itlbt, so that it will not be asserted if the translation will not actually be needed.

I.1.4.3 Group 2

Prefetch unit, caches

- obs_tap_bus_2[1:0] = pdu_i*_valid (compressed to 2 bits)

  Encoded count of number of valid instructions in the IBuffer.

  0==no instructions dispatched, 0x1 == one instruction dispatched, 0x2 == two instructions dispatched, 0x3 == three or four instructions dispatched.

- obs_tap_bus_2[2] = fetch_stall = pfc_ignore_fetch || ibcm_full || gpcq_qfull

  If this D-stage signal is asserted, no instructions will be enqueued in the IBuffer next cycle. It will be asserted if the IBuffer or GPCQ is full, or for prefetch stall events: NFA-PC mismatches, SP mispredictions, uTLB misses, branch mispredictions, or cache stalls (for E-cache accesses, snoops, ASI accesses, or flushes).

- obs_tap_bus_2[3] = pfc_non_fetch

  Asserted when the instruction prefetcher is stalled because the I-cache is busy (for an E-cache fetch, a snoop, ASI access, or flush).

- obs_tap_bus_2[4] = pdu_br_taken_c

  When obs_tap_bus_2[5] (pdu_br_resol_c) is asserted (i.e. a branch is resolved), this C-stage signal is asserted when a conditional branch (Bicc, BPcc, FBfcc, FBPfcc) is taken.

- obs_tap_bus_2[5] = pdu_br_resol_c

  Asserted when a DCTI (Bicc, BPcc, FBfcc, FBPfcc, JMPL, RETURN) reaches the C stage.

  Note: obs_tap_bus_0[11] (pdu_bad_pred_c) should only be asserted when this signal is asserted. obs_tap_bus_2[4] is only valid when this signal is asserted.

- obs_tap_bus_2[6] = pc.pcgen_ctl.pfc_spmiss_d

  This D-stage signal is asserted when a “Set misprediction” (SP miss) occurs (that is, when the instructions were fetched from the wrong bank of the I-cache, so the prefetcher must redo the fetch). This should cause the prefetcher to stall for 2 cycles.

  Note: as a result, obs_tap_bus_2[2] (fetch stall) should be asserted in the same cycle.

- obs_tap_bus_2[7] = imux_pcmiss_d1_f

This D-stage signal is asserted when there is an NFA-PC mismatch (that is, when the "next fetch address" from the NFRAM, used for the F-stage I-cache fetch, mismatches with the actual fetch PC, so the prefetcher must redo the fetch). This is sometimes referred to as a "PC miss". The prefetcher should stall for 2 cycles.

Note: as a result, obs_tap_bus_2[2] (fetch stall) should be asserted in the same cycle.

- obs_tap_bus_2[8] = ibd_pcrel_taken_d

  D-stage decode signal for the instructions from the current I-cache (or E-cache) fetch. Indicates that there is a PC-relative branch in the current fetch that is predicted-taken.

- obs_tap_bus_2[9] = ibd_regbr_d

  D-stage decode signal for the instructions from the current I-cache (or E-cache) fetch. Indicates that there is a register-indirect jump (JMPL or RETURN) in the current fetch.

- obs_tap_bus_2[10] = (copy of obs_tap_bus_2[0])
- obs_tap_bus_2[11] = iblock.icc_update_icache

  This signal is asserted when the I-cache or NFRAM should be updated for a cache fill (it is a component of the RAM write-enables).

- obs_tap_bus_2[12] = imu_stop

  IMU has encountered an exception, and will be suspended until told by the pipeline that the exception has been cleared by the instruction being annulled or flushed as it goes down the pipe, or reaching W stage and causing a trap. The imu_stop is cleared whether the instruction causes a trap or not. If imu_stop is left high and the CPU is hung, check for PDU waiting on a request to the ECU. Otherwise, look for cases of the exception instruction getting annulled or flushed without notifying the IMU.

- obs_tap_bus_2[13]= write D$

  Active when any byte of D$ is being modified, either from a store or D$ fill. For D$ misses, the D$ and D$ tags are written assuming that the data is a hit in the E$. If there is an E$ miss, the D$ will be updated properly when the data for the E$ miss is returned from the system.

- obs_tap_bus_2[14]= lsu_tag2_we

  D$ tag write enable.

I.1.4.4 Group 3

Load-store unit, E$ unit

- obs_tap_bus_3[3:0]= Snoop information

{ecu_pd_snoop_req, pdu_busy, ecu_ls_snoop_req, lsu_ec_dcache_busy};

- obs_tap_bus_3[7:4]= E$ request/cancel information

  If there is a read and it is not one of the following, it is the PDU (cacheable or noncacheable). Block loads and stores that hit the ecache will be distinctive by their OE/WE pattern (incrementing addresses).

  {ecu_ls_cancel_all, ecu_pd_cancel_all, ecu_ls_cancel_tag, ecu_ls_clear_tag};

- obs_tap_bus_3[8]= enq_n1

  Load buffer gets an entry enqueued. Often an n1-stage load cannot return data and must be put on the load buffer.

- obs_tap_bus_3[9]= ldb_zero_entries

  The load buffer is empty.

- obs_tap_bus_3[10]= raw_hit_target_n1

  The D$ access has hit. This is a "raw" signal and is based on the current state of the D$. It is possible that older loads in the Load Buffer can "adjust" the load/store in n1-stage into either a hit or miss based on how these older loads will change the state of the D$ by bringing in new data/overwriting old data.

- obs_tap_bus_3[11]= lsu_use_other

  lsu_use_other indicates from where load data is returning. If asserted, data is coming from the "other" bus. If deasserted, data is coming directly from the D$. The "other" bus transfers data for:

  - D$ misses
  - NC loads
  - diagnostic loads (load alternates) of external resources (e.g. SDB registers, E$ data RAM, E$ tag RAM)
  - loads (again, load alternates) of internal resources (e.g. I$, DMMU, IMMU, D$, ECU internal registers, etc.).

  In addition, it also carries data on D$ hits for signed loads (ldsb/ldsba, ldsh/ldsha, ldsw/ldswa) one cycle delayed. If a subsequent load is attempting to return data in the cycle following the signed load's D$ hit, it is forced to use the "other" bus and to be delayed one cycle as well (this scenario is often referred to as "delayed return mode").

- obs_tap_bus_3[12]= lsu_stb_dec_count

  An entry is dequeueing from the store buffer. This signal is asserted the cycle after the Store Buffer valid bit is deasserted. For writes to the E$, this is the cycle that the address is being driven from UltraSPARC IIi to the E$ RAMs.

- obs_tap_bus_3[13]= stb_block_ldb_ec_req

Store buffer gets priority over the load buffer for E$ request signals.

No Load requests to the E$ can be made in this cycle, because the Store Buffer has assumed priority to "drain" as it has hit a "high watermark" in the number of entries it contains.

- obs_tap_bus_3[14]= sab_addr_valid[0]

  Valid bit for store buffer entry 0. (Store buffer is not empty.)

I.1.4.5 Group 4

Information from EX on CWP state and changes.

- obs_tap_bus_4[7:0]= spr_cwpread_g[7:0]
- obs_tap_bus_4[10:8]= sprcntl_cwp_muxsel_g[2:0]
- obs_tap_bus_4[14:11] {sprcntl_cwpchange_e, sprcntl_cwpchange_c, sprcntl_cwpchange_n1, sprcntl_cwpchange_n3}

I.1.4.6 ALL1

When this group is chosen the observability bus is driven high at all times. This reduces the power consumption of UltraSPARC IIi since the pins are not toggling. The CPU and PCI test L5CLK's are also disabled.

Note – The ALL1 group is not the default group. If this feature is required in the system level environment the boot/initialization code must set GS bits accordingly.

I.1.5 Other UltraSPARC IIi Debug Features

In addition to the observability bus, the default value of the ECAD (address to the data SRAMS) is pdu_pa[21:4], which is the PDU's prefetch address

APPENDIX **J**

List of Compatibility Notes

The following text is a list of the comp[atibility notes that appear through the body of this manual. The page number for the original compatibility note in the body of the manual appears at the end of each entry in this list

Note 1: A read of any addresses labelled "Reserved" above returns zeros, and writes have no effect. 50

Note 2: If Configuration cycles are generated with compressed (E-bit==0) byte or halfword stores, or with random byte enable patterns using the PSTORE instruction, UltraSPARC IIi does not guarantee that AD[1:0] points to the first byte with a BE asserted.

Also, while not addressed by the PCI 2.1 specification UltraSPARC IIi can generate multi-databeat configuration reads and writes. 83

Note 3: There are no time out errors during table walk for the UltraSPARC IIi IOM. 102

Note 4: Bits in the DMA UE AFSR/AFAR are set, and the PA of the TTE entry is saved on Invalid, Protection (IOM miss), and TTE UE errors. This should aid debugging of software errors. If the Protection error had an IOM hit, the translated PA from the IOM is saved instead of the PA of the TTE entry. This may occur if a prior DMA read caused the IOM entry to be installed. 102

Note 5: Prior PCI-based UltraSPARC systems implemented a true LRU scheme. 103

Note 6: The IGN on UltraSPARC IIi is not programmable, and fixed to 0x1F. 108

Note 7: UltraSPARC IIi does not send interrupts to any devices. A write to these registers has no effect. 119

Note 8: UltraSPARC IIi does not send interrupts to any devices. A read of this register always returns zeros. 120

Note 9: UltraSPARC IIi only supports the interrupt data that were present in prior UltraSPARC-based systems; that is, bits 10:0 (INR) of ASI_SDB_INTR(0). All other bits are read as 0. 120

Note 10: Prior UltraSPARCs may have provided the first two registers at the same time. If code depends upon this unsupported behavior it must be modified for UltraSPARC IIi. 168

Note 11: When the processor is reset, UPA64S, PCI, and APB are also reset. 173

Note 12: Referenced and Modified bits are maintained by software. The Global, Privileged, and Writable fields replace the 3-bit ACC field of the SPARC-V8 Reference MMU Page Translation Entry. 200

Note 13: The UltraSPARC IIi MMU performs no hardware table walking. The MMU hardware never directly reads or writes to the TSB. 203

Note 14: The single context register of the SPARC-V8 Reference MMU has been replaced in UltraSPARC IIi by the three context registers shown in Figures 15-4, 15-5, and 15-6. 215

Note 15: In UltraSPARC IIi the virtual address is longer than the physical address; thus, there is no need to use multiple ASIs to fill in the high-order physical address bits, as is done in SPARC-V8 machines. 226

Note 16: UltraSPARC automatically caused the reset through the UPA. The UltraSPARC IIi CPU currently does not cause an automatic reset. 232

Note 17: If an E-cache data parity error occurs during a write-back, uncorrectable ECC is not forced to memory. However, the error information is logged in the AFSR and a disrupting data_access_error trap is generated. 236

Note 18: If PER is disabled, UltraSPARC IIi does not set DPE if it detects a parity error on PIO reads. This is inconsistent with the PCI 2.1 spec. 237

Note 19: If PER is disabled, UltraSPARC IIi does not set DPE if it detects a parity error on DMA writes. This is inconsistent with the PCI 2.1 spec. 238

Note 20: A new feature for UltraSPARC IIi, is that the VA of the offending DMA access is logged in the PCI DMA UE AFSR and AFAR, with the a bit set for identification as a DMA translation error. 239

Note 21: UltraSPARC IIi does not Target Abort on a a parity error resulting from a DMA read of E-cache. UltraSPARC caused a UE at the receiver of the data. Currently it is only reported with the same priority/trap as WP (but CP bit set). 246

Note 22: UltraSPARC IIi causes a Deferred Trap similarly to UltraSPARC for ETS, without a system reset. Software can determine if a system reset is necessary. 246

Note 23: The SDB name is inherited from UltraSPARC. It logs information about memory errors caused by the CPU core. Only the SDBH register is used. Current Solaris software interrogates if SDBL is non-zero, and ORs in a 1 to the logged pa[3] (which is always zero on UltraSPARC, but valid on UltraSPARC IIi). 247

Note 24: There is no Wakeup Reset support for power management, unlike that in prior UltraSPARC-based systems. 255

Note 25: Prior UltraSPARC-based systems used other hardware and programming models to control the UPA and memory interfaces. 266

Note 26: APB has a similar additional state for each of its PCI busses. See the APB User's Manual for details. 283

Note 27: This device ID is different from that of prior PCI-based UltraSPARC systems. 291

Note 28: A value of 0 means there is no latency timeout. 293

Note 29: ERR and ERRSTS are not present in prior PCI-based UltraSPARC systems. 296

Note 30: Unlike prior PCI-based UltraSPARC systems, UltraSPARC IIi arbitrates between IOMMU CSR access and DMA access. This property may allow software more flexibility. 299

Note 31: The Used bit does not exist in prior PCI-based UltraSPARC systems, and is used by the pseudo-LRU replacement algorithm. 299

Note 32: The IGN on UltraSPARC IIi is not programmable for the Partial Interrupt Mapping Registers, and is fixed to 0x1f. 301

Note 33: There is no RECEIVED state for DMA CE, DMA UE, or PCI Error Interrupts. They cause their interrupt FSMs to go from the IDLE to the PENDING state directly, when present and enabled. 302

Note 34: Note the "Graphics Int State" and Expansion UPA64S Int State" bits are moved from bits 38 and 39 (position in prior UltraSPARC systems) to bits 34 and 35 respectively. 309

Note 35: The UltraSPARC IIi PCI bus is hardwired to Bus
Number == 0 312

Note 36: UltraSPARC IIi aliases Functions 1-7 of its PCI Configuration space to its Function 0 PCI Configuration space. (Bus 0, Device 0). The PCI specification requires that zeros be returned and stores ignored. Since this address space is only accessible toUltraSPARC IIi PIO instructions, specifically boot PROM code, this aliasing should not be problematic because the boot PROM should never reference the UltraSPARC IIi Function 1-7 addresses (see Type 0 Configuration Address Mapping on page 312 for the address decode scheme). 313

Note 37: Unlike prior PCI-based UltraSPARC systems, UltraSPARC IIi does not use bit 31 of the PCI address for outgoing memory transactions, or bit 17 for outgoing IO transactions. APB also similarly preserves bits 31 and 17. 314

Note 38: Unlike prior PCI-based UltraSPARC systems, Pass-through does not zero PCI_Addr[31] 315

Note 39: Prior PCI-based UltraSPARC systems used PCI_Addr<40>, but note that [40:34] are all 1's for UPA64S addresses. 316

Note 40: A PCI DMA UE interrupt is generated whenever a primary DMA UE or Translation Error bit is set, even if by a CSR write. Ensure that software clears the AFSR before clearing the interrupt state and re-enabling the PCI Error Interrupt. (This behavior is similar to that of the ECU AFSR) 317

Note 41: This feature is absent in prior PCI-based UltraSPARC systems but should be compatible with existing Solaris code. 318

Note 42: A DMA CE interrupt is generated whenever a primary DMA CE bit is set, even if by a CSR write. Ensure that software clears the AFSR before it clears the interrupt state and re-enables the PCI Error Interrupt. (This behavior is similar to that of the ECU AFSR). 319

Note 43: Because of the smaller external cache data and tag, some adjustments are made to these diagnostic accesses. 380

APPENDIX **K**

Errata

K.1 Overview.

This document contains a list of errata for 1.2 and above of the UltraSPARC IIi CPU.

K.2 Errata Created by UltraSPARC-I

Erratum 32: Load from ITLB or DTLB may return wrong data if the load is after a store instruction to ITLB or DTLB that traps

The following is required to occur:

- Store to ASIs ASI_ITLB_DATA_ACCESS_REG or ASI_DTLB_DATA_ACESS_REG (ITLB or DTLB entries) traps.
- Load from ASIs ASI_ITLB_DATA_ACCESS_REG or ASI_DTLB_DATA_ACESS_REG (ITLB or DTLB entries).
- No intervening store instructions between the above Store and Load.

For example:

```
stx %reg,[..]ASI          ;if this instruction traps for some reason
                          ASI for ITLB 0x55 and for DTLB 0x5d

....                      ;the instructions dispatched following store

space                     ;does not contain any st or st to alternate
                          instruction

                          ;Reads TLB entry ASIs 0x55, 0x56 (for ITLB
ldx [..]ASI %reg          ;ASI  0x5d, 0x5e (for DTLB)
```

In the IMU/DMU, the address of the internal register to be written by a store is latched after the store is dispatched. A wait state is entered until the time the data is actually written. If this instruction traps, the control logic does not reset and remain in this wait state. A subsequent load from TLB entries can be corrupted by this wait state, resulting in the use of the internal address associated with the prior store instead of that from the load. However, this wait state is cleared by any store instruction.

Hence the problem does not exist if a store is executed between the trapping store and the load.

Software workaround: Add a Store instruction to any address space before loads from ITLB or DTLB, if none already exists.

Erratum 45: DONE/RETRY/SAVED/RESTORED with illegal fcn field executed in nonprivileged mode take privileged_opcode trap rather than illegal_instruction trap.

The following instruction conditions generate a privileged_opcode trap rather than the specified illegal_instruction trap.

```
DONE   for fcn = 2..31 executed in nonprivileged mode
RETRY  for fcn = 2..31 executed in nonprivileged mode
SAVED  for fcn = 2..31 executed in nonprivileged mode
RESTOREDfor fcn = 2..31 executed in nonprivileged mode
```

Software workaround: The opcode can be recognized by software to emulate the proper illegal_instruction behavior. This can be done with SPARC code in the privileged_opcode trap handler that does the following:

```
PRIVILEGED_OPCODE_HANDLER:
        rdpr    %tpc, %g1
        ld      [%g1], %g2
        setx    0xc1f80000, %g3, %g4
        and     %g4, %g2, %g4   ! %g4 has op/op3 of trapping instr.
```

```
        setx    0x3e000000, %g3, %g6
        and     %g6, %g2, %g6
        srl     %g6,  25, %g6   ! %g6 has fcn of trapping instr.
check_illegal_saved_restored:
        setx    0x81880000, %g3, %g5
        subcc   %g4, %g5, %g0   ! saved/restored opcode?
        bne     check_illegal_done_retry
        subcc   %g6, 2, %g0     ! illegal fcn value?
        bge     ILLEGAL_HANDLER
        nop
check_illegal_done_retry:
        setx    0x81f00000, %g3, %g5
        subcc   %g4, %g5, %g0   ! done/retry opcode?
        bne     not_illegal
        subcc   %g6, 2, %g0     ! illegal fcn value?
        bge      ILLEGAL_HANDLER
        nop
not_illegal:
        <handle privileged_opcode exception as desired here>
```

Erratum 47: JMPL instruction at boundary of Virtual address hole sign-extends %rd.

Virtual addresses between:

`0x0000 0800 0000 0000 and 0XFFFF F7FF FFFF FFFF`

inclusive, are termed out of range. This range is referred to as the Virtual address hole and is described in Section 4.2, *Virtual Address Translation* on page 23; also see Section 14.1.7, *44-bit Virtual Address Space* on page 178.

The following instruction sequence causes %rd to be loaded with the wrong value:

```
pc = 0x000007FF.FFFFFFFC jmpl address, %rd
pc = 0x00000800.00000000
```

The %rd is saved as: `0xFFFF F800 0000 0000`, when it should be the first address in the Virtual address hole: `0x0000 0800 0000 0000`.

The failure would be that an erroneous jmpl at the boundary (which should trap if the correct return address were used) would create a valid instead of invalid return address. This valid return address would not trap as a "VA hole" PC.

Software workaround: US-I errata require the OS to not map the 4 GB of instruction space immediately above and below the VA hole, so the OS would not map the following 4 GB ranges:

`lower range: 0x0000 07FF 0000 0000 to 0x0000 07FF FFFF FFFF`

```
upper range: 0xFFFF F800 0000 0000 to 0xFFFF F800 FFFF FFFF
```

Since the instruction address at the boundary is never mapped, a valid instruction is never executed at that PC.

Erratum 48: DONE/RETRY with TL=0 causes a privileged rather than an illegal instruction trap.

The SPARC Architecture Manual, Version 9 says an illegal instruction trap should be taken. Instead, a privileged trap is taken.

Erratum 49: ASI's 0x5c/5d/5e all cause ft[2] in the DMMU SFSR to be set according to the tlb entry.

The UltraSPARC I/II User's Manual says that the ft[2] bit of the D-MMU Synchronous Fault Status Register (loaded on traps) is set for Atomics (including 128-bit atomic load) to page marked uncacheable, and that the bit is zero for internal ASI accesses, except for atomics to DTLB_DATA_ACCESS_REG (0x5D), which update according to the TLB entry accessed. (See Section 15.4.4, *Data_access_exception Trap* on page 204 and *Table 15-13* on page 216).

The correction to the documentation is that all ASIs which access the D-MMU tlb have the same behavior, that is:

```
0x5C    ASI_DTLB_DATA_IN_REG
0x5D    ASI_DTLB_DATA_ACCESS_REG
0x5E    ASI_DTLB_TAG_READ_REG
```

For instance,

swapa [%g0] 0x5e, %g0

traps with ft[3:0] = 1000, if the mapping for VA==0x0 has cp==1 and cv==1.

Erratum 50: RDPR of TPC, TNPC, or TSTATE may not bypass correctly into arithmetic instructions that create condition codes, causing incorrect V/C bypass/use. (Z and N are apparently always correct)

The discovered failing instruction sequence is:

```
rdpr %tpc, %i0
subcc %i0, %g2, %i3
```

The 65th bit of the ALU used in the 2nd instruction can be incorrect. This should only affect the setting of the V and C flags by that instruction. It may also affect an integer divide that uses the result of the rdpr.

The code above might be used when software is checking for a range of PC values and uses the V or C flag to do a less-than, greater-than comparison. The problem may exist for rdpr's of other trap state.

The problem occurs on instructions that use the first-level shortloop into the diad 65 bit ALU on operands whose results are generated from the iexe_aludp1_aluout_65_e bus.

On second level and later conflicts the 65th bit was stripped off and shortlooped back in as zero. Only the first level shortloop allows a one on bit 64 to be shortlooped back into a following instruction.

The 65th bit can only be one either when information is read in from the trap_sr_e busses and sign extended into the 65th bit, or for a shift operation.

There is a family of failures that can occur on any instruction following and using the results of a preceding instructions usage of the trap_sr_e results bus.

The full range of rdpr/rdasr that could be of interest can be examined:
for non-zero bit 63. (fp stuff excluded)

rdpr of:

> TPC, TNPC, TSTATE, TT, TICK, TBA, PSTATE, TL, PIL, CWP, CANSAVE, CANRESTORE, CLEANWIN, OTHERWIN, WSTATE, and VER.

and rdasr of:

> Y_REG, COND_CODE_REG, ASI_REG, TICK_REG, PERF_CONTROL_REG, PERF_COUNTER, DISPATCH_CONTROL_REG, GRAPHIC_STATUS_REG, SOFTINT_REG, TICK_CMPR_REG

Since the MSB needs to be 1, not all of the above registers can cause the error (if they have bit 63 defined to be zero always), so apparently only

> rdpr of TPC, TNPC, TSTATE, TICK, and rdasr of TICK_REG, and PERF_COUNTER

can cause this error. It appears further that only reads from trap state are involved, that is, TPC, TNPC, or TSTATE.

Software workaround: Inhibit use of this bypass path by feeding the result of the rdpr through another operation before doing an instruction on it that sets condition codes or integer divides. That is, the example at the top could become:

```
rdpr %tpc, %i0
mov %i0,%i0
subcc %i0, %g2, %i3
```

Erratum 51: IMU miss, with mispredicted CTI and delayed issue of delay slot, can cause instruction issue to stop.

US-I, II, and IIi can stop issuing instructions (but be interruptible by XIR, and possibly other enabled trap conditions) due to a condition created, in one case, by this instruction sequence in an older Solaris interrupt trap handler:

`STXA`	using ASI in the range 0x46-0x5f, 0x76 or 0x77 (possibly any store) <0-n instructions. Maximum n is unknown.>
`JMPL`	
`MEMBAR #Sync`	

Apparently, the deadlock is most easily caused if the delay slot of the JMPL is a MEMBAR #Sync, or any instruction that synchronizes on the load or store buffers being empty. It appears that a delayed issue of the delay slot instruction is required, with the delay being probably 8 cycles or more after the CTI instruction.

The relevant part of all this is just causing the delay slot instruction issue to be delayed, in the presence of a mispredicted branch (the JMPL is mispredicted the first time it is installed into the I-cache). So there are more scenarios possible than those described.

The "delayed issue" requirement apparently does not include "delayed due to fetching the delay slot instruction".

It may also be possible to create the condition if the JMPL is replaced by other control transfer instructions, for example, CALL or RETURN or possibly any CTI. However, they must be mispredicted. There are a number of other conditions related to hits on I-cache state that are also apparently required.

The easiest way to get an IMU miss, for typical code execution scenarios, is when using a predicted VA from the Return Address Stack (RAS). This appears to be why the JMPL sequence exposes the problem. Also, it appears that the predicted information for the target may need to be a pc-relative branch, and that the predicted information may need to be marked invalid in the I-cache predecode RAM.

Note that the VAs in question are all predicted, and the combination of the predicted VA from the RAS, and a predicted branch displacement may result in a VA that is never mapped, rather than just temporarily in the IMU.

Since it is possible to trap out of this deadlock, it can only be detected as a performance loss, except when pstate.ie==0 and timer interrupts cannot occur. (for instance, in trap handlers).

Software workaround: Any code that

- turns off pstate, that is, disabling timer interrupts, or
- is very performance sensitive and which carries the possibility of mispredicted JMPL or branches with delay slots whose issue can be delayed (there are many cases; note that "delayed because not fetched yet" must also be included)

must guarantee:

No IMU miss on any predicted path for the prefetch PCs. This must be true for all behaviors of the RAS and the NFRAM, in generating predicted PCs, which may not reflect real execution.

For the OS, this amounts to requiring the RAS be initialized with CALLs to its known IMU-hitting VA space, specifically, CALLs that have return PCs 4 G-bytes away from the boundary of its IMU-hit VA space. The 4 G-bytes requirement helps ensure that predicted JMP targets are still within the IMU-hitting VA space.

Note that CALL instructions push onto the RAS before being issued, so it is possible for unexpected VAs to appear on the RAS, owing to predicted CALLs pointing to old I-cache pre-decode information.

Note that user code can still cause this IMU stop scenario. Since it is interruptible, execution resumes at the next interrupt (or, in the worst case, at the time slice), and the stop is not detected.

Erratum 53: Little-endian enabled integer LDD/STD do not register swap.

This applies to pages with the IE bit set in the TSB entry for that page, or to ldda/stda used with any of the "LITTLE" ASIs... that is:

ASI_AS_IF_USER_PRIMARY_LITTLE
ASI_AS_IF_USER_SECONDARY_LITTLE
ASI_NUCLEUS_LITTLE
ASI_PRIMARY_LITTLE
ASI_SECONDARY_LITTLE
ASI_SECONDARY_NOFAULT_LITTLE

The V9 architecture requirement is given in Section 6.3.1.22 “Little-Endian Addressing Convention” on page 69-70 of *The SPARC Architecture Manual, Version 9*:

doubleword or extended word: For the deprecated integer load/store double instructions (LDD/STD), two little-endian words are accessed. The word at the address specified in the instruction + 4 corresponds to the even register specified in the instruction. The word at the address specified in the instruction corresponds to the following odd-numbered register.

Instead of this requirement, US-I, II and IIi link the word address specified in the instruction to the even register, always. The word address plus 4 is linked to the odd register always.

Note that sections A27 and A53 of the of the *The SPARC Architecture Manual, Version 9* describe the LDD/STD instructions as behaving similarly. Use the descriptions in section 6.3.1.2.2 of the Architecture manual for the exclusion for little-endian behavior.

Erratum 58: Clarification on manipulation of the Used bit in iTLB and dTLB.

The dTLB and iTLB support a replacement algorithm based upon three status bits in each TLB entry, Locked, Used, and Valid. When software does a write of the I-TLB or the D-TLB Data In registers, using ASI 0x54 or 0x5C, the entry used for the write is selected depending upon the state of these bits.

The Valid bit is set when the TLB entry has valid data in it. The Used bit is set to 1 each time the entry is accessed for a translation. The Locked bit is set to lock the entry in the TLB.

Ordinarily the exact behavior of the Used bits is not of interest to software, and is only of interest in understanding the hardware. When there are no freely-available TLB entries (that is, with Valid == 0 or Used == 0), the hardware initiates a "Uclear" command to clear all the used bits in the TLB.

There is a case to consider in "lock-step" applications. An attempt by software to set the Used bit to 1 could result in an indeterminate value in this bit. This could cause "lock-step" CPUs to get out of sync., since the Used bit manipulations have to be exactly the same for two CPUs to operate identically.

Software should never write Used==1 (bit 0 of the Diag field, which is bit 41 of the Data In register), using Data In writes. This is because if a clear of the Used bits is being done in the same cycle by hardware, the results are indeterminate.

It appears there is no such constraint on Data Access writes.

The exact selection algorithm is:

```
if (there exists x : x.v == 0) {
  first such x;
} elseif (there exists y: y.u == 0 && y.l == 0) {
  first such y;
} elseif (there exists z: z.l == 0) {
  first such z;
} else {
  entry 63;
}
```

A hardware "uclear", a clear of all the Used bits, can be triggered in just about any TLB cycle, even if the TLB is doing a write, for example. A uclear is triggered when: all entries are valid, and none have Lock==0 and Used==0,

So, for example, locking an entry that never gets the Used bit set, does not inhibit the u-clear operation

Erratum 59: Clarification on use of CP==1, CV==0 (for instance, ASI_PHYS_USE_EC) to bypass the D-cache

The D-cache can return stale data if CP==1, CV==0 is used to bypass the cache, after use of CP==1 and CV==1, for loads and stores to a particular address.

The D-cache should be flushed after mixing use of any CP/CV settings for a physical address, including cacheable (DRAM) and noncacheable (IO) physical addresses. The term “noncacheable” in the user’s manual does not refer to “non-D-cacheable”. The term “virtually noncacheable” does refer to the “non-D-cacheable” CP==1, CV==0 case.

CP==1, CV==1: Cacheable, Virtually-cacheable

CP==1, CV==0: Cacheable, Virtually-noncacheable

CP==0, CV==1: Not Used

The D-cache can return stale data if CP==1, CV==0 is used to bypass the cache, after use of CP==1 and CV==1, for loads and stores to a particular address.

The D-cache should be flushed after mixing use of any CP/CV settings for a physical address, including cacheable (DRAM) and noncacheable (IO) physical addresses. The term “noncacheable” in the user’s manual does not refer to “non-D-cacheable”. The term “virtually noncacheable” does refer to the “non-D-cacheable” CP==1, CV==0 case.

CP==1, CV==1: Cacheable, Virtually-cacheable

CP==1, CV==0: Cacheable, Virtually-noncacheable

CP==0, CV==1: Not Used

CP==0, CV==0: Noncacheable

Only two entries in the D-cache need be flushed for each physical address {VA[13]==0,PA[12:0]} and {VA[13]==1,PA[12:0]}.

Q: When I do a load with a physical address, using ASI=0x14 (ASI_PHYS_USE_EC), causing CP==1 and CV==0, and the address hits in the D-cache, does the data come from the D-cache instead of from the E-cache

A: Note that the manual has a caveat that is similar to this case:

If CP==0 and CV==0, which indicates a “noncacheable” access, and the address is in the D-cache, data can be returned from the D-cache. The manual warns that the address should be flushed from the D-cache before changing its mapping.

Similarly, if CP==1, and CV==0, and the data is in the D-cache, data may be returned from the D-cache. However there are corner cases where it may not be returned.

For instance, with ASI_PHYS_USE_EC, the physical PA[13] is used to index the D-cache, where VA[13] would ordinarily be used. So the data might not be correctly returned if the real data were in VA[13]==0, but PA[13]==1. Ordinarily the rest of the PA bits will show a difference, so there is a miss in the D-cache, and go to the E-cache correctly. This takes advantage of knowing that a valid PA can only exist in one VA[13] mapping at a time in the D-cache. Note that this depends on how the addresses were mapped earlier, when the line was installed in the D-cache.CP==0, CV==0: Noncacheable

Only two entries in the D-cache need be flushed for each physical address {VA[13]==0,PA[12:0]} and {VA[13]==1,PA[12:0]}.

When a load with a physical address occurs, using ASI=0x14 (ASI_PHYS_USE_EC), causing CP==1 and CV==0, and the address hits in the D-cache, the data can come from the D-cache instead of from the E-cache .

If CP==0 and CV==0, which indicates a "noncacheable" access, and the address is in the D-cache, data can be returned from the D-cache. The manual warns that the address should be flushed from the D-cache before changing its mapping.

Similarly, if CP==1, and CV==0, and the data is in the D-cache, data may be returned from the D-cache. However there are corner cases where it may not be returned.

For instance, with ASI_PHYS_USE_EC, the physical PA[13] is used to index the D-cache, where VA[13] would ordinarily be used. So the data might not be correctly returned if the real data were in VA[13]==0, but PA[13]==1. Ordinarily the rest of the PA bits will show a difference, so there is a miss in the D-cache, and a correct reference to the E-cache. This takes advantage of knowing that a valid PA can only exist in one VA[13] mapping at a time in the D-cache. Note that this depends on how the addresses were mapped earlier, when the line was installed in the D-cache.

This ASI_PHYS_USE_EC load hitting on the D-cache behavior is not defined or tested, so software should not rely on it.

When a store is done with a physical address, using ASI=0x14 (ASI_PHYS_USE_EC), causing CP==1 and CV==0, and the address hits in the D-cache D-cache, the D-cache apparently does get updated. However, this behavior is not verified or guaranteed. Again, software should make sure the physical address is not in the D-cache, before accessing that address using CP==1, CV==0, whether by a TLB mapping, or using one of the special ASIs.

K.3 Errata created by UltraSPARC IIi

Erratum 1171: Noncacheable load/store using PA[40:0] that maps to the unused PBM PCI Configuration Space (function!=0) can result in a *deadlock*.

There are two situations:

- The first is an "illegal" case. Noncacheable load/store with PA[40:0] in the range 0x1FE.0100.0100–0x1FE.0100.07FF, and the ASI is 0x77 or 0x7F (SDB CSRs). Note that these PAs are unspecified in this manual. Normally, unspecified addresses like this can alias to other CSRs—see Section 19.4.3, *DMA Error Registers* on page 316—but in this case a deadlock may occur.
- The second case is a noncacheable load or store to the range to the range 0x1FE.0100.0100–0x1FE.0100.07FF. This is the PBM's PCI configuration space, for function!=0. The PBM has no valid CSRs for nonzero function ID.

The 2.1 PCI specification says that references to any unused configuration space should be a no-op.

Glossary

This glossary defines some important words and acronyms used throughout this manual. *Italicized words* within definitions are further defined elsewhere in the list.

alias Two virtual addresses are aliases of each other if they refer to the same physical address.

ASI Abbreviation for Address Space Identifier.

clean window A clean register window is one in which all of the registers contain either zero or a valid address from the current address space or valid data from the current address space.

coherence A set of protocols guaranteeing that all memory accesses are globally visible to all caches on a shared-memory bus.

consistency See *coherence* .

context A set of translations used to support a particular address space. See also *MMU* .

copyback The process of copying back a cache line in response to a hit while *snooping* .

CPI Cycles per instruction. The number of clock cycles it takes to execute one instruction.

current window The block of 24 *r* registers to which the Current Window Pointer (CWP) register points.

demap To invalidate a mapping in the MMU.

dispatch To issue a fetched instruction to one or more functional units for execution.

DMA Accesses by a master on the secondary bus to a target on the primary bus. Equivalent to upstream.

E-cache, E$ refers to the external, SRAM-based, second-level cache. It is also known as level-2 cache or L2 cache.

fccN One of the floating-point condition code fields *fcc0*, *fcc1*, *fcc2*, or *fcc3*.

floating-point exception An exception that occurs during the execution of an FPop instruction while the corresponding bit in FSR.TEM is set to 1. The exceptions are: *unfinished_FPop*, *unimplemented_FPop*, *sequence_error*, *hardware_error*, *invalid_fp_register*, and *IEEE_754_exception*.

floating-point IEEE-754 exception A *floating-point exception*, as specified by IEEE Std 754-1985.

floating-point trap type The specific type of a *floating-point exception*, encoded in the FSR.*ftt* field.

implementation-dependent An aspect of the architecture that may legitimately vary among implementations. In many cases, the permitted range of variation is specified in the SPARC-V9 standard. When a range is specified, compliant implementations shall not deviate from that range.

ISA **instruction set architecture**: an ISA defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. An ISA does not define clock cycle times, cycles per instruction, data paths, etc.

L2-cache This term is an abbreviation for *level-2 cache*. It refers to the external, SRAM-based, second-level cache.

may A key word indicating flexibility of choice with no implied preference.

MMU **Memory Management Unit**: a mechanism that implements a policy for address translation and protection among contexts. See also *virtual address*, *physical address*, and *context*.

module A master or slave device that attaches to the shared-memory bus.

next program counter (nPC) A register that contains the address of the instruction to be executed next, if a trap does not occur.

non-privileged An adjective that describes (1) the state of the processor when PSTATE.PRIV=0, i.e., *non-privileged mode*; (2) processor state that is accessible to software while the processor is in either *privileged mode* or *non-privileged mode*; e.g., non-privileged registers, non-privileged ASRs, or, in general, non-privileged state; (3) an instruction that can be executed when the processor is in either *privileged mode* or *non-privileged mode*.

non-privileged mode The mode in which the processor is operating when PSTATE.PRIV=0. See also *privileged*.

NWINDOWS The number of register windows present in a particular implementation.

optional A feature not required for SPARC-V9 compliance.

PCI Peripheral Component Interconnect (bus). A high-performance 32 or 64-bit bus with multiplexed address and data lines.

physical address An address that maps real physical memory or I/O device space. See also *virtual address*.

PIO Accesses by a master on the primary bus to a target on the secondary bus. Equivalent to downstream.

prefetchable A memory location 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.

Non-prefetchable 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 *side effect*.

privileged An adjective that describes (1) the state of the processor when PSTATE.PRIV=1, that is, *privileged mode*; (2) processor state that is only accessible to software while the processor is in *privileged mode*; e.g., privileged registers, privileged ASRs, or, in general, privileged state; (3) an instruction that can be executed only when the processor is in *privileged mode*.

privileged mode The processor is operating in privileged mode when PSTATE.PRIV=1.

program counter (PC) A register that contains the address of the instruction currently being executed by the *IU*.

RED_state **R**eset, **E**rror, and **D**ebug **state**. The processor is operating in RED_state when PSTATE.RED=1.

restricted An adjective used to describe an address space identifier (ASI) that may be accessed only while the processor is operating in *privileged mode*.

reserved Used to describe 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 field should only be written to zero by software. A reserved register field should read as zero in hardware; software intended to run on future versions of SPARC-V9 should not assume that the field will read as zero or any other particular value. Throughout this document, figures illustrating registers and instruction encodings always indicate reserved fields with an em dash '—'.

reset trap A vectored transfer of control to *privileged* software through a fixed-address reset trap table. Reset traps cause entry into *RED_state*.

rs1*, *rs2*, *rd The integer register operands of an instruction. *rs1* and *rs2* are the source registers; *rd* is the destination register.

shall A key word indicating a mandatory requirement. Designers shall implement all such mandatory requirements to ensure inter-operability with other SPARC-V9-conformant products. The key word "must" is used interchangeably with the key word shall.

should A key word indicating flexibility of choice with a strongly preferred implementation. The phrase "it is recommended" is used interchangeably with the key word should.

side effect A memory location is deemed to have side effects if additional actions beyond the reading or writing of data may 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.

snooping The process of maintaining coherency between caches in a shared-memory bus architecture. All cache controllers monitor (snoop) the bus to determine whether they have a copy of a shared cache block.

speculative load A load operation (e.g., non-faulting load) that is carried out before it is known whether the result of the operation is required. These accesses typically are used to speed program execution. 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.

supervisor software Software that executes when the processor is in *privileged mode* .

TLB **Translation Lookaside Buffer**: A hardware cache located within the MMU, which contains copies of recently used translations. Technically, there are separate TLBs for the instruction and data paths; the I-MMU contains the iTLB and the D-MMU the dTLB.

TLB hit The desired translation is present in the on-chip TLB.

TLB miss The desired translation is not present in the on-chip TLB.

trap A vectored transfer of control to supervisor software through a table, the address of which is specified by the privileged Trap Base Address (TBA) register.

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 (preferably within any guidelines given).

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 may deliver random results, may or may not cause a trap, may vary among implementations, and may vary with time on a given implementation.

unimplemented An architectural feature that is not directly executed in hardware because it is optional or is emulated in software.

unpredictable Synonymous with *undef ined* .

unrestricted An adjective used to describe an address space identifier (*ASI*) that may be used regardless of the processor mode; that is, regardless of the value of PSTATE.PRIV.

virtual address An address produced by a processor that maps all system-wide, program-visible memory. Virtual addresses usually are translated by a combination of hardware and software to physical addresses, which can be used to access physical memory.

writeback: The process of writing a dirty cache line back to memory before it is refilled.

Bibliography

General References

Books and Specifications

Weaver, David L., editor. *The SPARC Architecture Manual, Version 8*, Prentice-Hall, Inc., 1992.

Weaver, David L., and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*, Prentice-Hall, Inc., 1994.

Institute of Electrical and Electronics Engineers (IEEE) *1985. IEEE Standard for Binary Floating-Point Arithmetic*, *IEEE Std 754-1985*. New York: IEEE.

Institute of Electrical and Electronics Engineers (IEEE) 1990. *IEEE Std 1149.1-1990*, *IEEE Standard Test Access Port and Boundary-Scan Architecture*. New York: IEEE.

PCI Special Interest Group. *April 1994. PCI Local Bus Specification*, *Revision 2.1.* Portland, Oregon: PCI Special Interest Group.

Papers

Boney, Joel. "SPARC Version 9 Points the Way to the Next Generation RISC," *SunWorld*, October 1992, pp. 100-105.

Greenley, D., et. al., "UltraSPARC™: The Next Generation Superstar 64-bit SPARC," 40th Annual CompCon, 1995.

Kaneda, Shigeo. “A Class of Odd-Weight-Column SEC-DED-SbED Codes for Memory System Applications.” *IEEE Transactions on Computers*, August 1984.

Kohn, L., et. al., ”The Visual Instruction Set (VIS) in UltraSPARC™,” 40th annual CompCon, 1995.

Tremblay, Marc. “A Fast and Flexible Performance Simulator for Microarchitecture Trade-off Analysis on UltraSPARC,” DAC 95 Proceedings.

Zhou, C., et. al., “MPEG Video Decoding with UltraSPARC Visual Instruction Set,” 40th Annual CompCon, 1995.

Microelectronics Publications

These books and papers are available in printed form, and some are also available through the World Wide Web (WWW). See *On Line Resources* below for information about the SME WWW pages.

Data Sheets

UltraSPARC IIi Highly Integrated 64-bit RISC Processor, PCI Interface, SME1040: 805-0086

UltraSPARC IIi CPU; Highly Integrated 64-bit RISC; L2-Cache, DRAM, PCI Interfaces, SME1430: 805-7291

UltraSPARC IIi Advanced PCI Bridge (APB™), SME2411: 805-0088

User’s Guides

UltraSPARC User’s Manual: 802-7220

Advanced PCI Bridge User’s Manual: 805-1251

UltraSPARC-I Reset/Interrupt/Clock Controller User’s Manual: 805-0167

Other Materials

UltraSPARC IIi Processor White Paper (WPR-0029-01)

UltraSPARC Nested Trap White Paper (STB0045)

UltraSPARC Evaluating Processor Performance White Paper (STB0014)

UltraSPARC-II Advanced Branch Prediction and Single Cycle Following White Paper (STB0023)

UltraSPARC-II Advanced Memory Structure White Paper (STB0022)

UltraSPARC-II White Paper (STB0114)

UltraSPARC-II Prefetch White Paper (STB0116)

UltraSPARC-II Multiple Outstanding Requests White Paper (STB0117)

How to Contact

Microelectronics is a division of:

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA, U.S.A. 94303
Tel: 800 681-8845

On Line Resources

The Microelectronics Worldwide Web page is located at:

http://www.sun.com/microelectronics

It contains the latest information about the entire UltraSPARC IIi product line, and may be used to download HTML, PostScript, or Acrobat PDF copies of the II*i* data sheets.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

T

U

V

W

X

Y