UltraSPARC T1™ Supplement

to the *UltraSPARC Architecture 2005*

Draft D2.1, 14 May 2007

Privilege Levels: Hyperprivileged,
Privileged,
and Nonprivileged

Distribution: Public

Comments and "bug reports" regarding this document are welcome; they should be submitted to email address: UST1-editor@sun.com

Contents

Preface

Welcome to the UltraSPARC T1 Processor Supplement, D2.1. This document contains information about the processor-specific aspects of the architecture and programming of the UltraSPARC T1 processor, one of Sun Microsystems' family of processors compliant with UltraSPARC Architecture™. It is intended to supplement the *UltraSPARC Architecture 2005* with implementation-specific information.

Target Audience

This User's Guide is mainly targeted for programmers who write software for the UltraSPARC T1 processor. This manual contains a depository of information that is useful to operating system programmers, application software programmers and logic designers, who are trying to understand the architecture and operation of the UltraSPARC T1 processor. This manual is both a guide and a reference manual for programming of the processor.

Fonts and Notational Conventions

Fonts are used as follows:

- *Italic* font is used for emphasis, book titles, and the first instance of a word that is defined.

- *Italic* font is also used for terms where substitution is expected, for example, "f`cc`*n*", "virtual processor *n*", or "*reg_plus_imm*".

- *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."

- lowercase helvetica font is used for register field names (named bits) and instruction field names, for example: "The rs1 field contains...."

- UPPERCASE HELVETICA font is used for register names; for example, FSR.

- TYPEWRITER (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: %f0, ASI_PRIMARY, execute_state.

- When a register field is shown along with its containing register name, they are separated by a period ('.'), for example, FSR.cexc.

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

- An underscore character joins words in register, register field, exception, and trap names. **Note:** Such words may be split across lines at the underbar without an intervening hyphen. For example: "This is true whenever the integer_condition_ code field...."

The following notational conventions are used:

- The left arrow symbol (←) is the assignment operator. For example, "PC ← PC + 1" means that the Program Counter (PC) is incremented by 1.

- Square brackets ([ ]) are used in two different ways, distinguishable by the context in which they are used:

  - Square brackets indicate indexing into an array. For example, TT[TL] means the element of the Trap Type (TT) array, as indexed by the contents of the Trap Level (TL) register.

  - Square brackets are also used to indicate optional additions/extensions to symbol names. For example, "ST[D,Q]F" expands to all three of "STF", "STDF", and "STQF". Similarly, ASI_PRIMARY[_LITTLE] indicates two related address space identifiers, ASI_PRIMARY and ASI_PRIMARY_LITTLE. (Contrast with the use of angle brackets, below)

- Angle brackets (< >) indicate mandatory additions/extensions to symbol names. For example, "ST<D|Q>F" expands to mean "STDF" and "STQF". (Contrast with the second use of square brackets, above)

- Curly braces ({ }) indicate a bit field within a register or instruction. For example, CCR{4} refers to bit 4 in the Condition Code Register.

- A consecutive set of values is indicated by specifying the upper and lower limit of the set separated by a colon (:), for example, CCR{3:0} refers to the set of four least significant bits of register CCR. (Contrast with the use of double periods, below)

- A double period (..) indicates any *single* intermediate value between two given end values is possible. For example, NAME[2..0] indicates four forms of NAME exist: NAME, NAME2, NAME1, and NAME0; whereas NAME<2..0> indicates that three forms exist: NAME2, NAME1, and NAME0. (Contrast with the use of the colon, above)

- A vertical bar (|) separates mutually exclusive alternatives inside square brackets ([ ]), angle brackets (< >), or curly braces ({ }). For example, "NAME[A|B]" expands to "NAME, NAMEA, NAMEB" and "NAME<A|B>" expands to "NAMEA, NAMEB".

- The asterisk (*) is used as a wild card, encompassing the full set of valid values. For example, FCMP* refers to FCMP with all valid suffixes (in this case, FCMP<s|d|q> and FCMPE<s|d|q>). An asterisk is typically used when the full list of valid values either is not worth listing (because it has little or no relevance in the given context) or the valid values are too numerous to list in the available space.

- The slash (/) is used to separate paired or complementary values in a list, for example, "the LDBLOCKF/STBLOCKF instruction pair ...."

- The double colon (**::**) is an operator that indicates concatenation (typically, of bit vectors). Concatenation strictly strings the specified component values into a single longer string, in the order specified. The concatenation operator performs no arithmetic operation on any of the component values.

Notation for Numbers

Numbers throughout this specification are decimal (base-10) unless otherwise indicated. Numbers in other bases are followed by a numeric subscript indicating their base (for example, 1001_2, FFFF 0000_{16}). In some cases, numbers may be preceded by "0x" to indicate hexadecimal (base-16) notation (for example, 0xFFFF 0000). Long binary and hexadecimal numbers within the text may have spaces inserted every four characters to improve readability.

An en dash (–) with no spaces indicates a range, for example, 0001_{16}–0000_{16}.

Also see the colon (:) and double period (..) notation described in the previous section.

Informational Notes

This manual provides several different types of information in notes, as follows:

Note | General notes contain incidental information relevant to the paragraph preceding the note.

Programming Note	Programming notes contain incidental information about how software can use an architectural feature.
Implementation Note	An Implementation Note contains incidental information, describing how an UltraSPARC Architecture processor might implement an architectural feature.
V9 Compatibility Note	Note containing information about possible differences between UltraSPARC Architecture and SPARC V9 implementations. Such information may not pertain to other SPARC V9 implementations.

UltraSPARC T1 Basics

This chapter introduces the UltraSPARC T1 chip-level multithreaded (CMT) processor.

1.1 Background

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

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

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

TLP

Strand 1

Strand 2

Strand 3

Strand 4

ILP

Single strand executing
2 instructions per cycle

Executing   Stalled on Memory

FIGURE 1-1   Differences Between TLP and ILP

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

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

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

---

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

1.2   UltraSPARC T1 Overview

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

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

A block diagram of the UltraSPARC T1 chip is shown in FIGURE 1-2.
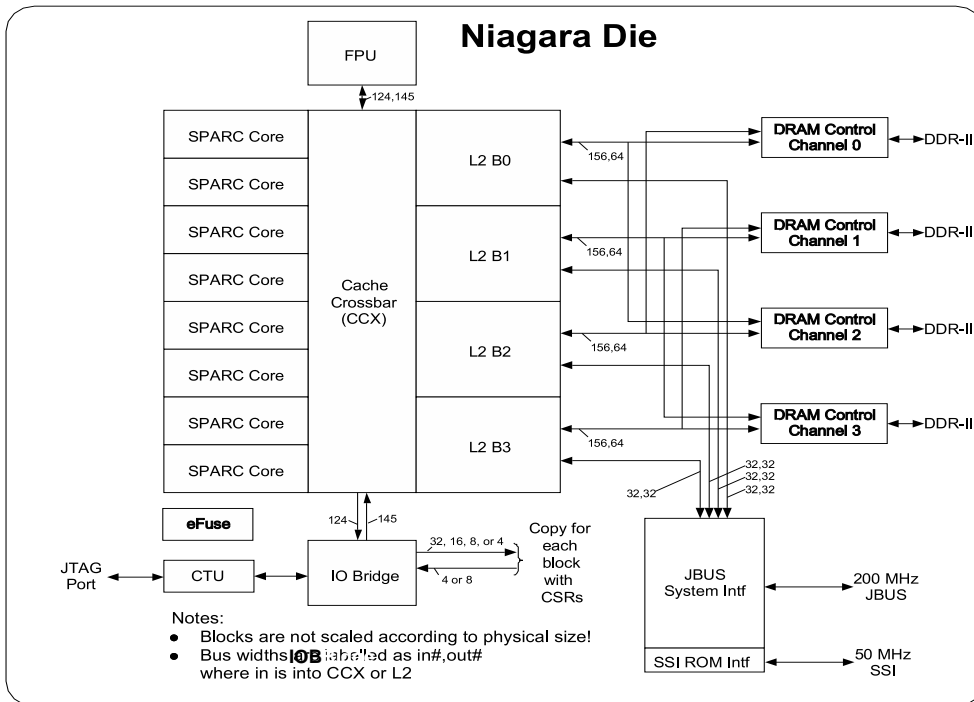
FIGURE 1-2   UltraSPARC T1 Chip Block Diagram

1.3   UltraSPARC T1 Components

This section describes each component in UltraSPARC T1:

- SPARC physical core
- Floating-point unit
- L2 cache
- DRAM controller
- IOB
- JBUS interface
- SSI ROM interface

1.3.1   SPARC Physical Core

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

The core pipeline consists of six stages: Fetch, Switch, Decode, Execute, Memory and Writeback. As shown in FIGURE 1-3, the Switch stage contains a strand instruction register for each strand. One of the strands is picked by the strand scheduler and the current instruction for that strand is issued to the pipe. While this is done, the hardware fetches the next instruction for that strand and updates the strand instruction register.

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

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

FIGURE 1-3 illustrates SPARC physical core.
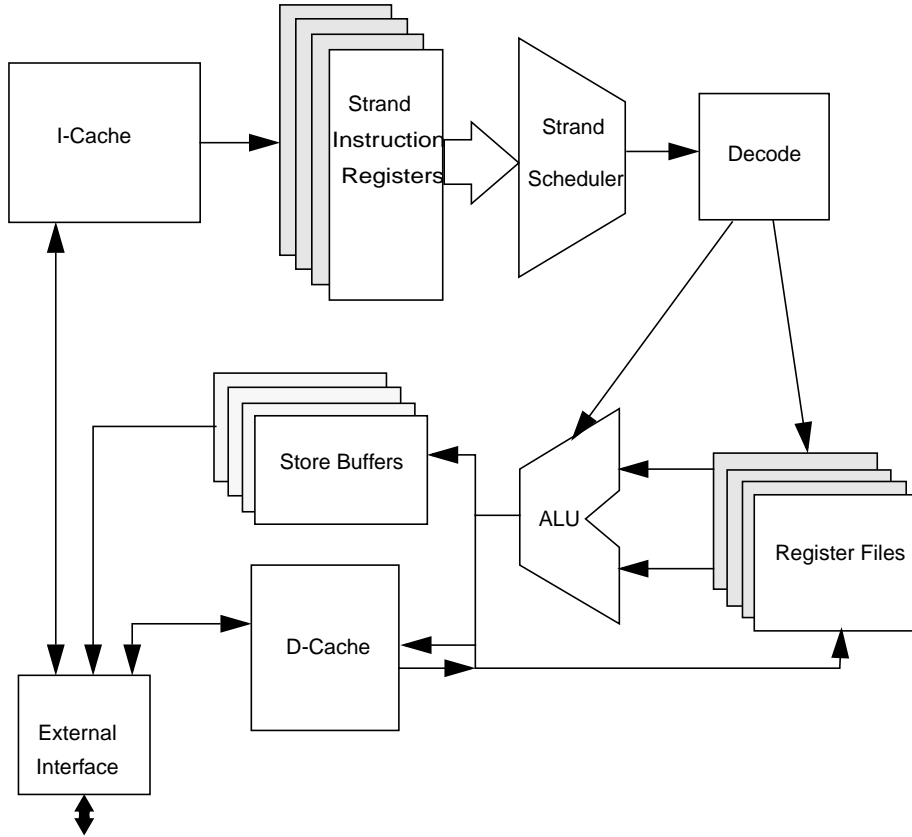
FIGURE 1-3 SPARC Core Block Diagram

1.3.2 Floating-Point Unit (FPU)

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

1.3.3 L2 Cache

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

1.3.4　DRAM Controller

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

UltraSPARC T1 uses DDR2 DIMMs, and can support one or two ranks of stacked or unstacked DIMMs. Each DRAM bank/port is two DIMMs wide (128b +16b ECC). All installed DIMMs on an individual bank/port must be identical, and the same total amount of memory (# of bytes) must be installed on each DRAM Controller port. The DRAM controller frequency is an exact ratio of the CMP core Frequency, where the CMP core frequency must be at least 4X the DRAM controller frequency. The DDR (Double Data Rate) data busses, of course, transfer data at twice the frequency of the DRAM Controller frequency.

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

1.3.5　IOB Unit

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

1.3.6　JBUS Interface (JBI)

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

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

1.3.7　SSI ROM Interface

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

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

Data Formats

The UltraSPARC T1 processor supports all UltraSPARC Architecture 2005 data formats; see the Data Formats chapter of the *UltraSPARC Architecture 2005* for details.

Registers

This chapter discusses the specifics of UltraSPARC T1 registers, as they differ from the register definitions in *UltraSPARC Architecture 2005*.

3.1 Ancillary State Registers (ASRs)

3.1.1 TICK Register

See the *UltraSPARC Architecture 2005* for a general description of this register.

The TICK register contains two fields: npt and counter. On an UltraSPARC T1 processor, the npt field is replicated per strand, while the counter field is shared by all four strands on a physical processor core. Hyperprivileged software on any strand can write the TICK register. A write of the TICK register will update both the shared counter as well as the writing strand's npt field (the npt fields for other strands will be unaffected). The counter increments each physical processor core clock but, on an UltraSPARC T1 processor, the least significant 2 bits of the counter field always read as 0.

3.1.2 General Status Register (GSR)

Each strand has a nonprivileged General Status register (GSR), as described in the *UltraSPARC Architecture 2005*.

All UltraSPARC Architecture 2005 GSR fields are supported in the UltraSPARC T1 implementation. However, the mask and scale fields are not directly written by VIS instructions; they are provided for use by software emulation.

3.1.3 Software Interrupt Register (SOFTINT)

Each strand has a privileged software interrupt register, as described in the *UltraSPARC Architecture 2005*.

The software interrupt register contains three fields: sm, int_level, and tm. Setting any of sm, tm, or SOFTINT{14} generates an *interrupt_level_14* exception. However, these bits are considered completely independent of each other. Thus, a Stick Compare event will only set bit 16 and generate *interrupt_level_14* exception, not also set bit 14.

UltraSPARC T1 Programming Note	It is possible (but difficult) in UltraSPARC T1 for software to clear a SOFTINT bit between the setting of that bit and the generation of the interrupt from the bit being set because (there is a three-cycle window between the setting of the bit and the interrupt in UltraSPARC T1). If software were to do this, it would see an *interrupt_level_n* interrupt, but would find no bit set in the SOFTINT register. Note that normal software would only clear a bit in response to taking the *interrupt_level_n* exception, so this race condition should not occur in normal operation.
UltraSPARC T1 Programming Note	It is possible, but even more difficult than the above case, for software to zero a SOFTINT bit as it is getting set to 1, while another core is accessing its SOFTINT register, with timing such that hardware decides to take a SOFTINT trap, but the SOFTINT register is clear by the time it decides the trap number. In this case, hardware will take a trap 40_{16}. Since software should only clear a bit that is known to be set, this should never happen in normal operation.

3.1.4 Tick Compare Register (TICK_CMPR)

Each strand has a privileged Tick Compare (TICK_CMPR) register, as described in the *UltraSPARC Architecture 2005*.

3.1.5 System Tick Register (STICK)

On an UltraSPARC T1 processor, the STICK register is an alias for the TICK register. Writes to STICK will be reflected in TICK, and vice versa. See the description of TICK above for the behavior of this register.

3.1.6 System Tick Compare Register (STICK_CMPR)

Each strand has a privileged System Tick Compare (STICK_CMPR) register, as described in the *UltraSPARC Architecture 2005*.

3.1.7 PCR and PIC Registers

TABLE 3-1 UltraSPARC T1-Specific Performance Instrumentation Registers

ASR Number	ASR Name	Access	priv	Replicated by Strand	Description
10_{16}	PCR	RW	Y^2	Y	Performance counter control register
11_{16}	PIC	RW	Y^1	Y	Performance Instrumentation Counter register

Notes:

1. Nonprivileged access with PCR.priv = 1 causes a *privileged_action* exception.

2. Nonprivileged access causes a *privileged_opcode* exception.

3.1.8 Strand Status Register

The Stand Status register (STRAND_STS_REG) is used to control CMT operation. It is specific to the UltraSPARC T1 implementation.

TABLE 3-2 UltraSPARC T1-Specific Strand Status Register

ASR Number	ASR Name	Access	priv	Replicated by Strand	Description
$1A_{16}$	STRAND_STS_REG	RW	Y^2	Y	Strand Status & Issue Control register

3.2 PR State Registers

3.2.1 Trap State (TSTATE)

Each virtual processor (strand) has *MAXTL*(6) Trap State (TSTATE) registers, as described in the *UltraSPARC Architecture 2005*.

3.2.2 Processor State Register (PSTATE)

Each virtual processor (strand) has a Processor State register, as described in the *UltraSPARC Architecture 2005*.

When not in hyperprivileged mode, disrupting traps destined for hyperprivilegd mode ignore the PSTATE.ie bit.

3.2.3 Trap Level Register (TL)

Each virtual processor (strand) has a Trap Level register, as described in the *UltraSPARC Architecture 2005*.

The maximum privileged trap level visible in privileged mode (*MAXPTL*) for UltraSPARC T1 is 2.

The maximum trap level (*MAXTL*) for UltraSPARC T1 is 6.

3.2.4 Global Level Register (GL)

Each virtual processor (strand) has a Global Level register, as described in the *UltraSPARC Architecture 2005*.

The maximum privileged global level visible in privileged mode (*MAXPGL*) for UltraSPARC T1 is 2.

The maximum global level (*MAXGL*) for UltraSPARC T1 is 3.

3.3 Floating-Point State Register (FSR)

Each virtual processor (strand) has a Floating-Point State register, FSR, as described in the *UltraSPARC Architecture 2005*.

UltraSPARC T1 does not provide a nonstandard floating-point mode, so the ns field of FSR is always 0.

On UltraSPARC T1, FSR.ver always reads as 0.

FSR.qne always reads as 0, because UltraSPARC T1 neither needs nor supports a floating-point queue (FQ).

3.4 Hyperprivileged Registers

This section describes UltraSPARC T1's hyperprivileged registers.

3.4.1 Hyperprivileged Processor State Register (HPSTATE)

Each UltraSPARC T1 virtual processor has a Hyperprivileged Processor State register (HPSTATE), as described in the *UltraSPARC Architecture 2005*.

			RW			*RW*		*RW*		*RW*
HPSTATE^H	—	i.d.	ibe	—	red	—	hpriv	—	tlz

HPSTATE^H | — | i.d.|ibe| — |red| — |hpriv| — |tlz
63   12 11 10 9   6 5 4   3 2   1 0
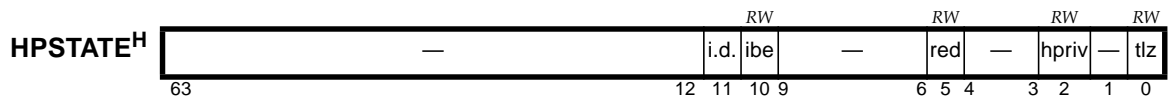
FIGURE 3-1 HPSTATE Fields

> **Programming Note** On UltraSPARC T1, whenever the HPSTATE register is written, care must be taken that the implementation-dependent bit HPSTATE{11} is *always* set to the value '1', to avoid unspecified behavior.

Full documentation on the Hypervisor Processor State register can be found in the *UltraSPARC Architecture 2005*.

Note that the tlz bit retains its current value when a trap is taken. This behavior is different from that specified in the *UltraSPARC Architecture 2005* which states that HPSTATE.tlz is reset to 0 when certain traps are taken.

3.4.2 Hyperprivileged Trap State Register (HSTATE)

Each UltraSPARC T1 virtual processor has a set of Hyperprivileged Trap State (HSTATE) registers, one per trap level. Full documentation on this register can be found in the UA-2005 specification.

3.4.3 Hyperprivileged Interrupt Pending Register (HINTP)

Each UltraSPARC T1 virtual processor has a Hyperprivileged Interrupt Pending (HINTP) register. Full documentation on this register can be found the UA-2005 specification.

3.4.4 Hyperprivileged Trap Base Address Register (HTBA)

Each UltraSPARC T1 virtual processor has a Hyperprivileged Trap Base Address (HTBA) register. Full documentation on this register can be found the UA-2005 specification. Note that UltraSPARC T1 only implements bits 47:14 of the tba field. Bits 63:48 are always sign-extended from bit 47.

3.4.5 Hyperprivileged Version Register (HVER)

All virtual processors on an UltraSPARC T1 physical processor share a read-only hyperprivileged Version (HVER) register. Writes to this register generate an *illegal_instruction* trap.

3.4.6 Hyperprivileged System Tick Compare Register (HSTICK)

Each UltraSPARC T1 virtual processor has a Hyperprivileged System Tick Compare (HSTICK) register. The hyperprivileged system TICK compare register contains two fields: int_dis and hstick_cmpr. A full 63-bit hstick_cmpr field is implemented in the register, but the least significant two bits are ignored when comparing against the STICK counter field. The int_dis bit controls whether a *hstick_match* trap is generated, and the hsp bit is set in the HINTP register when hstick_cmpr bits 62:2 match the STICK counter field.

3.4.7 Strand Status Register

On an UltraSPARC T1 processor, each virtual processor (strand) has an (UltraSPARC T1-specific) hyperprivileged Strand Status Register that contains control bits for stranded operation, as well as strand status. Hyperprivileged software can read/write the whole register. The format of this register is shown in TABLE 3-14.

TABLE 3-3 Strand Status Register – STRAND_STS_REG (ASR $1A_{16}$)

Bit(s)	Field	R/W	Description
63:52	—	R	*Reserved*
51:48	wait_s	R	Wait mask on store buffer for strands 3:0.
47:44	wait_o	R	Wait mask on other conditions for strands 3:0.
43:40	wait_i	R	Wait mask on instruction miss for strands 3:0.
39:36	—	R	*Reserved*
35:31	sfsm_state0	R	State of strand 0 fsm.

Bit(s)	Field	R/W	Description
30:26	sfsm_state1	R	State of strand 1 fsm.
25:21	sfsm_state2	R	State of strand 2 fsm.
20:16	sfsm_state3	R	State of strand 3 fsm.
15:13	—	R	*Reserved*
12:10	strand_id	R	ID of the physical processor core within the UltraSPARC T1 chip.
9:8	strand_id	R	ID of the strand (virtual processor) within the physical processor core.
7:3	—	R	*Reserved*
2	spec_en	RW	Enables speculative strand execution following loads. This bit is shared among all strands (that is, if one strand sets this bit to 1, all strands will have speculative execution enabled).
1	rsvd4	R	*Reserved*
0	active	RW	Controls whether strand is active; 1 = active, 0 = halted or idle.

Speculative strand execution refers to whether the hardware speculates that most loads (all loads except floating-point loads and LDD) will hit in the L1 Data cache. When spec_en = 1, instructions from a strand following the load will be issued after the load-use penalty is satisfied, but before hardware knows whether the load hit in the L1 data cache. These speculative instructions are lower priority than nonspeculative instructions from other strands, and will only be issued if no other strand has nonspeculative instructions ready for issue.

TABLE 3-15 lists the encodings for the sfsm_state* fields.

TABLE 3-4 sfsm_state Encodings

Strand State	Description
IDLE	00000
WAIT	00001
HALT	00010
RUN	00101
SPECULATIVE_RUN	00111
SPECULATIVE_READY	10011
READY	11001

Implementation Note | Privileged software access to the Strand Status register is *deprecated*. However, on UltraSPARC T1, privileged software does have limited access. An attempt to read or write the Strand Status register while in privileged mode does not cause an exception. A read returns only the active bit (all other bits read as 0); a write affects only the active bit (writes to other bits are ignored). Nonetheless, software should only access this register in hyperprivileged mode.

Instruction Set Overview

The UltraSPARC T1 processor implements the instruction set described in the *UltraSPARC Architecture 2005*. Additional UltraSPARC T1-specific details are described in this chapter.

4.1 State Register Access

UltraSPARC T1 supports the standard ASRs described in the *UltraSPARC Architecture 2005*.

In addition, UltraSPARC T1 provides the RDasr and WRasr instruction described in TABLE 4-1

TABLE 4-1 UltraSPARC T1-Specific RDasr and WRasr Instructions

ASR #	ASR Name	Description	R, W?	Priv?
26	strand_status	Strand Status and Control register	RW	Yes

4.2 Floating-Point Operate (FPop) Instructions

UltraSPARC T1 implements the floating-point instruction set described in the *UltraSPARC Architecture 2005*.

UltraSPARC T1 generates the correct IEEE Std 754-1985 results (impl. dep. #3).

All floating-point quad-precision operations cause an *fp_exception_other* trap with FSR.ftt = unimplemented_FPop, and system software must emulate those operations.

4.3 Reserved Opcodes and Instruction Fields

An attempt to execute an opcode to which no instruction is assigned causes a trap. Specifically:

- Attempting to execute a reserved FPop causes an *fp_exception_other* trap (with FSR.ftt = unimplemented_FPop).

- Attempting to execute any other reserved opcode causes an *illegal_instruction* trap.

- Attempting to execute a Tcc instruction with a nonzero value in the reserved field (bits 10:8 and 6:5 when i = 0 or bits 10:7 when i = 1) causes an *illegal_instruction* trap. See *Trap on Integer Condition Codes (Tcc)* on page 25.

See Appendix C, *Opcode Maps*, for a complete enumeration of the opcode assignments.

4.4 Register Window Management

N_REG_WINDOWS = 8 on UltraSPARC T1 (impl. dep. #2-V8). The state of the eight register windows is determined by the contents of the set of privileged registers described in the *UltraSPARC Architecture 2005*.

Instruction Definitions

5.1 Instruction Set Summary

The UltraSPARC T1 CPU implements both the standard UltraSPARC Architecture 2005 instruction set and a number of implementation-dependent extended instructions. Standard UltraSPARC Architecture 2005 instructions are documented in the *UltraSPARC Architecture 2005*. UltraSPARC T1 extended instructions are documented in *VIS Instructions* on page 25.

The superscripts and their meanings are defined in TABLE 5-1.

TABLE 5-1 Instruction Superscripts

Superscript	Meaning
D	Deprecated instruction
H	Hyperprivileged instruction
P	Privileged instruction

UltraSPARC T1 executes most UltraSPARC Architecture 2005 instructions in hardware. Those that trap and are emulated in software are listed in TABLE 5-2.

TABLE 5-2 UltraSPARC Architecture 2005 Instructions Not Directly Implemented by UltraSPARC T1 Hardware *(1 of 3)*

Instruction	Description	Exception Caused by Attempted Execution
ALLCLEAN	Mark all windows as clean	*illegal_instruction*
ARRAY{8,16,32}	3-D address to blocked byte address conversion	*illegal_instruction*
BMASK	Write the GSR.mask field	*illegal_instruction*
BSHUFFLE	Permute bytes as specified by the GSR.mask field	*illegal_instruction*
EDGE{8,16,32}{L}{N}	Edge boundary processing {little-endian} {non-condition-code altering}	*illegal_instruction*

TABLE 5-2 UltraSPARC Architecture 2005 Instructions Not Directly Implemented by UltraSPARC T1 Hardware *(2 of 3)*

Instruction	Description	Exception Caused by Attempted Execution
FABSq	Floating-point absolute value quad	*fp_exception_other* [unimplemented_FPop]
FADDq	Floating-point add quad	*fp_exception_other* [unimplemented_FPop]
FCMPq	Floating-point compare quad	*fp_exception_other* [unimplemented_FPop]
FCMPEq	Floating-point compare quad (exception if unordered)	*fp_exception_other* [unimplemented_FPop]
FCMPEQ{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 = src2	*illegal_instruction*
FCMPGT{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 > src2	*illegal_instruction*
FCMPLE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 ≤ src2	*illegal_instruction*
FCMPNE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 ≠ src2	*illegal_instruction*
FDIVq	Floating-point divide quad	*fp_exception_other* [unimplemented_FPop]
FdMULq	Floating-point multiply double to quad	*fp_exception_other* [unimplemented_FPop]
FEXPAND	Four 8-bit to 16-bit expand	*illegal_instruction*
FiTOq	Convert integer to quad floating-point	*fp_exception_other* [unimplemented_FPop]
FMOVq	Floating-point move quad	*fp_exception_other* [unimplemented_FPop]
FMOVqcc	Move quad floating-point register if condition is satisfied	*fp_exception_other* [unimplemented_FPop]
FMOVqr	Move quad floating-point register if integer register contents satisfy condition	*fp_exception_other* [unimplemented_FPop]
FMULq	Floating-point multiply quad	*fp_exception_other* [unimplemented_FPop]
FMUL8SUx16	Signed upper 8- x 16-bit partitioned product of corresponding components	*illegal_instruction*
FMUL8ULx16	Unsigned lower 8-bit x 16-bit partitioned product of corresponding components	*illegal_instruction*
FMUL8x16	8- x 16-bit partitioned product of corresponding components	*illegal_instruction*
FMUL8x16AL	Signed lower 8-bit x 16-bit lower α partitioned product of four components	*illegal_instruction*
FMUL8x16AU	Signed upper 8-bit x 16-bit lower α partitioned product of four components	*illegal_instruction*
FMULD8SUx16	Signed upper 8-bit x 16-bit multiply ← 32-bit partitioned product of components	*illegal_instruction*
FMULD8ULx16	Unsigned lower 8-bit x 16-bit multiply ← 32-bit partitioned product of components	*illegal_instruction*

Instruction	Description	Exception Caused by Attempted Execution
FNEGq	Floating-point negate quad	*fp_exception_other* [unimplemented_FPop]
FPACKFIX	Two 32-bit to 16-bit fixed pack	*illegal_instruction*
FPACK{16,32}	Four 16-bit/two 32-bit pixel pack	*illegal_instruction*
FPMERGE	Two 32-bit to 64-bit fixed merge	*illegal_instruction*
FSQRT(s,d,q)	Floating-point square root	*fp_exception_other* [unimplemented_FPop]
F(s,d,q)TO(q)	Convert between floating-point formats to quad	*fp_exception_other* [unimplemented_FPop]
FqTOi	Convert quad floating point to integer	*fp_exception_other* [unimplemented_FPop]
FqTOx	Convert quad floating point to 64-bit integer	*fp_exception_other* [unimplemented_FPop]
FSUBq	Floating-point subtract quad	*fp_exception_other* [unimplemented_FPop]
FxTOq	Convert 64-bit integer to floating-point	*fp_exception_other* [unimplemented_FPop]
IMPDEP1	Implementation-dependent instruction	*illegal_instruction*
IMPDEP2	Implementation-dependent instruction	*illegal_instruction*
INVALW[P]	Mark all windows as CANSAVE	*illegal_instruction*
LDQF	Load quad floating-point	*illegal_instruction*
LDQFA	Load quad floating-point into alternate space	*illegal_instruction*
LDSHORTF	Short FP load, zero-extend 8/16-bit load to a double-precision floating-point register	*data_access_exception*
NORMALW	Mark other windows as restorable	*illegal_instruction*
OTHERW	Mark restorable windows as other	*illegal_instruction*
PDIST	Distance between eight 8-bit components	*illegal_instruction*
POPC	Population count	*illegal_instruction*
PST	Eight 8-bit/four 16-bit/two 32-bit partial stores	*data_access_exception*
SHUTDOWN[D,P]	Shut down	*illegal_instruction*
STBLOCKF	64-byte block store with commit	*data_access_exception*
STQF	Store quad floating-point	*illegal_instruction*
STQFA	Store quad floating-point into alternate space	*illegal_instruction*
STSHORTF	Short FP store, 8-/16-bit store from a double-precision floating-point register	*data_access_exception*

5.2 Prefetch and Prefetch from Alternate Space

PREFETCH and PREFETCHA with fcn codes of 0–3 and 16–23 (10_{16}–17_{16}) are implemented; all map to the same operation that brings the cache line into the L2 cache. On an MMU miss or when the MMU is completely bypassed (PA{39:0} is set to VA{39:0}), the prefetch is dropped (weak prefetching).

Prefetch fcn codes 5_{16}–F_{16} cause an *illegal_instruction* trap. These operations are all "weak" prefetches; in some cases (on an MMU miss) the prefetch operation is dropped.

Note | Prefetches to I/O space (PA{39} = 1) are dropped by UltraSPARC T1.

Note | Since hypervisor accesses normally bypass the MMU, prefetches in hypervisor mode are generally NOPs. It is possible to get a hypervisor prefetch, by using PREFETCHA with `ASI_REAL` or `ASI_AS_IF_USER*`, to force MMU non-bypass.

5.3 Trap on Integer Condition Codes (Tcc)

See the *UltraSPARC Architecture 2005* for a complete description of the Tcc instruction.

UltraSPARC T1 Implementation Note | For the i = 0 variant of Tcc, UltraSPARC T1 does not check that reserved instruction bit 7 is 0. If bit 7 is set to 1 with i = 0, UltraSPARC T1 treats it as a valid Tcc instruction.

5.4 VIS Instructions

UltraSPARC T1 supports in hardware the VIS 2 SIAM instruction and a subset of the VIS 1 instructions.

All other VIS 1 and VIS 2 instructions (see TABLE 5-2 for a list) cause an *illegal_instruction* exception on UltraSPARC T1 and are emulated in software.

The use of VIS instructions on UltraSPARC T1 is strongly discouraged; the performance of even the implemented VIS instructions will often be below that of a comparable set of non-VIS instructions. This includes the block load and block store instructions. An UltraSPARC T1 physical processor core (four virtual processors) can only have a single outstanding floating-point operation (including block load, block store, and VIS instructions) in progress at any given time.

5.5 Partitioned Add/Subtract Instructions

See the *UltraSPARC Architecture 2005* for detailed descriptions of the FPADD and FPSUB instructions.

For good performance on UltraSPARC T1, the result of a single FPADD should not be used as part of a 64-bit graphics instruction source operand in the next instruction group.

Similarly, the result of a standard FPADD should not be used as a 32-bit graphics instruction source operand in the next instruction group.

5.6 Align Data

See the *UltraSPARC Architecture 2005* for detailed descriptions of the FALIGNDATA instruction.

For good performance on UltraSPARC T1, the result of FALIGNDATA should not be used as the source operand of a 32-bit SIMD instruction in the next instruction group.

5.7 F Register Logical Operate Instructions

See the *UltraSPARC Architecture 2005* for a description of the F register logical operate instructions (1-, 2-, and 3-operand).

For good performance on UltraSPARC T1, the result of a single logical operate instruction should not be used as part of the source operand of a 64-bit SIMD instruction in the next instruction group.

Similarly, the result of a standard logical operate instruction should not be used as the source operand of a 32-bit SIMD instruction source operand in the next instruction group.

5.8 Block Load and Store Instructions

For architectural descriptions of the LDBLOCKF and STBLOCKF instructions, see the *UltraSPARC Architecture 2005*.

UltraSPARC T1 Implementation Note On UltraSPARC T1, a block load forces a miss in the primary cache and will *not* allocate a line in the primary cache, but does allocate in the L2 cache. On UltraSPARC T1, block loads and stores from multiple virtual processors are not overlapped.

Compatibility Note These instructions were intended for use in transferring large blocks of data (more than 256 bytes); for example, in BCOPY and BFILL operations.

The use of block loads and stores on UltraSPARC T1 is deprecated; they are provided primarily for compatibility with existing software. UltraSPARC T1 provides a separate set of ASIs for high performance BCOPY and BFILL, as described in TABLE 9-3 on page 69. The performance of *parallel* BCOPY using appropriate ASIs (from among 22_{16}, 23_{16}, $E2_{16}$, $E3_{16}$, EA_{16}, and EB_{16}) will be 2.5 to 3.5 times that of a BCOPY using block loads and stores. The performance of a *single-threaded* BCOPY using these ASIs will be 15% to 50% better than that of a BCOPY using block loads and stores.

On UltraSPARC T1, to order an LDBLOCKF with respect to earlier stores, an intervening MEMBAR #Sync must be executed.

Similarly on UltraSPARC T1, STBLOCKF source data registers are not interlocked against completion of previous load instructions (even if a second LDBLOCKF has been performed). The previous load data must be referenced by some other intervening instruction, or an intervening MEMBAR #Sync must be performed. If the programmer violates these rules, data from before or after the load may be used. UltraSPARC T1 continues execution before all of the store data has been transferred. If store data registers are overwritten before the next block store or MEMBAR #Sync

instruction, then the following rule must be observed. The first register can be overwritten in the same instruction group as the STBLOCKF, 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 store correct data or the overwritten data. Block stores always operate under the relaxed memory order (RMO) memory model, regardless of the PSTATE.mm setting, and require a subsequent MEMBAR #Sync to order them with respect to following loads.

After an STBLOCKF instruction but before executing a DONE, RETRY, or WRPR to PSTATE instruction, there must be an intervening MEMBAR #Sync or a trap. If this is rule is violated, instructions after the DONE, RETRY, or WRPR to PSTATE may not see the effects of the updated PSTATE.

On UltraSPARC T1, LDBLOCKF 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 *Translation Table Entry (TTE)* on page 182). 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 (for example, 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 LDBLOCKF overlaps a previous or later store and there is no intervening MEMBAR, trap, or data reference, the LDBLOCKF may return data from before or after the store.

Compatibility Note | Prior UltraSPARC machines may have written loaded data into the first two registers at the same time. Software that depends on this unsupported behavior must be modified for UltraSPARC T1.

STBLOCKF 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 STBLOCKF 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 STBLOCKF overlaps a later load and there is no intervening trap or MEMBAR #StoreLoad instruction, the contents of the block are undefined. If the STBLOCKF 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 virtual processor memory model (always TSO in UltraSPARC T1); 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 strand; that is, no read-after-write or writer-after-read checking occurs between block loads and stores. Explicit MEMBARs must be used to enforce dependence ordering between block operations that reference the same address.

Typically, LDBLOCKF and STBLOCKF 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 must be preceded and followed by the appropriate MEMBARs to ensure that there are no hazards with loads and stores outside the loops. CODE EXAMPLE 5-1 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 register references in this code example are to 64-bit registers. Eight versions of this loop are needed to handle all the cases of double word misalignment between the source and destination.

CODE EXAMPLE 5-1   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
```

5.9 Block Initializing Store ASIs

The Block Initializing Store ASIs are specific to the UltraSPARC T1 implementation and are not guaranteed to be portable to other UltraSPARC Architecture implementations. They should only appear in platform-specific dynamically-linked libraries, hyperprivileged software, or in code generated at runtime by software (for example, a just-in-time compiler) that is aware of the specific implementation upon which it is executing.

Instruction	imm_asi	ASI Value	Operation	Assembly Language Syntax
ST{B,H,W,X,D}A	ASI_STBI_AIUP	22_{16}	64-byte block initialing store to primary address space, user privilege	st{b,h,w,x,d}a reg_{rd}, [reg_addr] imm_asi st{b,h,w,x,d}a reg_{rd}, [reg_plus_imm] %asi
ST{B,H,W,X,D}A	ASI_STBI_AIUS	23_{16}	64-byte block initialing store to secondary address space, user privilege	
ST{B,H,W,X,D}A	ASI_STBI_N	27_{16}	64-byte block initialing store to nucleus address space	
ST{B,H,W,X,D}A	ASI_STBI_AIUPL_L	$2A_{16}$	64-byte block initialing store to primary address space, user privilege, little-endian	
ST{B,H,W,X,D}A	ASI_STBI_AIUSL	$2B_{16}$	64-byte block initialing store to secondary address space, user privilege, little-endian	
ST{B,H,W,X,D}A	ASI_STBI_NL	$2F_{16}$	64-byte block initialing store to nucleus address space, little-endian	
ST{B,H,W,X,D}A	ASI_STBI_P	$E2_{16}$	64-byte block initialing store to primary address space	
ST{B,H,W,X,D}A	ASI_STBI_S	$E3_{16}$	64-byte block initialing store to secondary address space	
ST{B,H,W,X,D}A	ASI_STBI_PL	EA_{16}	64-byte block initialing store to primary address space, little-endian	
ST{B,H,W,X,D}A	ASI_STBI_SL	EB_{16}	64-byte block initialing store to secondary address space, little-endian	

Description   The UltraSPARC T1-specific block initializing store instructions are selected by using one of the block-initializing ASIs with integer store alternate instructions. These ASIs allow block-initializing stores to be performed to the same address spaces as normal stores. Little-endian ASIs access data in little-endian format; otherwise, the access is assumed to be big-endian.

Integer stores of all sizes are allowed with these ASIs, and STDA behaves as a standard store doubleword. All stores to these ASIs operate under relaxed memory ordering (RMO), regardless of the value of PSTATE.mm. Software must follow a sequence of these stores with a MEMBAR #Sync to ensure ordering with respect to subsequent loads and stores.

A store to one of these ASIs where the least-significant 6 bits of the address are nonzero (that is, not the first word in the cache line) behaves the same as a normal store (with RMO ordering).

A store to one of these ASIs where the least-significant 6 bits of the address are zero will load a cache line in the L2 cache with either all zeros or the existing memory data, and then update the beginning of the cache line with the new store data. This special store behavior ensures that the line maintains coherency when it is loaded into the cache, but will not generally fetch the line from memory (instead, initializing it with zeroes).

A store using one of these ASIs to a noncacheable location behaves the same as a normal store.

UltraSPARC T1 | On UltraSPARC T1, a noncacheable address is identified by
Implementation | PA{39} = 1.
Note |

Programming | These instructions are particularly useful in combination with
Note | load twin extended word instructions for transferring large
| blocks (more than 256 bytes) of data; for example, in
| implementing bcopy() and bfill() operations.

UltraSPARC T1 | On UltraSPARC T1, block initializing stores and load twin
Implementation | doublewords from multiple strands are fully overlapped.
Note |

Attempted use of any of these ASIs by a floating-point store alternate instruction (STFA, STDFA) causes a *data_access_exception* exception.

Access to any of these ASIs by an instruction with misaligned address causes a *mem_address_not_aligned* exception.

The following pseudocode shows how these ASIs can be used to do a quadword-aligned (on both source and destination) copy of *N* quadwords from *A* to *B* (where *N* > 3). Note that the final 64 bytes of the copy is performed using normal stores, to guarantee that all initial zeros in a cache line are overwritten with copy data.

```
%l0 ← [A]; %l1 ← [B]
prefetch [%l0]
for (i = 0;  i < N-4;  i++) {
   if (!(i % 4)) { prefetch [%l0+64] }
   ldda [%l0] #ASI_BLK_INIT_ST_P, %l2
   add %l0, 16, %l0
   stxa %l2, [%l1] #ASI_BLK_INIT_ST_P
   add %l1, 8, %l1
   stxa %l3, [%l1+8] #ASI_BLK_INIT_ST_P
   add %l1, 8, %l1
}
for (i = 0;  i < 4;  i++) {
   ldda [%l0] #ASI_BLK_INIT_ST_P, %l2
   add %l0, 16, %l0
   stx %l2, [%l1]
   stx %l3, [%l1+8]
   add %l1, 16, %l1
}
membar #Sync
```

An overlapped copy operation must avoid issuing a block-init store to a line before all loads from that line have been issued. Otherwise, one or more of the loads may see the interim "zero" side-effect value. This typically means that **abs**(A–B) must be 64.

| **UltraSPARC T1 Programming Notes** | (1) These ASIs are specific to UltraSPARC T1, to provide a high-performance mechanism for BCOPY operations, as an alternative to legacy block load and block store instructions (which rely on the floating-point register file and thus are limited by the single register file port). These ASIs are "Class N"(nonportable) features, so are only allowed in platform-specific dynamically linked libararies, in hyperprivileged code, and in code generated at runtime by software (for example, a just-in-time compiler) that is aware of the implementation upon which it is executing. |
| | (2) These ASIs provide a higher performance `bcopy()` or `bfill()` than the block loads and stores described in Section 5.8, due to their ability to overlap multiple loads and stores between strands and to avoid the unnecessary fetch from memory of the data that is overwritten by the store. The performance of parallel `bcopy()` using these ASIs will be 2.5 to 3.5 times that of a `bcopy()` using block loads and stores. The performance of a single-threaded `bcopy()` using these ASIs will be 15% to 50% better than that of a `bcopy()` using block loads and stores. |

Exceptions   *VA_watchpoint*
mem_address_not_aligned
data_access_exception

5.10 Load Twin Extended Word Instructions (nonprivileged)

> The Load Twin Extended Word Instructions are not guaranteed to be portable to other UltraSPARC Architecture implementations. They should only appear in platform-specific dynamically-linked libraries, hyperprivileged software, or in code generated at runtime by software (for example, a just-in-time compiler) that is aware of the specific implementation upon which it is executing.

Description   Load Twin Extended Word instructions are new in the UltraSPARC Architecture 2005; they are used to atomically read a 128-bit data item into a pair of integer registers.

See the *UltraSPARC Architecture 2005* for details.

Programming Note | These instructions are particularly useful in combination with block-initializing stores for transferring large blocks of data (more than 256 bytes); for example, in implementing bcopy() and bfill() operations. See the description of Block Initializing Stores for an example of how Load Twin Extended Word can be used in combination with those instructions.

UltraSPARC T1 Implementation Note | On UltraSPARC T1, a load twin extended word forces a miss in the primary cache and will *not* allocate a line in the primary cache, but does allocate in L2. On UltraSPARC T1, block initializing stores and load twin doublewords from multiple strands are fully overlapped.

See the description of Block Initializing Stores on page 31 for an example of how Load Twin Extended Word can be used in combination with those instructions.

UltraSPARC T1 Programming Notes

(1) These instructions, combined with store instructions using the UltraSPARC T1-specific Block Initializing Store ASIs, provide a high-performance mechanism for BCOPY operations, as an alternative to legacy block load and store (which rely on the floating-point register file and thus are limited by the single register file port). These ASIs are "Class N" (nonportable) features and are only allowed in platform-specific dynamically linked libararies and in code generated at runtime by software (for example, a just-in-time compiler) that is aware of the implementation upon which it is executing.

(2) These ASIs provide a higher performance `bcopy()` or `bfill()` than the block loads and stores described in Section 5.8, due to their ability to overlap multiple loads and stores between strands and to avoid the unnecessary fetch from memory of the data that is overwritten by the store. The performance of parallel `bcopy()` using these ASIs will be 2.5 to 3.5 times that of a `bcopy()` using block loads and stores. The performance of a single-threaded `bcopy()` using these ASIs will be 15% to 50% better than that of a `bcopy()` using block loads and stores.

See Also   Block Initializing Store ASIs on page 31.

5.11 Load Twin Extended Word Instructions (privileged)

Instruction	imm_asi	ASI Value	Operation	Assembly Language Syntax
LDTX	ASI_LDTX_N	27_{16}†	128-bit atomic load	ldda [*reg_addr*] *imm_asi*, *reg_rd* ldda [*reg_plus_imm*] %asi, *reg_rd*
LDTX	ASI_LDTX_REAL	26_{16}	128-bit atomic load, real addressing (RA{63:0} set to VA{63:0})	
LDTX	ASI_LDTX_NL	$2F_{16}$‡	128-bit atomic load, little endian	
LDTX	ASI_LDTX_REAL_L	$2E_{16}$	128-bit atomic load, real addressing (RA{63:0}) set to VA{63:0}), little endian	

† ASI 24_{16} (deprecated) is aliased to ASI 27_{16} in UltraSPARC T1.
‡ ASI $2C_{16}$ (deprecated) is aliased to ASI $2F_{16}$ in UltraSPARC T1.

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
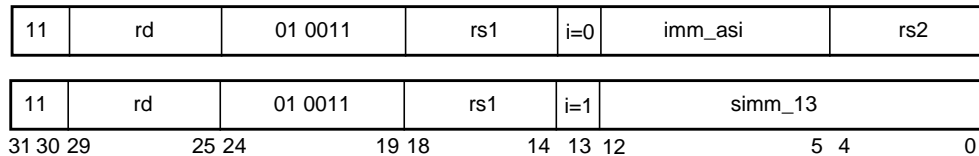
Compatibility Note | In previous UltraSPARC documents, these instructions were (loosely) referred to as "Quad LDD" instructions.

Description   These instructions atomically read a 128-bit data item into two 64-bit integer registers. 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-numbered register; the highest address 64-bits is placed in the odd-numbered register.

The access will *not* allocate a line in the primary cache on a primary miss, regardless of the setting of the cp bit in the TLB entry. ASI_LDTX_REAL{_L} bypasses the virtual-to-real portion of the translation, setting RA{63:0} = VA{63:0}. When the access to these ASIs bypasses the TLB, the physical address is set equal to the truncated virtual address (that is, PA{39:0} = VA{39:0}). If PA{39} = 0, the physical page attribute bit w is set to 1 and all other attribute bits are set to 0 for the bypass. If PA{39} = 1, the physical page attributes e and w are set to 1 and all other attribute bits are set to 0 for the bypass.

In addition to the usual exceptions for LDTX using a privileged ASI, a *data_access_exception* trap occurs if these ASIs are used with any instruction other than LDTX or LDDA (which share an opcode). A *mem_address_not_aligned* trap is taken if the access is not aligned on a 128-bit boundary.

Exceptions *VA_watchpoint*
mem_address_not_aligned (Checked for opcode implied alignment if the opcode is not LDDA)
data_access_exception

Traps

The UltraSPARC T1 processor implements the trap model described in the
UltraSPARC Architecture 2005.

Additional UltraSPARC T1-specific details are described in this chapter.

6.1 Trap Levels

Each UltraSPARC T1 virtual processor supports six trap levels (*MAXTL* = 6) Traps to
privileged mode while in privileged mode when TL = *MAXPTL* will trap instead to the
hyperprivileged mode using the *watchdog_reset* vector in the hyperprivileged trap
table, incrementing the TL, but will not enter RED_state, and the trap type will be
set to that of the trap that caused the error, not the watchdog trap type.

6.2 Traps to Hyperprivileged Mode

TABLE 6-1 lists the additional traps for support of hyperprivileged software. The
htrap_instruction trap is generated by execution of the HTrap on Integer Condition
Codes instruction, which is similar to the Trap on Integer Condition Codes
instruction, but with bit 7 of the software trap number set to 1. Execution of an
HTrap on Integer Condition Codes instruction by user code results in an
illegal_instruction trap.

TABLE 6-1　Additional Hypervisor Support Traps

TT #	Hardware Trap Name	Priority	Description
$7C_{16}$	*cpu_mondo_trap*	16	Disrupting
$7D_{16}$	*dev_mondo_trap*	16	Disrupting
$7E_{16}$	*resumable_error*	33	Disrupting

6.3　Implementation-Dependent Exceptions

TABLE 6-2 lists the UltraSPARC T1 implementation-dependent exceptions.

TABLE 6-2　UltraSPARC T1 Implementation-Dependent Exceptions

TT #	Hardware Trap Name	Priority	Description
074_{16}	*modular_arithmetic_interrupt*	16	Disrupting
060_{16}	*interrupt_vector*	26.3	Disrupting
078_{16}	*data_error*	13	Disrupting

UltraSPARC T1 implements a *modular_arithmetic_interrupt* for its modular arithmetic unit discussed in Chapter 16, *Modular Arithmetic*.

> **Implementation Note**　UltraSPARC T1 implements the *instruction_breakpoint* trap, which is documented in UltraSPARC Architecture. The trap is priority 7 (with slightly greater priority than *illegal_instruction*), and, when taken, clears the HPSTATE.ibe bit. The UltraSPARC T1 implementation does not follow the UltraSPARC Architecture specification in that HPSTATE.ibe is cleared only when an *instruction_breakpoint* trap is taken, regardless of mode. UltraSPARC Architecture specifies that HPSTATE.ibe is cleared on any trap to hyperprivileged mode, while remaining unchanged on traps to privileged mode.

UltraSPARC T1 implements precise and disrupting traps to handle catastrophic error exceptions on data accesses from/to memory (the SPARC V9 trap *data_access_error*). To help software distinguish between the precise and disrupting versions of the trap, a precise catastrophic memory error causes a *data_access_error*, while a disrupting catastrophic memory error causes a priority 13 *data_error* trap.

6.4 Trap Behavior

Traps on UltraSPARC T1 behave as specified in the *UltraSPARC Architecture 2005*.

6.5 Trap Masking

Traps are masked as specified in the *UltraSPARC Architecture 2005*.

Interrupt Handling

7.1 Overview

The chapter describes the hardware interrupt delivery mechanism for the UltraSPARC T1 chip.

Software interrupts are delivered to each virtual processor using the *interrupt_level_n* traps (41_{16}–$4F_{16}$) through the SOFTINT register (described in *Software Interrupt Register (SOFTINT)* on page 13). Details on software interrupt generation and servicing are available in the UltraSPARC Architecture specification.

I/O and CPU cross-call interrupts are delivered to each virtual processor using the *interrupt_vector* trap (60_{16}). *interrupt_vector* traps have a corresponding 64-bit ASI_SWVR_INTR_RECEIVE register. I/O devices and CPU cross-call interrupts contain a 6-bit identifier, which determines which interrupt vector (level) in the ASI_SWVR_INTR_RECEIVE register the interrupt will target. Each strand's ASI_SWVR_INTR_RECEIVE register can queue up to 64 outstanding interrupts, one for each interrupt vector. Interrupt vectors are implicitly prioritized with vector 63 having the highest priority and vector 0 being the lowest priority.

Each I/O interrupt source has a hardwired interrupt number, which is used to index a table of interrupt vector information (INT_MAN) in the I/O bridge (IOB). Generally, each I/O interrupt source will be assigned a unique virtual processor target and vector level. This association is defined by software programming of the interrupt vector and vc_id fields in the INT_MAN table in the IOB. Software must maintain the association between interrupt vector and hardware interrupt number to index the appropriate entry in the INT_MAN and INT_CTL tables.

7.2 Interrupt Flow

This section covers the interrupt flow for I/O and CPU cross-call interrupts.

7.2.1 Initialization

The interrupt vector traps are initialized by writing 0_{16} to Interrupt Receive Register described in Section 7.4.1.

I/O interrupt handling is initialized as follows:

1. Specifying the virtual processor/vector pair to receive "internal" IO interrupts by programming the INT_MAN table, described in *Interrupt Management Registers*.

2. Specifying the interrupt vector to receive external "mondo" interrupts by programming the J_INT_VEC register, described in *Interrupt Management Registers*.

3. Clearing the mask bits in the INT_CTL table, described in *Interrupt Management Registers*.

4. Clearing the busy bits in the J_INT_BUSY table, described in *Interrupt/Trap Vector Dispatch Register* on page 49.

7.2.2 Servicing

Interrupt vector traps are typically serviced by reading the Incoming Vector register described in Section 7.4.3. When this register is read by software, the 6-bit vector corresponding to the highest priority pending interrupt in the interrupt receive register is returned. The pending interrupt bit for that vector is cleared.

If the incoming interrupt matches the vector for "mondo" interrupts, the handler should then read the mondo source and data, from the J_INT_ADATA0/1 registers (described in TABLE 7-4 on page 49) and J_INT_ABUSY registers (described in *Mondo Interrupt Busy Table*). After reading these registers, it should enable receiving the next mondo interrupt to this virtual processor by clearing the busy bit in the J_INT_ABUSY register.

If the incoming interrupt matches the virtual processor and vector for internal error interrupts (device ID = 1), the handler should read the JBUS and SSI error logs, described in *JBI Error Registers* on page 173 and *SSI Error Registers* on page 187, plus check the DRAM Error Counter register, described in *DRAM Error Counter Register* on page 163, to determine the cause of the interrupt. It should service the interrupt appropriately, then clear the mask bit in the corresponding INT_MAN register.

If the incoming interrupt matches the virtual processor and vector for SSI interrupts (device ID = 2), the handler should service the interrupt appropriately, then clear the mask bit in the corresponding INT_MAN register.

7.2.3 Sources

CPU cross-call interrupts can be generated by writing the Interrupt Vector Dispatch register described in Section 7.4.2. Dispatching CPU cross-call interrupts is described in *Dispatching* on page 47.

TAP interrupts can be generated by writing the IOB Interrupt Vector/Trap Dispatch register described in Section 7.3.2.

Internal error interrupts (device ID = 1) are caused by JBI- and SSI-detected errors, as described in *IOP Error Summary* on page 187, plus DRAM Error Counter Underflow, as described in *DRAM Error Counter Register*.

SSI interrupts (device ID = 2) are caused by an assertion (edge trigger) on the EXT_INT_L pin.

JBUS mondo interrupts are caused by receiving an INT transaction on JBUS.

| **Programming Note** | The vectors used by interrupts are completely decided by software convention. However, since there is no ability to detect multiple simultaneous interrupts in the Interrupt Receive register, it is recommended that each potential interrupt source have a dedicated vector (bit) in the Interrupt Receive register. |

7.2.4 States

Each bit in the Interrupt Receive register can be in one of two states: set or cleared. If an incoming interrupt attempts to set an already set bit, the additional incoming interrupt will be lost (there is no overflow indication on the interrupt bits). Writes to the Interrupt Receive register will clear any bit in which the corresponding write data is 0, and reads to the Incoming Vector register will clear the bit of the highest-priority pending interrupt as a side-effect. If another interrupt attempts to set a bit on the same cycle as the bit is being cleared by an Interrupt Receive register write or Incoming Vector register read, the additional interrupt will take precedence over the clear and the bit will remain set.

JBUS interrupts have two states in the J_INT_BUSY table, namely, BUSY and IDLE (not BUSY). When a mondo INT transaction is received, if the current state is IDLE, the transaction is accepted (and acknowledged to the requestor by an ACK signal), state is changed to BUSY, the mondo data is stored in the J_INT_DATA0/1 table, and the bit specified by j_int_vec is set in the Interrupt Receive register of the virtual processor specified in the INT transaction. While in the BUSY state, any INT

transactions to the same virtual processor will be rejected and a negative-acknowledged (NACK) response sent back to the requestor, and the requestor is responsible for preserving that interrupt in a pending state. When software has adequately serviced the interrupt, it explicitly clears the busy bit in the J_INT_BUSY register to return to the IDLE state.

"Internal" I/O interrupts have three states in the INT_CTL table, namely IDLE, MASK, and PENDING. When an SSI or error interrupt is received, if the current state is IDLE, the state is changed to BUSY, and the bit specified by INT_MAN is set in the Interrupt Receive register of the virtual processor specified in INT_MAN. While in the BUSY state, another interrupt of the same type will set the pend bit in the INT_CTL table, going to the PENDING state. Further interrupts while in the PENDING state will set the pend bit again, having no real effect. When software has adequately serviced the interrupt, it explicitly clears the mask bit in the INT_CTL table, which will return it from the MASK state to the IDLE state (or will enable a pending interrupt to be taken so it would go to the MASK state from the PENDING state).

7.2.5 Prioritizing

Interrupt vector traps are implicitly prioritized by the Incoming Vector register described in Section 7.4.3 from bit 63 (highest) to bit 0 (lowest).

The priority of I/O interrupts is done by specifying the vector value in the INT_MAN table and in J_INT_VEC (see Section 7.3.1).

7.2.6 Dispatching

CPU cross-call interrupts can be generated by writing the Interrupt Vector Dispatch register described in *Interrupt Vector Dispatch Register* on page 53. Interrupts are always received by the destination and stores to this register will follow the TSO memory model (no MEMBAR #Sync is required). The store data supplies the destination virtual processor and vector. The bit corresponding to the specified vector is set in the interrupt receive register of the destination virtual processor. CPU cross-call interrupts can also be generated via the IOB Interrupt/Trap Vector Dispatch register described in Section 7.3.2

7.3 IOB Interrupt Registers

The following registers are defined for interrupt and reset management. The base address is defined below.

RegisterBaseAddress 1 IOBMAN – 98 0000 0000$_{16}$. The IO Bridge handles two types of interrupts: those generated on chip and those generated externally through JBUS.

TABLE 7-1 lists the device ID assignment.

TABLE 7-1   Device ID Assignments

Device	ID Range	Comment
Reserved	0	
Uncorrectable Error or correctable Error Count Overflow Interrupt	1	Used by several devices for errors. Not necessarily fatal in that the block will stop working, but indicates that the error is serious.
SSI Interrupt	2	SSI interrupt from EXT_INT_L pin.
Reserved	3	

On chip interrupt hardware contains an Interrupt Management Table and an Interrupt Control Table. Each internal "Device ID" in the I/O subsystem has an entry in each.

Device ID 0 is used internally by hardware, but is architecturally reserved.

Device ID 1 is used to report fatal and uncorrectable errors, including the rollover count of correctable ECC errors. Software will have to poll the detailed error states of different devices to determine the error type.

Device ID 2 is the interrupt from EXT_INT_L pin, of the SSI interface, which is intended for use as a console interrupt.

7.3.1   Interrupt Management Registers

The Interrupt Management registers specify the CPU ID to send the interrupt and the interrupt vector associated with the interrupt issued by the IOB on behalf of the device.

Each device will send its device ID to the I/O bridge. The device ID is used to index into the Interrupt Management table. The Interrupt Control register enables software to mask the interrupt individually. Software can write a 1 to the mask bit to block incoming interrupts. If the mask bit is 0 when an interrupt occurs, hardware will set the bit to 1 and block future interrupts. This will guarantee only a single interrupt per device will be issued to the CPU. Note that when the mask bit is set, the specific interrupt will not be issued, but rather the pending bit will be set. Software needs to reset the mask bit to zero to enable interrupts again. Before changing the Interrupt

Management register, software must set the device **mask** bit first. Note that the offset address of the corresponding device can be calculated by multiplying the device ID by 8 for **INT_MAN**, and add 400_{16} to 8 times the device ID for **INT_CTL**.

TABLE 7-2 shows the format of the Interrupt Management register.

TABLE 7-2   30 Interrupt Management – **INT_MAN** Register (0000_{16})

Bit	Field	Initial Value	R/W	Description
63:13	—	0	R	*Reserved*
12:8	cpu	X	rW	Virtual processor ID to manage the device.
7:6	—	0	R	*Reserved*
5:0	vector	X	RW	Interrupt Vector (encodes bit set in `ASI_SWVR_INTR_RECEIVE`).

TABLE 7-3 shows the format of the Interrupt Control register.

TABLE 7-3   31 Interrupt Control – **INT_CTL** Register (0400_{16})

Bit	Field	Initial Value	R/W	Description
63:3	—	0	R	*Reserved*
2	mask	1	RW	Set to 1 to mask the interrupt. If **mask** is zero when interrupt occurs, hardware will set it to 1 to block future interrupts.
1	clear	0	W	Write 1 to clear the pending bit, 0 to leave pending bit unchanged.
0	pend	0	R	Set to 1 if there is a pending interrupt, reset to 0 if the **clear** bit is set or after hardware issues the pending interrupt once the **mask** bit is cleared.

After setting the **mask** bit, software needs to issue a read on the **INT_CTL** register to guarantee the masking write is completed. Before the read completes there may be an interrupt in transit to the CPU.

Note that the **clear** field of **INT_CTL** is intended mainly for error cleanup, as clearing the pending bit via the **clear** field will cause software to miss both the pending interrupt and any possible additional interrupts that may arrive at the same time the pending bit is cleared.

The following register, the format of which is shown in TABLE 7-4, specifies the interrupt vector for JBUS Mondo interrupts and are shared among the 32 virtual processors.

TABLE 7-4   JBUS Interrupt Vector Register – J_INT_VEC (0000_{16}–$0A00_{16}$)

Bit	Field	Initial Value	R/W	Description
63:6	—	0	R	*Reserved*
5:0	vector	X	RW	Interrupt Vector for Mondo interrupts (encodes bit set in `ASI_SWVR_INTR_RECEIVE`).

J_INT_VEC performs the identical function for JBUS Mondo interrupts that INT_MAN performs for other I/O interrupts, except that the virtual processor ID is specified in the mondo interrupt transaction.

7.3.2   Interrupt/Trap Vector Dispatch Register

A strand may write to the following register to trigger an interrupt to another virtual processor. In addition, any virtual processor may be placed in the idle state, restarted from the halted or idle state, or sent a reset via this register.

TABLE 7-5 shows the format of the Interrupt Vector Dispatch register.

TABLE 7-5   32 Interrupt Vector Dispatch Register – INT_VEC_DIS (0800_{16})

Bit	Field	Initial Value	R/W	Description
63:18	—	X	R	*Reserved*
17:16	type	X	W	00 = interrupt, 01 = reset, 10 = idle, 11 = resume.
15:13	—	X	R	*Reserved*
12:8	vc_id	X	W	Destination virtual processor.
7:6	—	X	R	*Reserved*
5:0	vector	X	W	Interrupt Vector (encodes bit set in `ASI_SWVR_INTR_RECEIVE`) or Reset Trap Type (TT #). This field is unused if type is "idle" or "resume".

7.3.3   JBUS Mondo Data Tables

The following registers are used to manage the JBUS interrupts.

When the IO Bridge receives an interrupt, it sets the busy bit and acknowledges the interrupt. When the busy bit is set, it means an interrupt is waiting to be serviced or is being serviced. Software will reset the busy bit when it completes servicing the interrupt. If the busy bit is already set when an interrupt is received, a negative-acknowledged response (NACK) will be sent to the JBUS unit. The busy bit is set after a reset and software has to clear it to begin receiving interrupts. This register also indicates the source of the interrupt.

The base address of the interrupt table registers is defined below.

RegisterBaseAddress IOBINT – 9F 0000 0000$_{16}$. There are two JBUS Interrupt Mondo tables. The tables are read-only by software and the entries are updated by JBUS interrupts provided that the interrupt is not busy. The IO Bridge will acknowledge (ACK) the interrupt if it is not busy; otherwise, the IO Bridge will send a negative-acknowledged (NACK) response it.

TABLE 7-6 shows the format of the JBUS Interrupt Mondo Data 0 Table.

TABLE 7-6 JBUS Interrupt Mondo Data 0 – J_INT_DATA0 (0000$_{16}$–0400$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:0	data_0	X	R	First 64 bits of JBUS interrupt mondo data.

TABLE 7-7 shows the format of the JBUS Interrupt Mondo Data 1 Table.

TABLE 7-7 JBUS Interrupt Mondo Data 1 – J_INT_DATA1 (0000$_{16}$–0500$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:0	data_1	X	R	Second 64 bits of JBUS interrupt mondo data.

TABLE 7-8 shows the format of the JBUS Interrupt Alias Mondo Data 0 Table.

This register address is actually an alias for J_INT_DATA0[*Current_Virtual Processor*], so each virtual processor can read its own interrupt payload, without having to do an address calculation based on vc_id. This address should never be accessed by the TAP (since it does not have a vc_id).

TABLE 7-8 JBUS Interrupt Alias Mondo Data 0 – J_INT_DATA0 (0000$_{16}$–0600$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:0	data_0	X	R	First 64 bits of JBUS interrupt mondo data.

TABLE 7-9 shows the format of the JBUS Interrupt Alias Mondo Data 1 Table.

This register address is actually an alias for J_INT_DATA1[*Current_Virtual Processor*], so each virtual processor can read its own interrupt payload, without having to do an address calculation based on vc_id. This address should never be accessed by the TAP (since it does not have a vc_id).

TABLE 7-9 JBUS Interrupt Alias Mondo Data 1 – J_INT_ADATA1 (0000$_{16}$–0700$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:0	data_1	X	R	Second 64 bits of JBUS interrupt mondo data.

7.3.4 Mondo Interrupt Busy Table

TABLE 7-10 shows the format of the JBUS Interrupt Busy Table.

TABLE 7-10 JBUS Interrupt Busy –J_INT_BUSY (0000_{16}–0900_{16}) (

Bit	Field	Initial Value	R/W	Description
63:6	—	0	R	*Reserved*
5	busy	X	RW	Hardware sets busy to 1 when an interrupt is received. Hardware nacks an incoming interrupt if busy is set.
4:0	source	X	R	Hardware updates this when the interrupt is acknowledged. It contains the source ID supplied by the JBUS.

TABLE 7-11 shows the format of the JBUS Interrupt Busy Alias Table.

This register address is actually an alias for J_INT_BUSY[*Current_Virtual Processor*], so each virtual processor can update its own mondo interrupt busy bit, without having to do an address calculation based on vc_id. This address should never be accessed by the TAP (since it does not have a vc_id).

TABLE 7-11 JBUS Interrupt Busy – J_INT_ABUSY (0000_{16}–$0B00_{16}$)

Bit	Field	Initial Value	R/W	Description
63:6	—	0	R	*Reserved*
5	busy	X	RW	Hardware sets busy to 1 when an interrupt is received. Hardware nacks an incoming interrupt if busy is set.
4:0	source	X	R	Hardware updates this when the interrupt is acknowledged. It contains the source ID supplied by the JBUS.

7.4 CPU Interrupt Registers

7.4.1 Interrupt Receive Register

Each virtual processor (strand) has a hyperprivileged ASI_SWVR_INTR_RECEIVE register at ASI 72_{16}, VA{63:0} = 0. Each time an interrupt transaction arrives for that strand, the bit corresponding to the interrupt vector will be set. Bit zero of the register corresponds to interrupt vector number zero and so on. Interrupt vectors are implicitly prioritized with vector number 63 being the highest priority and vector number 0 being the lowest priority. Software writes to this register are **and**ed with the register contents to allow the software to selectively clear register bits, although

normally the incoming vector register described in Section 7.4.3 will be used. When an interrupt arrives at the same time as a register write, the interrupt will take precedence over the write and the bit will be set. Software can read this register to determine all pending interrupts, although normally the incoming vector register will be used. Nonprivileged access to this register causes a *privileged_action* trap. Privileged access to this register causes a *data_access_exception* trap.

Implementation Note	There is no double-buffering of interrupt bits. If multiple interrupt transactions arrive for the same interrupt vector, the corresponding bit will stay set, and the fact that multiple interrupts were received will be lost.

TABLE 7-12 defines the format of the ASI_SWVR_INTR_RECEIVE register.

TABLE 7-12   Interrupt Receive Register – ASI_SWVR_INTR_RECEIVE (ASI 72_{16}, VA 0_{16})

Bit Position	Field	Initial Value	R/W	Description
63:0	pending	X	RW	Pending interrupts

7.4.2   Interrupt Vector Dispatch Register

Each strand has a hyperprivileged write-only ASI_SWVR_UDB_INTR_W register at ASI 73_{16}, VA{63:0} = 0 that is used to send CPU cross-call interrupts to other virtual processors. Interrupts are always received by the destination and stores to this register will follow the TSO memory model (no MEMBAR #Sync is required).

The store data supplies an identifier for the destination virtual processor and vector. The bit corresponding to the specified vector is set in the Interrupt Vector Receive register of the destination virtual processor. The format of the register is shown in TABLE 7-13.

TABLE 7-13   Interrupt Vector Dispatch Register – ASI_SWVR_UDB_INTR_W (ASI 73_{16}, VA 0_{16})

Bit	Field	Initial Value	R/W	Description
63:18	—	0	R	*Reserved*
17:16	type	X	W	00 = interrupt; 01–11 *reserved* (alias to interrupt in UltraSPARC T1).
15:13	—	0	R	*Reserved*
12:8	vc_id	X	W	Destination virtual processor.
7:6	—	0	R	*Reserved*
5:0	vector	X	W	Interrupt vector (encodes bit set in ASI_SWVR_INTR_RECEIVE).

A read from this ASI causes a *data_access_exception* trap. Nonprivileged access to this register causes a *privileged_action* trap. Privileged access causes a *data_access_exception* trap.

> **Note** | If a write uses the reserved values of the type field, an interrupt is still generated.

7.4.3 Incoming Vector Register

Each strand has a hyperprivileged read-only `ASI_SWVR_UDB_INTR_R` register at ASI = 74_{16}, VA{63:0} = 0_{16}. When this register is read by software, the 6-bit vector corresponding to the highest priority pending interrupt in the interrupt receive register is returned. The pending interrupt bit for that vector is cleared. If no interrupt bits are set, this register will read as all zeros. When an interrupt arrives at the same time as the register is read, the interrupt will take precedence over the write and the bit will remain set. A store to this register will result in a *data_access_exception* trap.

TABLE 7-14 defines the format of the `ASI_SWVR_UDB_INTR_R` register.

TABLE 7-14   Incoming Vector Register – `ASI_SWVR_UDB_INTR_R` (ASI 74_{16}, VA 0_{16})

Bit	Field	Initial Value	R/W	Description
63:6	—	0	R	*Reserved*
5:0	vector	X	R	Interrupt vector

7.4.4 Interrupt Queue Registers

Each strand has eight `ASI_QUEUE` registers at ASI 25_{16}, VA{63:0} = $3C0_{16}$–$3F8_{16}$ that are used for communicating interrupts to the privileged mode operating system from the hypervisor. These registers contain the head and tail pointers for four supervisor interrupt queues: *cpu_mondo, dev_mondo, resumable_error,* and *nonresumable_error.*

The tail registers are read-only in privileged mode, and read/write in hyperprivileged mode. An attempted write to a tail register by privileged software generate a *data_access_exception* trap. The head registers are read/write by both privileged and hyperprivileged software.

Whenever the contents of the CPU_MONDO_HEAD and CPU_MONDO_TAIL registers are unequal, a *cpu_mondo* trap is generated. Whenever the contents of the DEV_MONDO_HEAD and DEV_MONDO_TAIL registers are unequal, a *dev_mondo* trap is generated. Whenever the contents of the RESUMABLE_ERROR_HEAD and RESUMABLE_ERROR_TAIL registers are unequal, a *resumable_error* trap is generated.

Unlike the other queue register pairs, the *nonresumable_error* trap is *not* automatically generated by hardware whenever the contents of the NONRESUMABLE_ERROR_HEAD and NONRESUMABLE_ERROR_TAIL registers are unequal; instead, hyperprivileged software must make it appear to privileged software as if a *nonresumable_error* trap has occured.

| Warning | There is a known "feature" in UltraSPARC T1 that affects LDXA/STXA by supervisor code to these ASI registers. If an immediately preceeding instruction is a store that takes a TLB-related trap, an LDXA can corrupt an unrelated IRF (integer register file) register, or a STXA may complete in spite of the trap. To prevent this, it is *required* to have a non-store or NOP instruction before any LDXA/STXA to these ASIs. If the LDXA/STXA is at a branch target, there must be a non-store in the delay slot. Nonprivileged software and hyperprivileged software are not affected by this. |

Programming Note	These registers are intended to be used as head and tail pointers into a queue in memory storing the mondo or error interrupt data. When the hypervisor takes an interrupt that it needs to pass on to the operating system, it stores the interrupt data into the end of the appropriate queue. Then hyperprivileged software updates the corresponding tail register to point beyond the new data, which causes a trap to be generated to privileged software (the operating system). Privileged software then processes the interrupt data from the head of the queue, updating the head register when the interrupt processing is completed.
	While the first interrupt is being serviced, more interrupts may be placed on the queue by the hypervisor. The operating system can read the tail pointer to service multiple interrupts at a time, or it can simply update the head pointer after each interrupt has been serviced and take a trap for each interrupt.
	When all pending interrupts of the appropriate type have been serviced, the head and tail pointers will be equal again, and no further traps will be generated until the hypervisor places new interrupt data on the queue.

TABLE 7-15 through TABLE 7-22 define the format of the eight interrupt queue registers.

TABLE 7-15   CPU Mondo Head Pointer – QUEUE_CPU_MONDO_HEAD (ASI 25_{16}, VA $3C0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	head	X	RW	Head pointer for CPU Mondo Interrupt Queue.
5:0	—	0	R	*Reserved*

TABLE 7-16   CPU Mondo Tail Pointer – QUEUE_CPU_MONDO_TAIL (ASI 25_{16}, VA $3C8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	tail	X	RW	Tail pointer for CPU Mondo Interrupt Queue.
5:0	—	0	R	*Reserved*

TABLE 7-17   Device Mondo Head Pointer – QUEUE_DEV_MONDO_HEAD (ASI 25_{16}, VA $3D0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	head	X	RW	Head pointer for Device Mondo Interrupt Queue.
5:0	—	0	R	*Reserved*

TABLE 7-18   Device Mondo Tail Pointer – QUEUE_DEV_MONDO_TAIL (ASI 25_{16}, VA $3D8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	tail	X	RW	Tail pointer for Device Mondo Interrupt Queue.
5:0	—	0	R	*Reserved*

TABLE 7-19   Resumable Error Head Pointer – QUEUE_RESUMABLE_HEAD (ASI 25_{16}, VA $3E0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	head	X	RW	Head pointer for Resumable Error Queue.
5:0	—	0	R	*Reserved*

TABLE 7-20   Resumable Error Tail Pointer – QUEUE_RESUMABLE_TAIL (ASI 25_{16}, VA $3E8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	tail	X	RW	Tail pointer for Resumable Error Queue.
5:0	—	0	R	*Reserved*

TABLE 7-21   Nonresumable Error Head Pointer – QUEUE_NONRESUMABLE_HEAD (ASI 25_{16}, VA $3F0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	head	X	RW	Head pointer for NonResumable Error Queue.
5:0	—	0	R	*Reserved*

TABLE 7-22   Nonresumable Error Tail Pointer – QUEUE_NONRESUMABLE_TAIL (ASI 25_{16}, VA $3F8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13:6	tail	X	RW	Tail pointer for NonResumable Error Queue.
5:0	—	0	R	*Reserved*

Memory Models

8.1 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 multiprocessor environments.

For a full description of the TSO memory model, see the *UltraSPARC Architecture 2005*.

UltraSPARC T1 supports only TSO, with the exception that accesses using certain ASIs (notably, block loads and block stores) may operate under RMO (impl. dep. #113-V9-Ms10).

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 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. Its value is always 0 on UltraSPARC T1. An UltraSPARC T1 virtual processor always operates under the TSO memory model.

Memory is logically divided into real memory (cached) and I/O memory (noncached, with and without side effects) spaces, based on bit 39 of the physical address (0 = real memory, 1= I/O memory) (impl. dep. #118-V9). Real memory spaces may be cached and can be accessed without side effects. For example, a read

(load) 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.

8.2 Supported Memory Models

The following sections contain brief descriptions of the two memory models supported by UltraSPARC T1. These definitions are for general illustration. Detailed definitions of these models can be found in *UltraSPARC Architecture 2005*. 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 72.

Notes | (1) Stores to UltraSPARC T1 Internal ASIs, block loads, and block stores are outside the memory model; that is, they need MEMBARs to control ordering. See Section 8.3.8, "Instruction Prefetch to Side-Effect Locations" on page 79 and Section 13.5.3, "Block Load and Store Instructions" on page 172.

(2) Atomic load-stores are treated as both a load and a store and can only be applied to cacheable address spaces.

8.2.1 Total Store Order

UltraSPARC T1 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 PA{39} set (that is, to I/O space) are all strongly ordered with respect to each other.

Compatibility Note	Previous UltraSPARC machines strongly order accesses when the TTE.e bit being set. The e bit is ignored by UltraSPARC T1 for the purposes of strong ordering; only PA{39} is used for determining strong ordering.

- An L2 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.

8.2.2 Relaxed Memory Order

UltraSPARC T1 implements the following programmer-visible properties for accesses through special ASIs that operate under the Relaxed Memory Order (RMO) model:

- There is no implicit order between any two memory references, either cacheable or noncacheable, except that noncacheable accesses with PA{39} set (that is, to I/O space) are all strongly ordered with respect to each other.

Compatibility Note	Previous UltraSPARC machines strongly order accesses based on the e bit being set. The e bit is ignored by UltraSPARC T1 for the purposes of strong ordering, only PA{39} is used for determining strong ordering.

- 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 #StoreLoad should be used between a store and a subsequent load at the same noncacheable address.

Address Spaces and ASIs

9.1 Physical Address Spaces

UltraSPARC T1 supports a 48-bit virtual address space and a 40-bit physical address space. The 40-bit physical address space is further broken into two sections, based on bit{39}. If bit{39} is a 0, the address maps to a memory location. If bit{39} is a 1, the address maps to an I/O location.

9.1.1 Access to Nonexistent Memory or I/O

Physical address bits 38:37 are always set to 0 on CPU requests to memory, so while no memory configuration will support memory this large, a request with these bits set to a nonzero value will be treated the same as if the bits were 0.

Accesses to nonexistent memory or I/O locations are treated as follows:

- A load access from a nonexistent memory or I/O location causes a *data_access_error* exception
- An instruction fetch from a nonexistent memory or I/O location causes an *instruction_access_error* exception
- A store access to a nonexistent memory or I/O location will be silently discarded by the system

9.1.2 Instruction Fetching from IO

Instruction fetching from I/O addresses is only permitted from the boot ROM space (FF 0000 0000$_{16}$ to FF FFFF FFFC$_{16}$). Instruction fetches from IO addresses outside the boot ROM space will take an *instruction_access_error* trap.

9.1.3 Supported vs. Unsupported Access Sizes to I/O

All I/O addresses that are internal to UltraSPARC T1 are 64-bit locations and only support 8-byte loads and stores; accesses in other sizes may cause traps or have other unexpected results.

UltraSPARC T1 supports 1-byte, 2-byte, 4-byte, and 8-byte loads and stores via the SSI bus (Boot ROM port). 16-byte loads are undefined, but are expected to perform an 8-byte read and duplicate the data. (UltraSPARC T1 cannot generate a 16-byte store.)

UltraSPARC T1 supports 1-byte, 2-byte, 4-byte, and 8-byte loads and stores, plus 16-byte loads, via JBUS (for external JBUS locations). Block loads are broken down by the LSU into four 16-byte loads, and are thus supported via JBUS.

Non-8-byte-aligned load accesses, except for JBUS and SSI locations, will result in *data_access_error* trap. Non-8B-aligned store accesses, except JBUS and SSI locations, will be silently discarded by the system.

Non-8-byte-aligned load accesses from internal JBUS or SSI locations are treated internally as 8-byte loads, with potentially undefined results. Non-8B-aligned store accesses to internal JBUS or SSI locations are treated internally as 8-byte stores, also with potentially undefined results.

9.1.4 48-bit Virtual Address Space

UltraSPARC T1 supports a 48-bit subset of the full 64-bit virtual address space (see FIGURE 9-1). 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 $0000\ 8000\ 0000\ 0000_{16}$ and $FFFF\ 7FFF\ FFFF\ FFFF_{16}$, 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 place *instructions* in memory pages within 4 Gbytes of the VA hole, to avoid problems due to prefetching of instructions from the VA hole in memory. UltraSPARC T1 assumes that this convention is followed, for similar reasons. Note that there are no trap mechanisms to detect a violation of this convention.

```
                                              FFFF FFFF FFFF FFFF

                                              FFFF 8001 0000 0000
                        See Note (1)          FFFF 8000 0000 0000

                                              FFFF 7FFF FFFF FFFF

  Out-of-Range VA
    ("VA Hole")

                                              0000 8000 0000 0000
                        See Note (1)          0000 7FFF FFFF FFFF

                                              0000 7FFE FFFF FFFF

                                              0000 0000 0000 0000
```
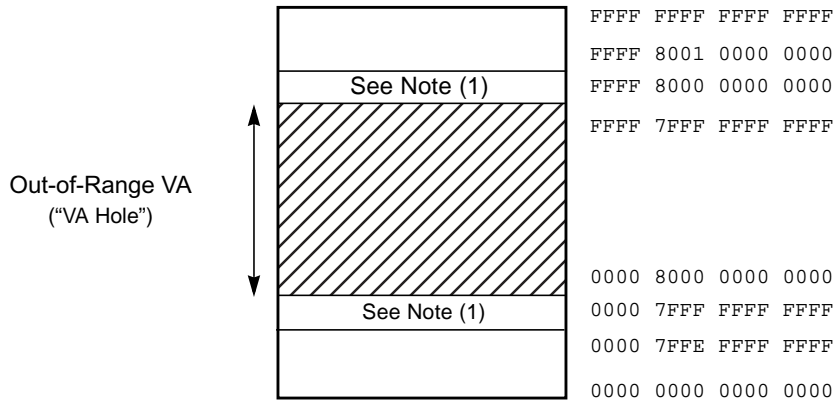
Note (1): Prior implementations restricted use of this region to data only.

FIGURE 9-1 UltraSPARC T1's 48-bit Virtual Address Space, With VA Hole

> **Note** | Throughout this document, when virtual address fields are specified as 64-bit quantities, bits 63:48 are assumed to be sign-extended from bit 47.

A number of state registers are affected by the reduced virtual address space. TBA, TPC, TNPC, and DMMU SFAR registers are 48 bits wide, sign-extended to 64 bits on read accesses. VA watchpointing is 48 bits, zero-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 = 0.

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 TPC[TL] set to 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.

When any other control transfer instruction traps, it sets TPC[TL] to the address of the target instruction along with setting 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 **xor**ing ones into the most significant 16 bits. See

also Section 13.9.6 "I-/D-MMU Synchronous Fault Status Registers (SFSR)" on page 13-210 and *Section 13.9.7 "I-/D-MMU Synchronous Fault Address Registers (SFAR)" on page 13-212*.

When a trap occurs on the delay slot of a taken branch or call whose target is out-of-range or is the last instruction below the VA hole, UltraSPARC T1 records the fact that NPC points to an out-of-range instruction in TNPC. If the trap handler executes a DONE or RETRY without saving TNPC, the *instruction_access_exception* trap is taken when the instruction at TNPC is executed. If TNPC is saved and subsequently restored by the trap handler, the fact that TNPC points to an out-of-range instruction is lost.

When a TLB data parity error occurs on a store that is followed by an instruction with its PC in the VA hole, UltraSPARC T1 records the fact that the PC points to an out-of-range instruction in TPC. If the trap handler executes a DONE or RETRY without saving TPC, the *instruction_access_exception* trap is taken when the instruction at TPC is executed. If TPC is saved and subsequently restored by the trap handler, the fact that TPC 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 48 bits, the trap handler must decode the load or store instruction if the full 64-bit virtual address is needed. See also Section 13.9.6 "I-/D-MMU Synchronous Fault Status Registers (SFSR)" on page 13-210 and Section 13.9.7 "I-/D-MMU Synchronous Fault Address Registers (SFAR)" on page 13-212.

9.1.5 I/O Address Spaces

I/O addresses are distinguished from memory addresses via their high-order physical address bit (bit 39). If bit 39 is 0, the address is a memory address. If bit 39 is a 1, the address is an I/O address.

The main function of the I/O subsystem is to coordinate data transfers between memory and different I/O devices. The architecture is shown in FIGURE 9-2. The I/O Bridge is the interface to the Cache Crossbar and serves as the CPU-I/O interfaces.The JBUS interface unit maintains coherency between external devices and the memory system. Its datapath connects to the L2 controller unit.
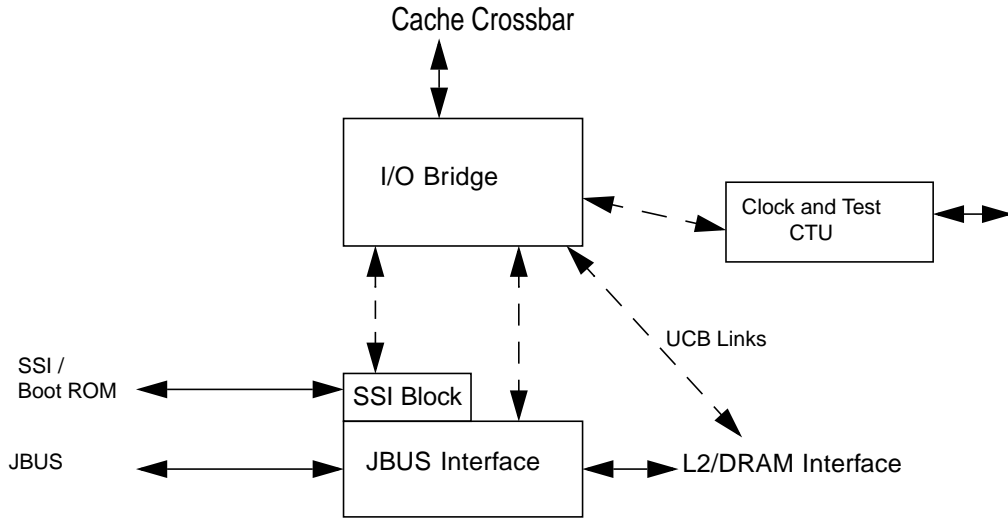
FIGURE 9-2   I/O Subsystem Architecture

In the following, we discuss each of the functional blocks in detail. Each hardware block has a base address and is defined by the 'RegisterBaseAddress' caption.

TABLE 9-1 summarizes the block base address assignment, which defines the eight most significant bits of the physical address.

TABLE 9-1   Block Address Assignment

Address Range (PA{39:32})	Block Name	Comment
00_{16}–$7F_{16}$	DRAM	
80_{16}	JBUS1	
81_{16}–95_{16}	—	*Reserved*
96_{16}	CLKU	Clock Unit
97_{16}	DRAMCSR	Control registers
98_{16}	IOBMAN	Management Block
99_{16}	TAP	TAP Unit
$9A_{16}$–$9D_{16}$	RSVD4	*Reserved*
$9E_{16}$	TAP2ASI	TAP access to ASI space (Use only from TAP, not from cores)

TABLE 9-1   Block Address Assignment *(Continued)*

Address Range (PA{39:32})	Block Name	Comment
$9F_{16}$	IOBINT	Interrupt Table
$A0_{16}$–BF_{16}	L2CSR	Control registers and diagnostic access
$C0_{16}$–FE_{16}	JBUS2	
FF_{16}	BOOT	Boot ROM

9.1.6   I/O Bridge

The I/O Bridge is the center of the control register access network. It is connected to the SPARC physical cores through the CPU crossbar and distributes the I/O requests to different devices through their associated Unit Control Blocks (UCB). TABLE 9-2 summarizes the different UCBs that are connected to the IOB. All blocks have a separate input bus from and a separate return bus to the IOB.

TABLE 9-2   UCB List

UCB Name	Comment
JBUS	
DRAM CSR	Two UCBs are implemented.
TAP	
Clock Unit	
Boot ROM / SSI	

9.2 Alternate Address Spaces

TABLE 9-3 summarizes the ASI usage in UltraSPARC T1; any ASI not listed behaves as described in the *UltraSPARC Architecture 2005* specification. The Section column lists where the operation of the ASI is explained. For several internal ASIs, a range of legal VAs is listed. An access outside the legal VA range will be aliased to a legal VA by ignoring the upper address bits.

Notes (1) All internal, nontranslating ASIs in UltraSPARC T1 can only be accessed using LDXA and STXA. This is different than UltraSPARC I/II, where LDDFA and STDFA can also be used to access internal ASIs. Using LDDFA and STDFA to access an internal ASI in UltraSPARC T1 results in a *data_access_exception* trap.

(2) ASIs 80_{16}–FF_{16} are unrestricted (nonprivileged, privileged, and hyperprivileged software may access). ASIs 00_{16}–$2F_{16}$ are restricted to privileged and hyperprivileged, while ASIs 30_{16}–$7F_{16}$ are restricted to hyperprivileged software only.

TABLE 9-3 UltraSPARC T1 ASI Usage *(1 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
00_{16}–03_{16}			Any	—	*data_access_exception* (SFSR.ct =3)	
05_{16}–$0B_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
$0D_{16}$–$0F_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
12_{16}–13_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
$1A_{16}$–$1B_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
$1C_{16}$	ASI_REAL_LITTLE	RW	Any	—	Nonallocating in L1 cache, same as ASI_REAL_IO_LITTLE for I/O addresses	9.2.1
20_{16}	ASI_SCRATCHPAD	RW	0_{16}–18_{16}	Y	Scratchpad registers 0–3	9.2.3
		RW	20_{16}–28_{16}	—	*data_access_exception* (SFSR.ct = 3)	9.2.3
		RW	30_{16}–38_{16}	Y	Scratchpad registers 6–7	9.2.3

TABLE 9-3   UltraSPARC T1 ASI Usage *(2 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section

22_{16}	`ASI_LDTX_AIUP,`
	
	`ASI_STBI_AIUP`
		RW	Any	—	(See *UltraSPARC Architecture 2005*)
					
					`ASI_STBI_AIUP` is used for Block-Initializing stores, As If User, Primary Context
						5.10

23_{16}	`ASI_LDTX_AIUS,`
	
	`ASI_STBI_AIUS`
		RW	Any	—	(See *UltraSPARC Architecture 2005*)
					
					`ASI_STBI_AIUS` is used for Block-Initializing stores, As If User, Secondary Context
						5.10

24_{16}	`ASI_TWINX (ASI_LDTX),` `ASI_QUAD_LDD`[D]†, `ASI_NUCLEUS_QUAD_LDD`[D]†	R	Any	—	128-bit atomic Load Twin Doubleword (deprecated; superseded by ASI 27_{16})	
25_{16}	`ASI_QUEUE`	RW	0_{16}–$3B8_{16}$	—	Load/store does NOP	
		RW	$3C0_{16}$–$3F8_{16}$	Y	(See *UltraSPARC Architecture 2005*)	

27_{16}	`ASI_LDTX_N,`
	
	`ASI_STBI_N`
		RW	Any	—	(See *UltraSPARC Architecture 2005*)
					
					`ASI_STBI_N` is used for Block-Initializing stores, Nucleus Context
						5.10

28_{16}–29_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	

$2A_{16}$	`ASI_LDTX_AIUP_L,`
	
	`ASI_STBI_AIUP_L`
		RW	Any	—	(See *UltraSPARC Architecture 2005*)
					
					`ASI_STBI_AIUP_L` is used for Block-Initializing stores, As If User, Primary Context, Little Endian
						5.10

$2B_{16}$	`ASI_LDTX_AIUS_L,`
	
	`ASI_STBI_AIUS_L`
		RW	Any	—	(See *UltraSPARC Architecture 2005*)
					
					`ASI_STBI_AIUS_L` is used for Block-Initializing stores, As If User, Seondary Context, Little Endian
						5.10

$2C_{16}$	`ASI_TWINX_LITTLE (ASI_LDTX_L),` `ASI_QUAD_LDD_LITTLE`[D]†, `ASI_NUCLEUS_QUAD_LDD_LITTLE`[D]†	R	Any	—	128-bit atomic Load Twin Doubleword, little endian (deprecated; superseded by ASI $2F_{16}$)	
$2D_{16}$			Any	—	*data_access_exception* (SFSR.CT=3)	

TABLE 9-3   UltraSPARC T1 ASI Usage *(3 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$2F_{16}$	ASI_LDTX_NL,	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.10
	ASI_STBI_NL				ASI_STBI_NL is used for Block-Initializing stores, Nucleus context, Little-Endian	
30_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
31_{16}	ASI_DMMU_CTXT_ZERO_TSB_BASE_PS0	RW	0_{16}	Y	DMMU Context Zero TSB Base PS0	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
32_{16}	ASI_DMMU_CTXT_ZERO_TSB_BASE_PS1	RW	0_{16}	Y	DMMU Context Zero TSB Base PS1	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
33_{16}	ASI_DMMU_CTXT_ZERO_CONFIG	RW	0_{16}	Y	DMMU Context Zero Config Register	13.9.4
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.4
34_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
35_{16}	ASI_IMMU_CTXT_ZERO_TSB_BASE_PS0	RW	0_{16}	Y	IMMU Context Zero TSB Base PS0	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
36_{16}	ASI_IMMU_CTXT_ZERO_TSB_BASE_PS1	RW	0_{16}	Y	IMMU Context Zero TSB Base PS1	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
37_{16}	ASI_IMMU_CTXT_ZERO_CONFIG	RW	0_{16}	Y	IMMU Context Zero Config Register	13.9.4
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.4
38_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
39_{16}	ASI_DMMU_CTXT_NONZERO_TSB_ BASE_PS0	RW	0_{16}	Y	DMMU Context Nonzero TSB Base PS0	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3

TABLE 9-3 UltraSPARC T1 ASI Usage *(4 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$3A_{16}$	ASI_DMMU_CTXT_NONZERO_TSB_BASE_PS1	RW	0_{16}	Y	DMMU Context Nonzero TSB Base PS1	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
$3B_{16}$	ASI_DMMU_CTXT_NONZERO_CONFIG	RW	0_{16}	Y	DMMU Context Nonzero Config Register	13.9.4
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.4
$3C_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
$3D_{16}$	ASI_IMMU_CTXT_NONZERO_TSB_BASE_PS0	RW	0_{16}	Y	IMMU Context Nonzero TSB Base PS0	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
$3E_{16}$	ASI_IMMU_CTXT_NONZERO_USB_BASE_PS1	RW	0_{16}	Y	IMMU context nonzero TSB Base PS1	13.9.3
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.3
$3F_{16}$	ASI_IMMU_CTXT_NONZERO_CONFIG	RW	0_{16}	Y	IMMU Context Nonzero Config Register	13.9.4
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	13.9.4
40_{16}	ASI_STREAM_MA	RW	0_{16}–78_{16}	—	Load/store does NOP	
		RW	80_{16}	N	Modular Arithmetic Control Register	16.1.1
		RW	88_{16}	N	Modular Arithmetic Physical Address register (MPA)	16.1.2
		RW	90_{16}	N	Modular Arithmetic Memory Address Register (MA_ADDR)	16.1.3
		RW	98_{16}	N	Modular Arithmetic NP Register	16.1.4
		RW	$A0_{16}$	N	Wait for async MA operation to complete	16.1.5
		RW	$A8_{16}$–$F8_{16}$	—	Load/store does NOP	
41_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
42_{16}	ASI_SPARC_BIST_CONTROL	RW	0_{16}	N	SPARC BIST Control register	?
42_{16}	ASI_INST_MASK_REG	RW	8_{16}	N	SPARC Instruction Mask register	?
42_{16}	ASI_LSU_DIAG_REG	RW	10_{16}	N	Load/Store Unit Diagnostic register	?

TABLE 9-3 UltraSPARC T1 ASI Usage *(5 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
42_{16}		RW	18_{16}–$F8_{16}$	—	Load/store does NOP	
44_{16}	ASI_STM_CTL_REG	RW	0_{16}	N	Self-timed Margin Control register	?
45_{16}	ASI_LSU_CONTROL_REG	RW	0_{16}	Y	Load/Store Unit Control Register	?
		RW	8_{16}–$F8_{16}$	—	Load/store does NOP	
46_{16}	ASI_DCACHE_DATA	RW	Any	Y	D-cache data array diagnostics access	?
47_{16}	ASI_DCACHE_TAG	RW	Any	Y	D-cache tag and valid bit diagnostics access	?
48_{16}–$4A_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
$4B_{16}$	ASI_SPARC_ERROR_EN_REG	RW	0_{16}	N	SPARC Error Enable reg (synchronous ecc/parity errors)	?
$4C_{16}$	ASI_SPARC_ERROR_STATUS_REG	RW	0_{16}	Y	SPARC Error Status register	?
$4D_{16}$	ASI_SPARC_ERROR_ADDRESS_REG	RW	0_{16}	Y	SPARC Error Address register	?
$4E_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
$4F_{16}$	ASI_HYP_SCRATCHPAD	RW	0_{16}–38_{16}	Y	Hyperprivileged Scratchpad	9.2.4
50_{16}	ASI_ITSB_TAG_TARGET	R	0_{16}	Y	IMMU Tag Target register	13.9.5
50_{16}	ASI_IMMU	RW	8_{16}–10_{16}	—	Load/store does NOP	
		RW	18_{16}	Y	IMMU Synchronous Fault Status register	13.9.6
		RW	20_{16}	—	Load/store does NOP	
		RW	28_{16}		*data_access_exception* (SFSR.ct = 3)	
		RW	30_{16}	Y	IMMU TLB Tag Access register	13.9.8
		RW	38_{16}–$F8_{16}$	—	Load/store does NOP	
51_{16}	ASI_IMMU_TSB_PS0_PTR_REG	R	0_{16}	Y	IMMU TSB PS0 Pointer register	13.9.10
		R	8_{16}–$F8_{16}$	—	Load does NOP	

TABLE 9-3 UltraSPARC T1 ASI Usage *(6 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
52_{16}	ASI_IMMU_TSB_PS1_PTR_REG	R	0_{16}	Y	IMMU TSB PS1 Pointer register	13.9.10
		R	8_{16}–$F8_{16}$		Load does NOP	
53_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
54_{16}	ASI_ITLB_DATA_IN_REG	W	0_{16}–$7F8_{16}$	Y	IMMU Data In register	13.9.11
55_{16}	ASI_ITLB_DATA_ACCESS_REG	RW	0_{16}–$7F8_{16}$	Y	IMMU TLB Data Access register	13.9.11
56_{16}	ASI_ITLB_TAG_READ_REG	R	0_{16}–$1F8_{16}$	Y	IMMU TLB Tag Read register	13.9.11
57_{16}	ASI_IMMU_DEMAP	W	Any	Y	IMMU TLB Demap	13.7
58_{16}	ASI_DTSB_TAG_TARGET	R	0_{16}	Y	DMMU Tag Target register	13.9.5
58_{16}	ASI_DMMU	RW	8_{16}–10_{16}	—	*data_access_exception* (SFSR.ct = 3)	
		RW	18_{16}	Y	DMMU Synchronous Fault Status register	13.9.6
		R	20_{16}	Y	DMMU Synchronous Fault Address register	13.9.7
		RW	28_{16}	—	*data_access_exception* (SFSR.ct = 3)	
		RW	30_{16}	Y	DMMU TLB Tag Access register	13.9.8
		RW	38_{16}	Y	DMMU VA Data Watchpoint register	?
		RW	40_{16}	—	*data_access_exception* (SFSR.ct = 3)	
		RW	48_{16}–78_{16}	—	Load/store does NOP	
		RW	80_{16}	Y	I/DMMU Partition ID	13.9.9
		RW	88_{16}–$F8_{16}$	—	Load/store does NOP	
59_{16}	ASI_DMMU_TSB_PS0_PTR_REG	R	0_{16}	Y	DMMU TSB PS0 Pointer register	13.9.10
		R	8_{16}–$F8_{16}$	—	Load does NOP	
$5A_{16}$	ASI_DMMU_TSB_PS1_PTR_REG	R	0_{16}	Y	DMMU TSB PS1 Pointer register	13.9.10
		R	8_{16}–$F8_{16}$	—	Load does NOP	

TABLE 9-3   UltraSPARC T1 ASI Usage *(7 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$5B_{16}$	ASI_DMMU_TSB_DIRECT_PTR_REG	R	0_{16}	Y	DMMU TSB Direct Pointer register	13.9.10
		R	8_{16}–$F8_{16}$	—	Load does NOP	
$5C_{16}$	ASI_DTLB_DATA_IN_REG	W	0_{16}–$7F8_{16}$	Y	DMMU Data In register	13.9.11
$5D_{16}$	ASI_DTLB_DATA_ACCESS_REG	RW	0_{16}–$7F8_{16}$	Y	DMMU TLB Data Access register	13.9.11
$5E_{16}$	ASI_DTLB_TAG_READ_REG	R	0_{16}–$1F8_{16}$	Y	DMMU TLB Tag Read register	13.9.11
$5F_{16}$	ASI_DMMU_DEMAP	W	Any	Y	DMMU TLB Demap	13.7
60_{16}	ASI_TLB_INVALIDATE_ALL	W	0_{16}	Y	IMMU TLB Invalidate register	13.7
		W	8_{16}	Y	DMMU TLB Invalidate register	13.7
		W	10_{16}–$F8_{16}$	—	Store does NOP	
61_{16}–65_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
66_{16}	ASI_ICACHE_INSTR	RW	Any	Y	I-cache data array diagnostics access	?
67_{16}	ASI_ICACHE_TAG	RW	Any	YN	I-cache tag and valid bit diagnostics access	?
68_{16}–71_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
72_{16}	ASI_SWVR_INTR_RECEIVE	RW	0_{16}	Y	Interrupt Receive register	7.4.1
73_{16}	ASI_SWVR_UDB_INTR_W	W	0_{16}	Y	Interrupt Vector Dispatch register	7.4.2
74_{16}	ASI_SWVR_UDB_INTR_R	R	0_{16}	Y	Incoming Vector register	7.4.3
75_{16}–$7F_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
84_{16}–87_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
$8C_{16}$–BF_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
$C0_{16}$	ASI_PST8_P		Any	—	*data_access_exception* (SFSR.ct = 0)[1]	
$C1_{16}$	ASI_PST8_S		Any	—	*data_access_exception* (SFSR.ct = 1)[1]	
$C2_{16}$	ASI_PST16_P		Any	—	*data_access_exception* (SFSR.ct = 0)[1]	
$C3_{16}$	ASI_PST16_S		Any	—	*data_access_exception* (SFSR.ct = 1)[1]	

TABLE 9-3   UltraSPARC T1 ASI Usage *(8 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$C4_{16}$	`ASI_PST32_P`		Any	—	*data_access_exception* $(SFSR.ct = 0)^1$	
$C5_{16}$	`ASI_PST32_S`		Any	—	*data_access_exception* $(SFSR.ct = 1)^1$	
$C6_{16}$– $C7_{16}$			Any	—	*data_access_exception* $(SFSR.ct = 3)$	
$C8_{16}$	`ASI_PST8_PL`		Any	—	*data_access_exception* $(SFSR.ct = 0)^1$	
$C9_{16}$	`ASI_PST8_SL`		Any	—	*data_access_exception* $(SFSR.ct = 1)^1$	
CA_{16}	`ASI_PST16_PL`		Any	—	*data_access_exception* $(SFSR.ct = 0)^1$	
CB_{16}	`ASI_PST16_SL`		Any	—	*data_access_exception* $(SFSR.ct = 1)^1$	
CC_{16}	`ASI_PST32_PL`		Any	—	*data_access_exception* $(SFSR.ct = 0)^1$	
CD_{16}	`ASI_PST32_SL`		Any	—	*data_access_exception* $(SFSR.ct = 1)^1$	
CE_{16}– CF_{16}			Any	—	*data_access_exception* $(SFSR.ct = 3)$	
$D0_{16}$	`ASI_FL8_P`		Any	—	*data_access_exception* $(SFSR.ct = 0)^2$	
$D1_{16}$	`ASI_FL8_S`		Any	—	*data_access_exception* $(SFSR.ct = 1)^2$	
$D2_{16}$	`ASI_FL16_P`		Any	—	*data_access_exception* $(SFSR.ct = 0)^2$	
$D3_{16}$	`ASI_FL16_S`		Any	—	*data_access_exception* $(SFSR.ct = 1)^2$	
$D4_{16}$– $D7_{16}$			Any	—	*data_access_exception* $(SFSR.ct = 3)$	
$D8_{16}$	`ASI_FL8_PL`		Any	—	*data_access_exception* $(SFSR.ct = 0)^2$	
$D9_{16}$	`ASI_FL8_SL`		Any	—	*data_access_exception* $(SFSR.ct = 1)^2$	
DA_{16}	`ASI_FL16_PL`		Any	—	*data_access_exception* $(SFSR.ct = 0)^2$	
DB_{16}	`ASI_FL16_SL`		Any	—	*data_access_exception* $(SFSR.ct = 1)^2$	
DC_{16}– DF_{16}			Any	—	*data_access_exception* $(SFSR.ct = 3)$	
$E0_{16}$	`ASI_BLK_COMMIT_P`	RW	Any	—	*data_access_exception* $(SFSR.ct = 0)^3$	

TABLE 9-3 UltraSPARC T1 ASI Usage *(9 of 9)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$E1_{16}$	ASI_BLK_COMMIT_S	RW	Any	—	*data_access_exception* (SFSR.ct = 1)[3]	
$E4_{16}$– $E9_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
EC_{16}– EF_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	
$F2_{16}$– $F7_{16}$			Any	—	*data_access_exception* (SFSR.ct = 3)	
FA_{16}– FF_{16}			Any	—	*data_access_exception* (SFSR.ct = 3)	

† This ASI name has been changed, for consistency; although use of this name is deprecated and software should use the new name, the old name is listed here for compatibility.

1. ASIs $C0_{16}$–$C5_{16}$, $C8_{16}$–CD_{16}, $D0_{16}$–$D3_{16}$, $D8_{16}$–DB_{16}, and $E0_{16}$–$E1_{16}$ are checked for a VA watchpoint and will generate a *VA_Watchpoint* trap if the watchpoint conditions are met. They are also checked for word-alignment and doubleword-alignment on STDFA, and will generate a *mem_address_not_aligned* trap if the effective address (R[rs1] + R[rs2]; note that R[rs2] is not used as a mask) is not word-aligned or a *stdf_mem_address_not_aligned* trap is the address is word-aligned, but not doubleword-aligned.

2. ASIs $D0_{16}$–$D3_{16}$ and $D8_{16}$–DB_{16} are checked for a VA watchpoint and will generate a *VA_Watchpoint* trap if the watchpoint conditions are met. They are also checked for word-alignment and doubleword-alignment on STDFA and LDDFA, and will generate a *mem_address_not_aligned* trap if the address is not word-aligned or a *stdf_mem_address_not_aligned*/ *lddf_mem_address_not_aligned* trap is the address is word-aligned, but not doubleword-aligned.

3. ASIs $E0_{16}$–$E1_{16}$ are checked for a VA watchpoint and will generate a *VA_Watchpoint* trap if the watchpoint conditions are met. They are also checked for word-alignment and doubleword-alignment on STDFA, and will generate a *mem_address_not_aligned* trap if the address is not word-aligned or a *stdf_mem_address_not_aligned* trap is the address is word-aligned, but not doubleword-aligned.

9.2.1 ASI_REAL and ASI_REAL_LITTLE

The ASI_REAL[_LITTLE] ASIs are used to bypass virtual-to-real address translation. The real address is set equal to the virtual address (that is, RA{63:0} ← VA{63:0}), the dTLB performs the real-to-physical translation, and the attriubtes used are those present in the matching TTE.

This ASI is used to bypass the data MMU for memory addresses (PA{39} = 0). When the access to this ASI bypasses the TLB, the physical address is set equal to the truncated virtual address (that is, PA{39:0} = V A{39:0}). The physical page attribute bit w is set to 1 and all other attribute bits are set to 0 for accesses to this ASI. Since the cp page attribute bit is clear, load accesses using this ASI will always fetch their data from the L2 cache. Using this ASI for an I/O address (PA{39} = VA{39} = 1) is permitted, and will follow the same page attributes (w = 1, all other attributes 0).

> **Programming Note** Although it is permitted to use ASI_REAL (or ASI_REAL_LITTLE) for an I/O access (PA{39 = VA{39} = 1) when bypassing the TLB, it is not recommended to do so because the e bit is not set for the access. ASI_REAL_IO and ASI_REAL_IO_LITTLE should be used instead.

9.2.2 ASI_REAL_IO and ASI_REAL_IO_LITTLE

This ASI is used to bypass the data MMU for I/O addresses (PA{39} = 1).When the access to this ASI bypasses the TLB, the physical address is set equal to the truncated virtual address (that is, PA{39:0} = VA{39:0}). The physical page attributes e and w are set to 1 and all other attribute bits are set to 0 for accesses to this ASI. Using this ASI for a memory address (PA{39} = VA{39} =0) is permitted, and will follow the same page attributes (e = 1, w = 1, all other attributes 0). When the access to this ASI requires an RA-to-PA translation, the real address is set equal to the virtual address (that is, RA{63:0} = VA{63:0}), and the attributes used are those present in the matching TTE.

Note	An atomic load-store operation is not permitted to these ASIs; an attempt to execute one will result in a *data_access_exception* exception.

Implementation Note	Accesses to this ASI that require a real-to-physical translation but cause a trap such as *data_access_exception* that updates the Synchronous Fault Status register will load the SFSR using the attributes from the TLB. If there is no matching TLB entry, then the attributes used will be all set to 0

9.2.3 ASI_SCRATCHPAD

Each strand has a set of six privileged ASI_SCRATCHPAD registers, accessed through ASI 20_{16} with VA{63:0} = 0_{16}, 8_{16}, 10_{16}, 18_{16}, 30_{16}, or 38_{16}. These registers are for scratchpad use by privileged software. VA 20_{16} and 28_{16} may be used to access two additional scratchpad registers. However, access to those two scratchpad registers will be much slower than to the other six (because accesses to them will cause a *data_access_exception* trap and the access will be emulated by hyperprivileged software).

TABLE 9-4 defines the format of these registers.

TABLE 9-4   Scratchpad – `ASI_SCRATCHPAD` (ASI 20_{16}; VA 0_{16}, 8_{16}, 10_{16}, 18_{16}, 30_{16}, or 38_{16})

Bit	Field	Initial Value	R/W	Description
63:0	scratchpad	X	RW	Scratchpad.

> **Warning** | There is a known "feature" in UltraSPARC T1 that affects LDXA/STXA by privileged code to these ASI registers. If an immediately preceeding instruction is a store that takes a TLB-related trap, an LDXA can corrupt an unrelated IRF (integer register file) register, or a STXA may complete in spite of the trap. To prevent this, it is *required* to have a non-store or NOP instruction before any LDXA/STXA to this ASI. If the LDXA/STXA is at a branch target, there must be a non-store in the delay slot. Nonprivileged software and hypervisor software are not affected by this.

9.2.4   ASI_HYP_SCRATCHPAD

Each strand has a set of eight hyperprivileged `ASI_HYP_SCRATCHPAD` registers at ASI: $4F_{16}$, VA{63:0} = 0_{16}–38_{16}. These registers are for scratchpad use by hyperprivileged software and for aliased access to the privileged scratchpad registers.

TABLE 9-5 defines the format of the `ASI_HYP_SCRATCHPAD` registers.

TABLE 9-5   Hyperprivileged Scratchpad – `ASI_HYP_SCRATCHPAD` (ASI $4F_{16}$, VA 0_{16}–38_{16})

Bit	Field	Initial Value	R/W	Description
63:0	scratchpad	X	RW	Scratchpad.

> **Note** | There is only a single set of eight scratchpad registers, six of which are accessible via both `ASI_SCRATCHPAD`, and `ASI_HYP_SCRATCHPAD`. `ASI_SCRATCHPAD` is intended to used primarily by privileged code, and only has access to the first four and last two registers of the eight entry scratchpad array. `ASI_HYP_SCRATCHPAD` can only be accessed in hyperprivileged mode, and has full access to all eight scratchpad registers. The registers at VA 20_{16} and 28_{16} are accessible exclusively via `ASI_HYP_SCRATCHPAD`.

CHAPTER **10**

Performance Instrumentation

10.1 Performance Control Register

Each virtual processor has a privileged Performance Control register (PCR). Nonprivileged accesses to this register cause a *privileged_opcode* trap. The performance control register contains six fields: ovfh, ovfl, sl, ut, st, and priv.

- ovfh and ovfl are state bits associated with the PIC.h and PIC.l overflow traps and are provided in this register to allow swapping out of a process that is in the state between the counter overflowing and the overflow trap being generated.

- sl controls which events are counted in PIC.l.

- ut controls whether user-level (nonprivileged) events are counted.

- st controls whether supervisor-level (privileged) events are counted. Hyperprivileged events are never counted.

- priv controls whether the PIC register can be read or written by nonprivileged software.

The format of this register is shown in TABLE 10-1. Note that changing the fields in PCR does not affect the PIC values. To change the events monitored, software needs to disable counting via PCR, reset the PIC, and then enable the new event via the PCR.

TABLE 10-1 Performance Control Register – PCR (ASR 10_{16})

Bit	Field	Initial Value	R/W	Description
63:10	—	0	R	*Reserved*
9	ovfh	0	RW	If 1, PIC.h has overflowed, and the next count event will cause a disrupting trap (*pic_overflow*) to hyperprivileged software. The trap will appear to be precise to the instruction following the event.
8	ovfl	0	RW	If 1, PIC.l has overflowed
7	—	0	R	*Reserved*

TABLE 10-1   Performance Control Register – PCR (ASR 10_{16})

Bit	Field	Initial Value	R/W	Description
6:4	sl	0	RW	Selects one of eight events to be counted for PIC.l, per TABLE 10-2.
3	—	0	R	*Reserved*
2	ut	0	RW	If ut = 1, count events in user mode; otherwise, ignore user mode events.
1	st	0	RW	If st = 1, count events in supervisor mode; otherwise, ignore supervisor mode events.
0	priv	0	RW	If priv = 1, prevent access to PIC by user-level code. If priv = 0, allow access to PIC by user-level code.

TABLE 10-2 contains the settings for the sl field.

TABLE 10-2   sl Field Settings

Event Names	Encoding	PIC	Description
Instr_cnt	sl = XXX	H	Number of completed instructions. Annulled, mispredicted, or trapped instructions are not counted.[1]
SB_full	sl = 000	L	Number of store buffer full cycles.[2]
FP_instr_cnt	sl = 001	L	Number of completed floating-point instructions.[3] Annulled or trapped instructions are not counted.
IC_miss	sl = 010	L	Number of instruction cache (L1) misses.
DC_miss	sl = 011	L	Number of data cache (L1) misses for loads (store misses are not included as the cache is write-through, non-allocating).
ITLB_miss	sl = 100	L	Number of instruction TLB miss trap taken (includes real_translation misses).
DTLB_miss	sl = 101	L	Number of data TLB miss trap taken (includes real_translation misses).
L2_imiss	sl = 110	L	Number of secondary cache (L2) misses due to instruction cache requests.
L2_dmiss_ld	sl = 111	L	Number of secondary cache (L2) misses due to data cache load requests.[4]

1. Tcc instructions that are cancelled due to encountering a higher-priority trap are still counted. The most likely traps to cause a Tcc to be cancelled are *data_error, pic_overflow, trap_level_zero,* and *hstick_match.* These are all very low probability traps. Loads that encounter a data parity error in the DTLB are still counted. Again, this is a very low probability trap.

2. SB_full increments every cycle a strand (virtual processor) is stalled due to a full store buffer, regardless of whether other strands are able to keep the processor busy. The overflow trap for SB_full is not precise to the instruction following the event that occurs when ovfl is set (the trap may occur on either the instruction following the event that occurs when ovfl is set, or on either of the next two instructions).

3. Only floating point instructions which execute in the shared FPU are counted. The following instructions are executed in the shared FPU: FADDS, FADDD, FSUBS, FSUBD, FMULS, FMULD, FDIVS, FDIVD, FSMULD, FSTOX, FDTOX, FXTOS, FXTOD, FITOS, FDTOS, FITOD, FSTOD, FSTOI, FDTOI, FCMPS, FCMPD, FCMPES, FCMPED.

4. L2 misses due to stores cannot be counted by the performance instrumentation logic.

10.2 SPARC Performance Instrumentation Counter

Each strand (virtual processor) has a Performance Instrumentation Counter register (PIC). Access privilege to PIC is controlled by the setting of PCR.priv. When PCR.priv = 1, a nonprivileged access to this register causes a *privileged_action* trap. The PIC counter contains two fields, h and l. The PIC.h field always counts the number of completed instructions. The PIC.l field counts the event selected by PCR.sl.

The ut and st fields for PCR control whether events from user (nonprivileged) mode, supervisor (privileged) mode, both, or neither are counted. Hyperprivileged events are never counted. Whenever PCR.ovfh is set (which normally occurs when the PIC.h counter overflows, but may also be set via a write to the PCR), a *pic_overflow* exception and subsequent disrupting trap is generated on the next event that increments the counter. The trap is disrupting, but is delivered at very high priority (only resets and *trap_level_zero* are higher priority). This trap will appear to be precise to the instruction following the one that caused the event, as long as a *trap_level_zero* trap does not occur on the same cycle as the instruction following the one that caused the event and the *pic_overflow* trap is not masked (PSTATE.ie = 1 and PIL < 15).

Implementation Note	For a DTLB miss, the trap may be taken for an instruction that didn't execute (the trap doesn't include the effect of other higher-priority traps superseding the *fast_data_access_MMU_miss* trap).
Programming Note	A WRasr to PCR that modifies the ovfh or ovfl bit behaves as if the ovfh or ovfl bit was modified before the WRasr is executed.
	This implies that if all of the following conditions are true, a performance counter overflow (*pic_overflow*) trap will be taken (to hyperprivileged software) on the instruction following the WRasr:
	• a WRasr is executed in privileged mode that sets ovfh or ovfl to 1
	• PCR.st = 1
	• the WRasr generates the event being counted

The counter overflow trap is signaled via bit 15 of the SOFTINT register. When this condition occurs, bit 15 of the SOFTINT register is set to one, which will generate an *interrupt_level_15* trap to the appropriate strand, if PSTATE.ie = 1 and pil < 15. The priority of this trap is higher than the normal *interrupt_level_15* trap, at priority 2 (where it is higher priority than *fast_instruction_MMU_miss* but lower priority than *trap_level_zero*).

The format of the PIC register is shown in TABLE 10-3.

TABLE 10-3 Performance Instrumentation Counter Register – PIC (ASR 11_{16})

Bit	Field	Initial Value	R/W	Description
63:32	h	0	RW	Instruction counter.
31:0	l	0	RW	Programmable event counter, event controlled by PCR.sl.

10.3 DRAM Performance Counter

RegisterBaseAddress DRAM CSR Registers – 97 0000 0000$_{16}$. Each DRAM channel has a pair of performance counters, packed into a single register, plus a register to control what is counted. TABLE 10-4 through TABLE 10-6 describe the registers.

TABLE 10-4 DRAM Performance Control Register – DRAM_PERF_CTL_REG (0000000400$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:8	—	X	R	*Reserved*
7:4	sel0	0	R	Select code for performance counter 0.
3:0	sel1	0	R	Select code for performance counter 1.

TABLE 10-5 DRAM Performance Counter Register – DRAM_PERF_COUNT_REG (0000000408$_{16}$)

Bit	Field	Initial Value	R/W	Description
63	sticky0	0	RW	Sticky overflow for counter 0.
62:32	counter0	0	RW	Performance counter 0
31	sticky1	0	RW	Sticky overflow for counter 1.
30:0	counter1	0	RW	Performance counter 1

TABLE 10-6 DRAM Performance Counter Select Codes

Event Name	Select	Description
mem_reads	0000	Read transactions.
mem_writes	0001	Write transactions.
mem_read_write	0010	Read + write transactions.
bank_busy_stalls	0011	Bank busy stalls; incremented by one each cycle there are requests in the queue, but none can issue because of bank conflicts
rd_queue_latency	0100	Read queue latency; incremented by n each cycle, where n is the number of read transactions in the queue.

TABLE 10-6   DRAM Performance Counter Select Codes

Event Name	Select	Description
wr_queue_latency	0101	Write queue latency; incremented by n each cycle, where n is the number of write transactions in the queue
rw_queue_latency	0110	(Read + Write) queue latency; incremented by n each cycle, where n is the number of transactions in the queue.
wr_buf_hits	0111	Writeback buffer hits; incremented by one each time a read transaction is deferred because it conflicts with a queued write transaction.
	1xxx	*Reserved*.

10.4   JBUS Performance Counters

RegisterBaseAddress 13 JBI – 80 0000 0000$_{16}$.   JBI has a pair of performance counters, packed in a single 64-byte register. Control of the counters is through a separate CSR, which specifies what is counted in each counter. Turning counting on and off is controlled by specifying to count something vs. counting nothing (sel = 0_{16}). Counts are reinitialized by writing zero to the counters.

For the latency count events, the JBI keeps track of the number of outstanding transactions of the particular type, then incrementing the count each cycle by the # of outstanding transactions. If the other counter counts # of read transactions, you can calculate average latency by dividing (Total Latency / # of Read transactions).

The most significant bit of each counter is a sticky overflow bit, which can only be cleared by a CSR write.

Implementation Note	There is a potential starvation case affecting JBUS or PCI PIO read returns, if multiple strands keep a continuous stream of reads to JBI internal registers. The only JBI internal registers likely to be exposed to non-hyperprivileged access are the JBI performance counters. Because of the potential starvation case, access to these registers should not be given to potentially malicious users.

Refer to TABLE 10-7 and TABLE 10-8.

TABLE 10-7   186 JBUS Performance Counter Control – JBI_PERF_CTL (0002_{16}-0000_{16})

Bit	Field	Initial Value	R/W	Description
63:8	—	X	R	*Reserved*
7:4	event_sel1	X	RW	Which event to count in counter1.
3:0	event_sel2	X	RW	Which event to count in counter2.

TABLE 10-8 JBUS Performance Counter Select Encodings

Event Name	sel Value	Description
	0_{16}	Nothing; doesn't count.
jbus_cycles	1_{16}	JBUS cycles (that is, Time)
dma_reads	2_{16}	DMA read transactions (inbound)
dma_read_latency	3_{16}	Total DMA read latency
dma_writes	4_{16}	DMA write transactions
dma_write8	5_{16}	DMA WR8 subtransactions
ordering_waits	6_{16}	Ordering Waits: # of JBI $\rightarrow$ L2 queues blocked each cycle
	7_{16}	
pio_reads	8_{16}	PIO read transactions (outbound)
pio_read_latency	9_{16}	Total PIO
	A_{16}	
	B_{16}	
aok_dok_off_cycles	C_{16}	AOK_OFF or DOK_OFF seen (cycles)
aok_off_cycles	D_{16}	AOK_OFF seen (cycles)
dok_off_cycles	E_{16}	DOK_OFF seen (cycles)
	F_{16}	

TABLE 10-9 187 JBUS Performance Counters – JBI_PERF_COUNT (0002_{16}-0008_{16})

Bit	Field	Initial Value	R/W	Description
63	sticky1	X	RW	Sticky overflow bit for counter 1.
62:32	count1	X	RW	Performance counter 1.
31	sticky2	X	RW	Sticky overflow bit for counter 2.
30:0	count2	X	RW	Performance counter 2.

Clocks, Reset, RED_state, and Initialization

11.1   Clock Unit

The clock unit block contains the control registers for chipwide clocking.

RegisterBaseAddress 6 CLKU – 96 0000 0000$_{16}$.   The following register, whose format is shown in TABLE 11-1, contains the clock divisors for the PLL.

TABLE 11-1   Clock Divider – CLK_DIV (0000$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:42	—	X	R	*Reserved*
61:52	dmult	8	RW	Common multiple. Number of DRAM clock cycles from coincident rising edges on CMP_CLK, DRAM_CLK, and SSI_CLK (JBUS_CLK ÷ 4) to another set of coincident rising edges. This will always be cmult × (cdiv ÷ ddiv). However, it may *never* be less than 8 for correct hardware operation. Reset only on POR.
51:42	jmult	8	RW	Common multiple. Number of JBUS clock cycles from coincident rising edges on CMP_CLK, DRAM_CLK, and SSI_CLK (JBUS_CLK ÷ 4) to another set of coincident rising edges. This will always be cmult × (cdiv ÷ jdiv). However, it may *never* be less than 8 for correct hardware operation. Reset only on POR.
41:28	cmult	32	RW	Common multiple. Number of CMP clock cycles from coincident rising edges on CMP_CLK, DRAM_CLK, and SSI_CLK (JBUS_CLK ÷ 4) to another set of coincident rising edges. Will usually be the least common multiple (LCM) of cdiv, jdiv × 4, and ddiv after removing common factors. However, it may *never* be less than 16 for correct hardware operation. Reset only on POR.
27	—	X	R	*Reserved*

TABLE 11-1   Clock Divider – CLK_DIV (0000$_{16}$) *(Continued)*

Bit	Field	Initial Value	R/W	Description
26	change	0	RW	Software should set this bit to force a frequency change on the next warm reset, if it changes any of the *div fields below. Hardware clears this bit after the reset, either warm reset or POR.
25:21	—	X	R	*Reserved*
20:16	ddiv	16	RW	Dram clock divisor. Reset only on POR.
15:13	—	X	R	*Reserved*
12:8	jdiv	16	RW	JBUS clock divisor. Reset only on POR.
7:5	—	X	R	*Reserved*
4:0	cdiv	4	RW	CMP clock divisor. Reset only on POR. Only values of 2, 4, and 8 are supported and tested.

The clock frequencies of the CMP core and for DRAM are a function of the input J_CLK and the ddiv, jdiv, and cdiv fields. CMP core frequency is (JBUS frequency × jdiv ÷ cdiv), and DRAM frequency is (JBUS frequency × jdiv ÷ ddiv).

> **Caution** | The values of mult, ddiv, jdiv, and cdiv in the CLK_DIV register and all fields in the CLK_JSYNC and CLK_DSYNC register are interdependent and must be changed together in a coherent fashion. Illegal values exist that will prohibit correct functional operation. In addition, valid values should not be reverse-engineered by experimentation. Values exist that may work at some process, voltage and temperature points, but do not allow sufficient margin for correct electrical operation across PVT combinations.

The recommended procedure for doing a frequency change with warm reset is listed below:

1. Update the CLK_JSYNC register to the values needed for the new frequency point.

2. Update the CLK_DSYNC register to the values needed for the new frequency point.

3. Write the CLK_DIV register to the values needed for the new frequency point. This write should have the change bit set to a 1 and must be a 64 bit write (doubleword store).

4. Cause the warm reset to happen by writing a register in the bus controller or signaling the system controller.

The Clock Control register, whose formats are shown in TABLE 11-4, contains the cluster clock enables and clocking control.

TABLE 11-2   Clock Control – CLK_CTL (0008_{16})

Bit	Field	Initial Value	R/W	Description
63	ossdis	1	RW	If 1, and clkdis = 1, a debug trigger will stop clocks in all clusters simultaneously. Otherwise, clocks are disabled at rate specified by cken_delay field. Reset only on POR.
62	clkdis	0	RW	If 1, disable clocks upon assertion of debug trigger. Reset only on POR.
61	srarm	0	RW	If 1, arm self-refresh of DRAM upon assertion of warm-reset. Reset only on POR.
60:55	—	X	R	*Reserved*
54:48	cken_delay	$7F_{16}$	RW	Number of CMP cycles between disabling individual cluster clock enables, for turning clocks off, if ossdis = 0. Actual value is clk_en_delay + 2, so if field is $7F_{16}$, interval is 129 CMP cycles between cken changes. Minimum legal value for cken_delay is 01_{16}. Reset only on POR.
47:35	—	X	R	*Reserved*
34	misc	1	RW	Clock enable for MISC block
33	dbg	1	RW	Clock enable for DBG pad block
32	efc	1	RW	Clock enable for EFC (EFuse).
31	iob	1	RW	Clock enable for IOB
30	jbusr	1	RW	Clock enable for JBUS_R IO pads
29	jbusl	1	RW	Clock enable for JBUS_L IO pads
28	—	X	RW	*Reserved*. (Was clock enable for BSC)
27	jbi	1	RW	Clock enable for JBI
26:25	—	X	RW	*Reserved*. (Was clock enable for Ethernet)
24	—	0	RO	*Reserved* (would be CTU clock enables).
23:20	ddr	F_{16}	RW	Clock enables for memory IO pads
19	fpu	1	RW	Clock enable for the floating-point unit.
18	ccx	1	RW	Clock enable for the crossbar.
17:16	dram	3_{16}	RW	Clock enables for the memory controllers.
15:12	sctag	F_{16}	RW	Clock enables for the four banks of L2 Tag.
11:8	scdata	F_{16}	RW	Clock enables for the four banks of L2 Data.
7:0	sparcore	FF_{16}	RW	Clock enables for the eight SPARC physical cores.

Note | Note that the clock enables are reset on both POR and warm resets. This means that software trying to disable clocks to particular clusters must turn off the clock enable after the last reset.

The register shown in TABLE 11-5 controls the DLLs to the DRAMs.

TABLE 11-3   Clock DLL Control – CLK_DLL_CNTL (0018_{16})

Bit	Field	Initial Value	R/W	Description
63:45	—	X	R	*Reserved*
44:40	dbg_delay	0_{16}	RW	Number of CMP cycles to delay stopping clocks, on a debug trigger.
39	—	X	R	*Reserved*
38	stretch_mode	0_{16}	RW	Clock stretch mode. 0 = precise mode, 1 = multiple clock mode
37	—	X	R	*Reserved*
36:32	stretch	02_{16}	RW	Clock stretch count
31:20	—	X	R	*Reserved*
19	ddr3_dll_ovf	X	R	DRAM3 deskew overflow
18	ddr3_dll_lock	X	R	DRAM3 deskew lock
17:15	ddr3_dll_delay	3_{16}	RW	DRAM3 deskew delay
14	ddr2_dll_ovf	X	R	DRAM2 deskew overflow
13	ddr2_dll_lock	X	R	DRAM2 deskew lock
12:10	ddr2_dll_delay	3_{16}	RW	DRAM2 deskew delay
9	ddr1_dll_ovf	X	R	DRAM1 deskew overflow
8	ddr1_dll_lock	X	R	DRAM1 deskew lock
7:5	ddr1_dll_delay	3_{16}	RW	DRAM1 deskew delay
4	ddr0_dll_ovf	X	R	DRAM0 deskew overflow
3	ddr0_dll_lock	X	R	DRAM0 deskew lock
2:0	ddr0_dll_delay	3_{16}	RW	DRAM0 deskew delay

The register shown in TABLE 11-7 controls generation of JBUS clock synchronization pulses

TABLE 11-4   Clock JBUS Sync – CLK_JSYNC (0028_{16})

Bit	Field	Initial Value	R/W	Description
63:40	—	X	R	*Reserved*
39:38	jsync_rcv2	1	RW	Number of cycles after jsync_trn2 to generate a receive sync pulse.
37	—	X	R	*Reserved*
36:32	jsync_trn2	0	RW	Count value for which one transmit sync pulse will be generated.
31:30	jsync_rcv1	1	RW	Number of cycles after jsync_trn1 to generate a receive sync pulse.
29	—	X	R	*Reserved*
28:24	jsync_trn1	0	RW	Count value for which one transmit sync pulse will be generated.
23:22	jsync_rcv0	1	RW	Number of cycles after jsync_trn0 to generate a receive sync pulse.
21	—	X	R	*Reserved*
20:16	jsync_trn0	0	RW	Count value for which one transmit sync pulse will be generated.

TABLE 11-4 Clock JBUS Sync – CLK_JSYNC (0028_{16}) *(Continued)*

Bit	Field	Initial Value	R/W	Description
15:13	—	X	R	*Reserved*
12:8	jsync_init	2	RW	Initial value for JBUS sync counter. The counter will be loaded with this value once after each PLL lock.
7:5	—	X	R	*Reserved*
4:0	jsync_period	3	RW	Period for JBUS sync pulse generation. Defines new value for decrementing counter when 0 (the terminal count) is reached.

Please see note with CLK_DIV register about the proper sequence for updating the CLK_JSYNC and CLK_DSYNC registers.

The register shown in TABLE 11-8 controls generation of DRAM clock synchronization pulses.

TABLE 11-5 Clock DRAM Sync– CLK_DSYNC (0030_{16})

Bit	Field	Initial Value	R/W	Description
63:40	—	X	R	*Reserved*
39:38	dsync_rcv2	1	RW	Number of cycles after dsync_trn2 to generate a receive sync pulse.
37	—	X	R	*Reserved*
36:32	dsync_trn2	0	RW	Count value for which one transmit sync pulse will be generated.
31:30	dsync_rcv1	1	RW	Number of cycles after dsync_trn1 to generate a receive sync pulse.
29	—	X	R	*Reserved*
28:24	dsync_trn1	0	RW	Count value for which one transmit sync pulse will be generated.
23:22	dsync_rcv0	1	RW	Number of cycles after dsync_trn0 to generate a receive sync pulse.
21	—	X	R	*Reserved*
20:16	dsync_trn0	0	RW	Count value for which one transmit sync pulse will be generated.
15:13	—	X	R	*Reserved*
12:8	dsync_init	2	RW	Initial value for JBUS sync counter. The counter will be loaded with this value once after each PLL lock.
7:5	—	X	R	*Reserved*
4:0	dsync_period	3	RW	Period for JBUS sync pulse generation. Defines new value for decrementing counter when 0 (the terminal count) is reached.

11.2 Reset Status Register

RegisterBaseAddress 1 IOBMAN – 98 0000 0000$_{16}$. The chip reset status, shown in TABLE 11-9, is maintained for all chip-wide reset and power management commands. The reset source bits in this register are writable to allow software to clear them after the chip reset sequence is complete, in order for strand (virtual processor) warm resets to be distinguished from chip resets. Hardware will copy the current reset status into a shadow status whenever a reset occurs.

TABLE 11-6   Chip Reset Status Register – RSET_STAT (0810_{16})

Bit	Field	Initial Value	R/W	Description
63:12	—	X	R	*Reserved*
11	freq_s	X	R	Shadow status of FREQ
10	por_s	X	R	Shadow status of POR
9	wmr_s	X	R	Shadow status of WMR
8:4	—	X	R	*Reserved*
3	freq	0	RW	Set to one if the reset is a warm reset that changed frequency.
2	POR	1	RW	Set to one if the reset is from PWRON_RST pin.
1	WMR	0	RW	Set to one if the reset is from the WRM_RST pin.
0	—	X	R	*Reserved*

11.3 Reset Overview

A reset is anything that causes an entry to RED_state. Two classes of resets exist: chipwide and strand. Chipwide resets affect all subsystems in a chip and are generated from the power-on and warm reset signals sourced from the external system. Strand resets are generated from writes to the INT_VEC_DIS register, as well as software resets and error conditions. In addition to forcing entry to RED_state, various resets cause different effects in initializing processor state, as discussed in the following sections. Reset priorities (from highest to lowest), per the *UltraSPARC Architecture 2005*, are POR, WMR, XIR, WDR, SIR. Resets are not maskable, that is, resets ignore PSTATE.ie).

11.4 Chipwide Resets and Power Management

Chipwide resets affect all virtual processors in a chip, as well as all I/O, cache, and DRAM subsystems. A chipwide reset is categorized as either a power-on or a warm reset. Power-on reset is used when the chip power and clock inputs are outside of their operating specifications. Warm reset is used when the power and clock inputs are stable. Warm reset is typically used to modify clock frequencies or ratios, or to reinitialize the chip after an unrecoverable hardware or software failure.

11.4.1 Power-on Reset (POR)

A power-on reset occurs when the PWRON_RST pin is asserted and then deasserted. The PWRON_RST pin must be asserted until the CPU voltages and input clocks reach their operating specifications. When the PWRON_RST 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}.

Since POR and warm reset share the same trap type and trap vector, the RSET_STAT register, described in *Reset Status Register* on page 96, has separate POR and warm reset bits to allow software to distinguish between POR and warm resets.

During power-on reset, all pending transactions are canceled. Strand 0 of the first available physical core begins executing at the *RSTV$_{ADDR}$* (reset trap vector address) base plus POR offset, while the remaining strands start out inactive. BIST testing may optionally be initiated by software as part of the chip initialization sequence.

After a power-on reset, software must initialize values specified as *unknown* in *Machine State after Reset and in* `RED_State` on page 104. In particular, I-cache tags, D-cache tags, and L2-cache tags must initialized before enabling the caches. The iTLB and dTLB also must be initialized before enabling memory management.

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.

11.4.2 Warm Reset (WMR)

A warm reset occurs when the J_RST_P pin is asserted and then deasserted. When a warm reset is received, all other resets and traps except POR are ignored. Warm reset has the same trap type and vector as power-on reset: a trap type of 001_{16} at physical address offset 20_{16}. Software can distinguish between POR and the various sources of warm reset by checking the RSET_STAT register. When a warm reset occurs, the memory controller will place the DRAM in self-refresh state before resetting itself to preserve the state of DRAM. Warm reset does not automatically do BIST testing, but BIST testing can be initiated by software after the reset is completed. After warm reset, strand 0 of the first available physical core begins executing at the RED_state_trap_vector base plus POR offset, while the remaining strands start out inactive.

After a warm reset, software must initialize values specified as unknown in *Machine State after Reset and in RED_State* on page 104. If there was a clean shutdown, the primary instruction, primary data, and L2 caches and main memory are still valid. Otherwise, I-cache tags, D-cache tags, and L-cache tags should be initialized before enabling the caches. The iTLB and dTLB also must be initialized before enabling memory management.

Note that if a warm reset is received without software first placing the chip in a quiescent state, the hardware will still maintain the state of the primary instruction, primary data, L2 caches, main memory, and all error registers/logs. However, the caches and main memory may no longer be completely coherent after the warm reset as any transactions in flight when the warm reset was received will have been lost. In particular, dirty lines in the process of being written back to main memory may have been dropped.

11.5 Strand Resets

Strands receive resets via writes to the INT_VEC_DIS register. Strand resets do not set any bits in the RSET_STAT register.

11.5.1 Warm Reset (WMR)

A strand (virtual processor) can be sent a warm reset (WMR) via the INT_VEC_DIS register. The warm reset generates a POR, which has a trap type of 001_{16} at physical address offset 20_{16}. Software can distinguish a strand warm reset from a chipwide warm reset by reading the RSET_STAT register. Since strand resets do not set any

bits in this register and software will zero the chipwide reset bits after the reset sequence has been completed, a RSET_STAT with all zero source bits will indicate to the strand that it received a strand warm reset.

11.5.2   Externally Initiated Reset (XIR)

An externally initiated reset is sent to a virtual processor via a write to the INT_VEC_DIS register; it causes an XIR, which has a trap type of 003_{16} at physical address offset 60_{16}. It has higher priority than all other resets except WMR. XIR is used for system debug.

11.5.3   Watchdog Reset (WDR) and `error_state`

A SPARC V9 WDR can be generated from the virtual processor reset register, INT_VEC_DIS, and traps to physical address offset 40_{16} and trap type 002_{16}. In addition, when a strand encounters a trap when TL = *MAXTL*, it enters `error_state` and signals itself internally to take a WDR trap. CWP updates due to window traps that cause watchdog traps are the same as the no watchdog trap case.

11.5.4   Software-Initiated Reset (SIR)

An SIR interrupt can be generated from the virtual processor reset register, INT_VEC_DIS. A software-initiated reset is also invoked on a strand by issuing an SIR instruction while operating in hyperprivileged mode. This strand reset has a trap type of 004_{16} at physical address offset 80_{16}.

11.6   Strand Suspension

UltraSPARC T1 uses a different set of primitives than UltraSPARC Architecture for controlling strand activity.

An active strand can be placed in the inactive state via a pair of mechanisms, referred to as "halt" and "idle." A "resume" mechanism is provided which causes halted or idle strands to resume execution. In addition, halted and idle strands both respond to resets and halted strands respond to interrupts as well (an idle strand ignores interrupts).

A strand may place itself in the halted state by executing a "halt" instruction. HALT is a synthetic instruction that maps to a WRASR %asr26 with a data value that has bit 0 clear. A halted strand does not execute any instructions beyond the halt. While

in the halted state, the strand will respond to interrupts, resumes, and resets. Receipt of an interrupt will take the strand to the active state, at which point if PSTATE.ie = 1, it will take the interrupt. Once the interrupt is serviced the strand will resume execution of the instruction following the halt (that is, the strand remains active). If PSTATE.ie = 0, the interrupt will remain pending and the strand will resume execution with the instruction following the halt. If the halted strand is sent a resume, it will resume execution with the instruction following the halt. Finally, if the halted strand is sent a reset, it will take a reset trap of the appropriate reset type.

A strand may be placed in the idle state by receiving an idle message. The idle message is generated via the INT_VEC_DIS register. An idle strand does not execute any instructions beyond where it received the idle message. While in the idle state, the strand will respond to resume or resets only. Interrupts will have no effect on idle strands. If the idle strand is sent a resume, it will resume execution where it left off. If the idle strand is sent a reset, it will take a reset trap of the appropriate reset type.The following diagram shows the transitions between the halted, idle, and active states. FIGURE 11-1 shows the transitions between the halted, idle, and active states

FIGURE 11-1 Transitions Between Halted, Idle, and Active States
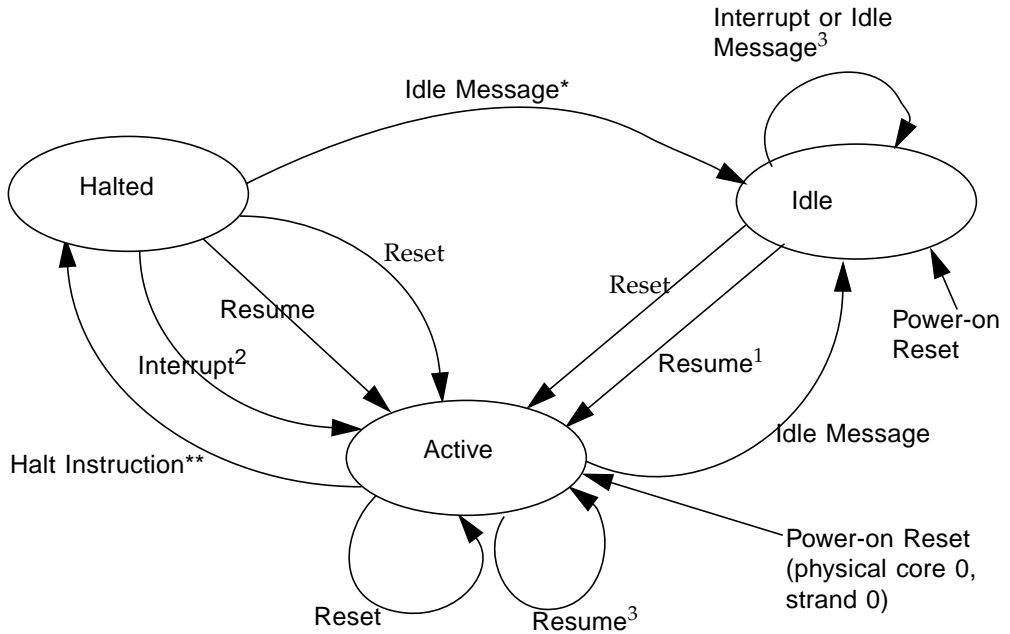
1 — Illegal when immediately following power-on reset; otherwise legal.
2 — Wakes up strand, but interrupt serviced only if PSTATE.ie = 1.
3 — Ignored.
* — When the strand goes through this arc, it may actually execute a few instructions
that follow the halt instruction which put it in the Halt state.
** — If an interrupt is pending when the HALT instruction is issued, the strand will transition
to the halted state and then back to the active state, effectively making the halt instruction
a NOP when interrupts are pending.

Notes (1) Placing a strand in the idle or halted state has no effect on cache coherence. Caches will continue to maintain coherence even if all strands that access that cache are placed in the idle or halted states.

(2) Error logging will continue to take place even while a strand is in the idle or halted state (for example, a modular arithmetic error will update the SPARC Error Status register of an idle or halted strand).

(3) Repeatedly sending a strand resume message (or idle/resume message pairs) in a tight loop can prevent the receiving strand from making forward progress. This is most likely to occur when the receiving strand has its instruction cache disabled or repeatedly takes a long-latency trap (such as an ITLB miss). Spacing the resume messages by 10 microseconds or more should be sufficient to allow the receiving strand to make forward progress.

Programming Notes (1) UltraSPARC T1 does not provide a mechanism for determining when a strand that has been sent an idle message actually becomes idle or when a strand that executes a halt instruction retires the halt instruction.

(2) If a strand desires to send an interrupt to itself to take it out of the halted state, a race condition exists where the interrupt could be received before the HALT instruction completes. To avoid this race, the following sequence can be used:
    set PSTATE.ie = 0
    set up interrupt
    HALT
    set PSTATE.ie = 1
Having interrupts disabled while setting up the interrupt guarantees that the interrupt will not be taken before the HALT instruction. The halted state is exited when the interrupt is received, even though PSTATE.ie = 0. Setting PSTATE.ie = 1 will result in the interrupt being taken.

(3) The HALT instruction can be used as an interrupt barrier. If, while expecting an interrupt, PSTATE.ie = 1 and a HALT instruction is executed, UltraSPARC T1 guarantees that the interrupt will be taken before any instructions following the HALT are executed.

11.7   RED_state

RED_state is an acronym for Reset, Error, and Debug State. It serves two mutually exclusive purposes:

1. An indication, during trap processing, that there are no more available trap levels —that is, if another nested trap is taken, the strand will enter error_state and halt.

2. Provision of a restricted 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 when TL < *MAXTL* (if TL = *MAXTL*, the strand enters error_state)
- Setting of HPSTATE.red by system software

RED_state is indicated by HPSTATE.red = 1, regardless of the value of TL. Executing a DONE or RETRY instruction in RED_state restores the stacked copy of the HPSTATE register, which zeroes the HPSTATE.red flag if it was zero in the stacked copy. System software can also set or clear the HPSTATE.red flag with a WRHPR instruction, which also forces the strand to enter or exit RED_state, respectively. In this case, the WRHPR instruction should be placed in the delay slot of a jump instruction, so that the PC can be changed in concert with the state change.

> **Note** | Setting TL = *MAXTL* using a WRHPR instruction neither sets RED_state nor alters any other machine state. The values of RED_state and TL are independent.

A reset or trap that sets HPSTATE.red to 1 (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 I-cache, 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 is always disabled in RED_state (this overrides the enable bit in the LSU Control register).

When HPSTATE.red is explicitly set to 1 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 maintain coherence while in RED_state.

11.8 RED_state Trap Vector

When a strand 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_state trap and takes the values:

- POR or WMR: 20_{16}
- XIR: 60_{16}
- WDR: 40_{16}
- SIR: 80_{16}
- other: $A0_{16}$

In an UltraSPARC T1 virtual processor, the RSTV base address is *RSTVADDR* FFFF FFFF F000 0000_{16}.

11.9 Machine State after Reset and in RED_State

TABLE 11-10 and TABLE 11-11 show processor state created as a result of any reset, or after entering RED_state.

TABLE 11-7 Architectural Processor State After Reset and When Entering RED_state

Name	Fields	POR	WMR (strand)	WDR	XIR	SIR	RED_State
Integer registers		Unknown	Unchanged				
Floating-Point registers		Unknown	Unchanged				
RSTVADDR		FFFF FFFF F000 0000_{16}					
PSTATE	ag	0 (Alternate globals not selected)					
	cle	0 (Current not little endian)					
	tle	0 (trap not little endian)					
	ig	0 (Interrupt globals not selected)					
	mg	0 (MMU globals not selected)					
HPSTATE	(bit 11)	0 (must be set to 1 by software)					
	red	1 (RED_state)					
	hpriv	1 (Hyperprivileged mode)					
	tlz	0 (tlz traps disabled)					
TBA	tba_high49	Unknown	Unchanged				

TABLE 11-7 Architectural Processor State After Reset and When Entering RED_state *(Continued)*

Name	Fields	POR	WMR (strand)	WDR	XIR	SIR	RED_State
HTBA	tba_high50	Unknown	Unchanged				
Y		Unknown	Unchanged				
PIL		Unknown	Unchanged				
CWP		Unknown	Unchanged except for register window traps				
TPC[TL]		Unknown	PC				
TNPC[TL]		Unknown	NPC				
CCR		Unknown	Unchanged				
ASI		Unknown					
TSTATE[TL]	gl	Unknown		GL			
	ccr	Unknown		CCR			
	asi	Unknown		ASI			
	pstate	Unknown		PSTATE			
	cwp	Unknown		CWP			
HTSTATE[TL]	hpstate	Unknown		HPSTATE			
TICK	npt	1		Unchanged			
	counter	Unknown	Count				
CANSAVE		Unknown	Unchanged				
CANRESTORE		Unknown	Unchanged				
OTHERWIN		Unknown	Unchanged				
CLEANWIN		Unknown	Unchanged				
WSTATE	other	Unknown	Unchanged				
	normal	Unknown	Unchanged				
HVER	manuf	$003E_{16}$					
	impl	0023_{16}					
	mask	Mask dependent (4 bits major, 4 bits minor)					
	maxtl	6					
	maxgl	3					
	maxwin	7					
GSR	all	0	Unchanged				
FSR	all	0	Unchanged				
FPRS	all	Unknown	Unchanged				
TICK_CMPR	int_dis	1 (off)		Unchanged			
	tick_cmpr	Unknown	Unchanged				

TABLE 11-8　UltraSPARC T1-Specific Processor State After Reset and When Entering RED_state

Name	Fields	POR	WMR (strand)	WDR	XIR	SIR	RED_State
PCR	all	0 (off)		Unchanged			
PIC		0		Unchanged			
LSU Control register	all	0 (off)					
VA_WATCHPOINT		Unknown	Unchanged				
I/D/L2 tags and data		Unknown	Unchanged				
L2 directory		All Invalid	Unchanged				
Store buffer		Empty	Empty	Unchanged			Empty
iTLB/dTLB	Mappings	Unknown	Unchanged				
Partition identifier		0		Unchanged			
SPARC Error Enable register	all	0 (trapping disabled)		Unchanged			
SPARC Error Status register	all	Unknown	Unchanged				
SPARC Error Address register		Unknown	Unchanged				
L2 Error Enable register	all	0 (reporting disabled)		Unchanged			
L2 Error Status register	all	Unknown	Unchanged				
L2 Error Address register		Unknown	Unchanged				
DRAM Error Status register	all	Unknown	Unchanged				
DRAM Error Address register		Unknown	Unchanged				
All I/O error registers		Unknown	Unchanged				

Note　On a chipwide warm reset (WMR), the WMR (strand column) applies, except that the TPC[TL] and TNPC[TL] are both unknown, and the FSR and GSR are both set to 0.

11.10　Boot Sequence

11.10.1　Overview of Software Initialization Sequence

Assumptions:

- L2 Tag, Data, and VUAD arrays, when written (BISTed) to zeros, are initialized to empty with good parity and good ECC.
- L2 Directory (of L1 tags) is marked invalid on reset.
- L1 I-cache, L1 D-cache, when written (BISTed) to zeros, initialized to good parity.
- L1 I-tags, L1 D-tags need to be explicitly ASI written to invalid, with good parity.
- Integer Register File – Explicit SW init.
- Other Register Files – Explicit SW init.
- ITLB, DTLB – cleared in SW with `ASI_*TLB_INVALIDATE_ALL`.
- Stream Unit (MA arrays) – Explicit SW init.
- Main Memory – Fast ECC initialization with Block Init Store instructions.

Sequence:

1. Read RSET_STAT register, which indicates POR.

2. Initialize CLK_DIV register with desired ratios.

3. Clear error logs. (Alternatively, this could be moved to later.)

4. Initialize (JBUS) registers on I/O Bridge chip that need reset to take effect.

5. Write to I/O Bridge chip, to initiate warm reset.

6. Come out of warm reset, again at reset vector.

7. Read RSET_STAT register, which indicates warm reset, with clock change.

8. If at-speed BIST is desired, launch BIST on L2 cache and local core's L1 caches; otherwise, launch BISI to initialize the L2 cache and local core's L1 caches.

9. Initialize local core's L1 tags.

10. Wait for BIST_DONE indications.

11. Enable error detection on L2 cache and local L1 caches.

12. Enable L2 cache and local L1 caches.

13. Copy bootstrap code into L2 cache, using Block Init Store instructions.

14. Branch to bootstrap code (now executing from cache).

15. Copy/decode code segments from PROM space to cacheable space, using Block Init Store instructions.

16. Initialize DRAM interface blocks.

17. Force refresh asynchronicity, by zeroing out the refresh counters on each DRAM controller, at precise intervals (N/4 of the refresh interval).

18. Initialize main memory, using Block Init Store instructions.

19. Initialize rest of blocks on the chips.

20. Send reset trap interrupts to strand 0 on other available cores

21. Strand 0 in each available physical core initializes physical core state (such as L1 BIST and enable L1$).

22. Strand 0 on each cores send reset trap interrupts to strands 1–3 on same core.

23. All virtual processors initialize strand-specific state.

24. Hand off to POST.

11.10.2 Overview of Warm Reset Software Initialization Sequence

Assumptions:

- Need to check whether error reset or SW-generated reset.
- What needs to be reinitialized?
  - D-cache tags look like valid is cleared, so parity is undefined.
  - Directory is cleared, so L1-I and L1-D need to be invalidated.

Sequence:

1. Read RSET_STAT register, which indicates warm reset.

2. Check local error logs.

3. Check remote error logs (for example, in IO Bridge chip).

4. If errors, go to crash-dump handling.

5. If no errors, initialize/clear L1 caches.

6. Turn on caches.

7. Initialize strand-specific state.

8. Send reset trap interrupts to other available virtual processors.

9. All virtual processors initialize strand-specific state.

10. Strand 0 in each available physical core initializes physical core state (such as clear and enable L1$).

11. Continue, as desired by SW.

11.11 Reset and Halt/Idle/Resume Summary

TABLE 11-12 summarizes the actions taken for chipwide reset and power management commands. In the table, BASE = FFFF FFFF F000 0000$_{16}$ (*RSTVADDR*).

TABLE 11-9 Chipwide Reset and Power Management Commands Actions

Event	Chip State	Auto-BIST	Physical Core 0, Strand 0	All other strands	Boot Vector
			Virtual Processor		
Power-on reset pin asserted	Awake	No	Active	Idle	BASE | 20$_{16}$
Warm reset pin asserted	Awake	No	Active	Idle	BASE | 20$_{16}$

TABLE 11-13 summarizes the actions taken for strand interrupt, reset, halt, idle, and resume commands. In the table, BASE = FFFF FFFF F000 0000$_{16}$ (*RSTVADDR*).

TABLE 11-10 Strand Interrupt, Reset, Halt, Idle, and Resume Commands Actions

Event		Target Strand	Boot Vector
INT_VEC_DIS or `ASI_SWVR_UDB_INTR_W` interrupt	Strand idle	No effect	
INT_VEC_DIS or `ASI_SWVR_UDB_INTR_W` interrupt	Strand active or halted	Interrupt posted to `ASI_SWVR_INTR_RECEIVE`	—
INT_VEC_DIS reset		Active	BASE | VECTOR
INT_VEC_DIS idle		Idle	—
Execution of halt instruction		Halted	—
INT_VEC_DIS resume	Strand active	No effect	
INT_VEC_DIS resume	Strand idle or halted	Active	Resumes from suspended PC

> **Note** | After a POR or being brought out of the sleep state, an idle strand must be activated by a reset, not a resume.

Error Handling

This chapter describes the error detection, reporting, and handling for the UltraSPARC T1 chip.

12.1 Error Classes

Errors on UltraSPARC T1 fall into three classes: fatal (FE), uncorrectable (UCE), and correctable (CE).

Fatal errors are generated whenever the hardware detects a condition where an error has occurred and the extent to which the error may have propagated is unbounded. An example of a fatal error is a valid bit corruption in the L2 cache, which implies that global cache coherence has been lost. Since a fatal error may have corrupted key operating system or hypervisor data structures, fatal errors generate an immediate warm reset to the UltraSPARC T1 chip.

Uncorrectable errors are errors for which the hardware is unable to take corrective action, but the extent to which the error may have propagated is tightly bounded. Examples of uncorrectable errors include a data parity error on the DTLB, and a double-bit ECC error in the L2 cache. Note that the uncorrectable status for these errors refers only to the UltraSPARC T1 hardware. Software may be able to correct a hardware-uncorrectable error. For example, the data parity error in the DTLB may be able to be corrected in software by forcing the TLB entry with the bad parity to be invalid, which will then be reloaded over with a new TTE entry and good parity on the subsequent TLB miss. Likewise, while a double-bit ECC error in the L2 cache cannot be corrected by software, the extent to which the error could have propagated is limited to the address space of any process that has access to that memory location, and software may be able to keep the system running by killing all processes that could be affected by the error and then scrubbing the bad memory location. Uncorrectable errors are reported through several traps, including precise and disrupting traps.

Finally, correctable errors are errors that the UltraSPARC T1 hardware can correct automatically. These errors do not corrupt any system or process state, although they may have a performance impact on the process encountering them. Examples of correctable errors are single-bit ECC errors in the L2 cache or a data parity error in the instruction cache. Correctable errors generate a disrupting *ECC_error* trap. This trap can be used by software to keep track of correctable error frequency and location to aid in diagnosing failed and failing hardware components.

12.2 CMT Error Overview

Errors detected in the CMT and memory subsystem are reported in three sets of error registers: SPARC, L2 cache, and DRAM. Errors are reported in the SPARC Error register set in program order. Errors are reported in the L2 cache and DRAM error sets in the order the errors occur. The L2 cache Error Enable register controls whether errors associated with the L2 cache and DRAM are reported back to the initiator. If the L2 cache error enable bit (ceen or nceen, depending on whether the error is correctable or uncorrectable) is not set, error information is not reported back to the initiator. For the uncorrectable case (nceen clear), this means that bad data will be executed/used, and thus the nceen bit is only intended to be cleared during heavily controlled phases of diagnostic operation. The per-strand SPARC Error Enable registers control whether the strand takes a trap as a result of any reported errors.

The SPARC, L2, and DRAM error registers have the ability to log detailed information for a single error. The same error logs information for all three classes of errors: fatal, uncorrectable, and correctable. Fatal and uncorrectable errors will overwrite earlier correctable error information. The error registers have bits to indicate if multiple errors have occurred. There are two bits for multiple errors: meu (multiple uncorrectable errors) and mec (multiple correctable errors). TABLE 12-1 lists the multiple error logging and overwrite behavior of the error registers (a CE for the main logged error with the meu bit set is not possible due to FE and UCE being higher priority than CE in logging).

TABLE 12-1 Multiple Error Logging and Overwrite Behavior of the Error Registers

Main Logged Error	Additional CE Status Bit Logged?	meu	mec	Description
FE or UCE	No	0	0	Single FE or UCE encountered and logged.
FE or UCE	Yes	0	0	Single FE or UCE encountered and logged. One or more correctable errors encountered before the FE or UCE but details on the correctable errors not logged.

Main Logged Error	Additional CE Status Bit Logged?	meu	mec	Description
FE or UCE	No	0	1	Single FE or UCE encountered and logged. One or more correctable errors encountered simultaneous to or after the FE or UCE but details on the correctable errors not logged.
FE or UCE	Yes	0	1	Single FE or UCE encountered and logged. One or more correctable errors encountered before the FE or UCE but details on the correctable errors not logged. One or more correctable errors encountered simultaneous to or after the FE or UCE but details on the correctable errors not logged.
FE or UCE	No	1	0	Two or more FE or UCEs encountered, details on the highest priority FE or UCE encountered on the earliest cycle with an error logged (that is, a higher priority error encountered on a later cycle does not overwrite the earlier error).
FE or UCE	Yes	1	0	Two or more FE or UCEs encountered, details on the highest priority FE or UCE encountered on the earliest cycle with an error logged (that is, a higher priority error encountered on a later cycle does not overwrite the earlier error). One or more correctable errors encountered before the FE or UCE but details on the correctable errors not logged.
FE or UCE	No	1	1	Two or more FE or UCEs encountered, details on the highest priority FE or UCE encountered on the earliest cycle with an error logged (that is, a higher priority error encountered on a later cycle does not overwrite the earlier error). One or more correctable errors encountered simultaneous or later than the first FE or UCE but details on the correctable errors not logged.
FE or UCE	Yes	1	1	Two or more FE or UCEs encountered, details on the highest priority FE or UCE encountered on the earliest cycle with an error logged (that is, a higher priority error encountered on a later cycle does not overwrite the earlier error). One or more correctable errors encountered simultaneous or later than the first FE or UCE but details on the correctable errors not logged. One or more correctable errors encountered before the FE or UCE but details on the correctable errors not logged.
CE	No	0	0	Single CE encountered and logged.
CE	No	0	1	Two or more CEs encountered, details on the highest priority CE encountered on the earliest cycle with an error logged (that is, a higher priority error encountered on a later cycle does not overwrite the earlier error).

TABLE 12-2 lists a high-level summary of the CMT error handling.

TABLE 12-2   CMT Error Handling Summary

Error and Abbreviation	Severity	Logs	Notes
ITLB Data Parity (IMDU)	Uncorrectable	SPARC	Error scrubbed by SW invalidate
ITLB CAM Parity (IMTU)	Uncorrectable	SPARC	Only detectable by SW scrubber
DTLB Data Parity (DMSU for stores; DMDU for loads and ASI access)	Uncorrectable	SPARC	Error scrubbed by SW invalidate
DTLB CAM Parity (DMTU)	Uncorrectable	SPARC	Only detectable by SW scrubber
I-cache Data Parity (IDC)	Correctable	SPARC	Error scrubbed by HW miss fill
I-cache Tag Parity (ITC)	Correctable	SPARC	Error scrubbed by HW invalidate of all ways in the set and miss fill
D-cache Data Parity (DDC)	Correctable	SPARC	Error scrubbed by HW miss fill
D-cache Tag Parity (DTC)	Correctable	SPARC	Error scrubbed by HW invalidate of all ways in the set and miss fill
Int RegFile ECC: transient single (IRC)	Correctable	SPARC	Error scrubbed in HW
Int RegFile ECC: hard single, multiple (IRU)	Uncorrectable	SPARC	
FP RegFile ECC: transient single (FRC)	Correctable	SPARC	Error scrubbed in HW
FP RegFile ECC: hard single, multiple (FRU)	Uncorrectable	SPARC	
Mod Arith Memory Parity (MAU)	Uncorrectable	SPARC	Can be corrected via SW retry
PIO Read or Ifetch Error (NCU)	Uncorrectable	SPARC	
L2 Data ECC: single (LDAC for SPARC access; LDWC for writeback: LDRC for dma; LDSC for hardware scrubber)	Correctable	L2	L2 may need SW scrub
L2 Data ECC: multiple (LDAU for SPARC access; LDWU for writeback; LDRU for dma; LDSU for hardware scrubber)	Uncorrectable	(SPARC), L2	
L2 Tag ECC: single only (LTC)	Correctable	L2	L2 may need SW scrub
L2 Directory Parity (LRU)	Fatal	L2	Detected on scrub, causes warm reset
L2 VUAD Parity (LVU)	Fatal	L2	VAD errors only (no parity on U), causes warm reset
DRAM ECC: single nibble (DAC for access, DSC for hardware scrub)	Correctable	L2, Mem	Memory may need SW scrub
DRAM ECC: multiple nibble (DAU for access, DSU for scrub)	Uncorrectable	(SPARC), L2, Mem	
DRAM address out of bounds (DBU)	Uncorrectable	(SPARC), L2, Mem	Most likely the result of a software error

12.3 SPARC Error Descriptions

12.3.1 ITLB Data Parity Error (IMDU)

Each ITLB data entry is protected with parity. ITLB data parity is checked with each instruction translation as well as with loads to `ASI_ITLB_DATA_ACCESS_REG`. When a parity error is detected the error information is captured in the SPARC Error Status and SPARC Error Address registers. For a load from `ASI_ITLB_DATA_ACCESS_REG`, if the SPARC Error Enable nceen bit is set, a precise *data_access_error* trap is generated to the requesting strand. For a translation, if the SPARC Error Enable nceen bit is set, a precise *instruction_access_error* trap is generated to the requesting strand.

> **Note** | If the SPARC Error Enable nceen bit is cleared, the ITLB data entry with the parity error will be used for translation. Depending on which bit was in error, this could lead to an access to an unexpected address (for an address bit error), an *instruction_access_exception* for user access to a p bit in error, etc.

12.3.2 ITLB Tag Parity Error (IMTU)

Each ITLB tag entry is protected with parity. The ITLB tag parity is only checked when accessed via a load from `ASI_ITLB_TAG_READ_REG`, not during normal instruction translations. This implies that the parity bit is present for software scrubbing only. When reading an ITLB entry with parity error the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable nceen bit is set, a precise *data_access_error* trap is generated to the requesting strand.

12.3.3 DTLB Data Parity Error on Load and Atomics (DMDU)

Each DTLB data entry is protected with parity. The DTLB data parity is checked with each atomic, load or store translation as well as with loads to `ASI_DTLB_DATA_ACCESS_REG`. This section describes the error behavior for a load or atomic translation or a load from the `ASI_DTLB_DATA_ACCESS_REG`. *DTLB Data Parity Error on Store (DMSU)* on page 116 describes the error behavior for a store translation. When a parity error is detected the error information is captured in the

SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable nceen bit is set, a precise *data_access_error* trap is generated to the requesting strand.

> **Note** | If the SPARC Error Enable nceen bit is cleared, the DTLB data entry with the parity error will be used for translation. Depending on which bit was in error, this could lead to an access to an unexpected address (for an address bit error), a *data_access_exception* for user code access to a p bit in error, etc.

> **Implementation Note** | If the SPARC Error Enable nceen bit is set, the DTLB data entry with the parity error will still be checked for access privileges by UltraSPARC T1. Since *data_access_exception* and *fast_data_access_protection* are higher priority than *data_access_error*, those traps may be taken instead of *data_access_error*. DMDU examples that will generate a *data_access_exception* instead are (a) the nfo bit is flipped from 0 to 1 and the access is non-nofault, (b) the priv bit is flipped from 0 to 1 and the access is from user privilege, or (c) the e bit is flipped from 0 to 1 and the access is no-fault.

12.3.4   DTLB Data Parity Error on Store (DMSU)

Each DTLB data entry is protected with parity. The DTLB data parity is checked with each atomic, load or store translation as well as with loads to ASI_DTLB_DATA_ACCESS_REG. This section describes the error behavior for a store translation. Section 12.3.3 describes the error behavior for a load or atomic translation or a load from the ASI_DTLB_DATA_ACCESS_REG. When a parity error is detected the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable nceen bit is set, a precise *data_access_error* is generated to the requesting strand.

> **Note** | If the SPARC Error Enable nceen bit is cleared, the DTLB data entry with the parity error will be used for translation. Depending on which bit was in error, this could lead to an access to an unexpected address (for an address bit error), a *data_access_exception* for user code access to a p bit in error, etc.

12.3.5   DTLB Data Parity Error on PREFETCH

If a DTLB parity error is detected during a prefetch, the parity error is ignored and the prefetch continues, using the translated address. If the prefectch is from I/O space, it is ignored. If it is not from I/O space, useless data may be prefetched into the L2 cache or it may cause an out-of-bounds data access error (DAU/DBU) if the error causes the translated address to point to non-existent memory.

12.3.6   DTLB Tag Parity Error (DTU)

Each DTLB tag entry is protected with parity. The DTLB tag parity is only checked when accessed via a load from ASI_DTLB_TAG_READ_REG, not during normal instruction translations. This implies that the parity bit is present for software scrubbing only. When reading a DTLB entry with parity error the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable nceen bit is set, a precise *data_access_error* trap is generated to the requesting strand.

12.3.7   I-cache Data Parity Error (IDC)

Each 32-bit instruction (plus a corresponding hardware-specific performance acceleration bit called the "switch" bit) in the I-cache data array is protected with parity. The I-cache data parity is checked with each instruction fetch. When a parity error is detected the error information is captured in the SPARC Error Status register (SESR) and SPARC Error Address registers. In addition, if the SPARC Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the requesting strand. Hardware corrects the error by fetching the instruction line from the L2 cache and replacing the erroneous data with the data from the L2 cache.

Implementation Note | If an IDC is detected on an instruction that also detects an IRC, the IDC should be logged and the IRC should set MEC. Instead, the hardware logs both IDC and IRC in the SESR, with SEAR containing IRF information. From a practical standpoint, the chance of this occurring naturally is truly infinitesimal, so this case is only of interest to validation code.

12.3.8 I-cache Tag Parity Error (ITC)

The I-cache tag and valid bit are protected with parity. The I-cache tag parity is checked with each instruction fetch. The parity is checked for each of the four possible ways in the set. If a parity error is found in any of the ways, the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the requesting strand. All four ways in the set are invalidated and a cache miss refill is forced.

Implementation Note | If an ITC is detected on an instruction that also detects an IRC, the ITC should be logged and the IRC should set MEC. Instead, the hardware logs both ITC and IRC in the SESR, with SEAR containing IRF information. From a practical standpoint, the chance of this occurring naturally is truly infinitesimal, so this case is only of interest to validation code.

12.3.9 D-cache Data Parity Error (DDC)

Each byte in the D-cache is protected with parity. The D-cache data parity is checked with each data load. When a parity error is detected the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the requesting strand. Hardware corrects the error by fetching the data line from the L2 cache and replacing the error data with the data from the L2 cache. Parity errors are ignored on data stores, as new parity can be generated due to a parity bit being kept for each byte in the cache.

12.3.10 D-cache Tag Parity Error (DTC)

The D-cache tag and valid bit are protected with parity. The D-cache tag parity is checked with each data load. The parity is checked for each of the four possible ways in the set. If a parity error is found in any of the ways, the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition,

if the SPARC Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the requesting strand. All four ways in the set are invalidated and a cache miss refill is forced.

12.3.11   IRF Correctable ECC Error (IRC)

The integer register file (IRF) is protected by SECDED ECC. ECC is checked for each register operand of an instruction. When a correctable ECC error is detected, the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the strand that encountered the error. Hardware automatically corrects the register contents.

Notes | (1) IRF single-bit errors are corrected by a microtrap mechanism that attempts to retry the instruction up to three times. This microtrap mechanism will succeed in correcting single-bit transient errors, but will be unable to correct a single-bit persistent error. Persistent single-bit errors are thus classified as IRF uncorrectable ECC errors (IRU); in this case, if the SPARC Error Enable ceen bit is cleared (or PSTATE.ie is cleared when HPSTATE.hpriv is set), hardware will log an IRC error for the retries and then take an IRU when the three retries have completed unsuccessfully.

(2) Due to the microtrap retry mechanism for correcting IRF single-bit errors, the *ECC_error* trap will be taken before the instruction that encountered the single-bit error has completed execution. This implies that a persistent single-bit error will need to be detected in the *ECC_error* trap handler when SPARC Error Enable ceen is set (and PSTATE.ie is set if HPSTATE.hpriv is set), as the entry into the handler will occur before the three retries can cause the IRU error.

(3) If multiple IRF correctable ECC errors are encountered for the same instruction (for example, rs1 and rs2 both have correctable ECC errors), this will be treated as a single IRC error by the SPARC Error Status register, and the syndrome and source register will be logged for the correctable error in priority of rs1 > rs2 > rs3. rs3 is only checked for instructions that use it as a source operand, which includes stores, CAS, and MOVcc (rd is rewritten to itself for a MOV that doesn't meet the condition).

(4) If both uncorrectable and correctable IRF ECC errors are encountered for the same instruction (for example, rs1 has a correctable error, while rs2 has an uncorrectable error), correction of the correctable error for use in the instruction will be disabled (which does not matter since the other operand has an uncorrectable error), and only the uncorrectable IRU error will be indicated to the SPARC Error Status register. Hardware will still correct the register contents for the correctable error. Note that this overriding of a correctable error by an uncorrectable error implies that if the SPARC Error Enable ceen bit is set but the SPARC Error Enable nceen bit is cleared, no trap will be generated for the correctable error.

(5) With the way register management works in UltraSPARC T1, there can be unexpected behavior with IRF error propagation. This affects %l, %o, %i, but not %g regs. The current register window for a thread is in a particular high-speed register file. Saves and restores (and wrpr %cwp) actually copy registers between the high-speed file and a "backing store" that contains the rest of the register windows. If there is an error in the current register window, a "save" will copy/move the error to the backing store, then allocate a new frame that still has the error in it, thus replicating the error in another frame, effectively in an uninitialized register in that frame. If that register is initialized (written to) before getting dereferenced, the replicated error will not be seen.

Implementation Note | If an IRC is detected on an instruction that also detects either an IDC or ITC, the IDC/ITC should be logged and the IRC should set MEC. Instead, the hardware logs both IDC/ITC and IRC in the SESR, with SEAR containing IRF information. From a practical standpoint, the chance of this occurring naturally is truly infinitesimal, so this case is only of interest to validation code.

12.3.12   IRF Uncorrectable ECC Error (IRU)

The integer register file is protected by SECDED ECC. ECC is checked for each register operand of an instruction. When an uncorrectable ECC error is detected the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable nceen bit is set, a precise *internal_processor_error* trap is generated to the strand that encountered the error.

Notes (1) Software will need to initialize all integer registers in all register windows after reset to prevent code which reads uninitialized registers from causing *internal_processor_error* traps.

(2) If multiple IRF uncorrectable ECC errors are encountered for the same instruction (for example, rs1 and rs2 both have uncorrectable ECC errors), this will be treated as a single IRU error by the SPARC Error Status register, and the syndrome and source register will be logged for the uncorrectable error in priority of rs1 > rs2 > rs3. rs3 is only checked for instructions that use it as a source operand, which includes stores, CAS, and MOVcc (rd is rewritten to itself for a MOV that doesn't meet the condition).

(3) If both uncorrectable and correctable IRF ECC errors are encountered for the same instruction (for example, rs1 has a correctable error, while rs2 has an uncorrectable error), correction of the correctable error for use in the instruction will be disabled (which does not matter since the other operand has an uncorrectable error), and only the uncorrectable IRU error will be indicated to the SPARC Error Status register. Hardware will still correct the register contents for the correctable error. Note that this overriding of a correctable error by an uncorrectable error implies that if the SPARC Error Enable ceen bit is set but the SPARC Error Enable nceen bit is cleared, no trap will be generated for the correctable error.

Implementation PREFETCHA instructions check for IRU on their address
Note generation.

However, PREFETCH instructions do *not* check for IRU on their address generation.In the rare case where the prefetch has an IRU on the address generation, no error will be reported, and the prefetch may end up loading an unexpected cache line into the L2 cache (or may end up as a NOP due to a TLB miss or translation violation).

Note | With the way register management works in UltraSPARC T1, there can be unexpected behavior with IRF error propagation. This affects %l, %o, %i, but not %g regs. The current register window for a thread is in a particular high-speed register file. Saves and restores (and wrpr %cwp) actually copy registers between the high-speed file and a "backing store" that contains the rest of the register windows. If there is an error in the current register window, a "save" will copy/move the error to the backing store, then allocate a new frame that still has the error in it, thus replicating the error in another frame, effectively in an uninitialized register in that frame. If that register is initialized (written to) before getting dereferenced, the replicated error will not be seen.

The floating-point register file is protected by SECDED ECC. ECC is checked for each register operand of an instruction. When a correctable ECC error is detected the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the strand that encountered the error. Hardware automatically corrects the register contents.

Notes | (1) FRF single-bit errors are corrected by a microtrap mechanism that attempts to retry the instruction up to three times. This microtrap mechanism will succeed in correcting single-bit transient errors, but will be unable to correct a single-bit persistent error. Persistent single-bit errors are thus classified as FRF uncorrectable ECC errors (FRU); in this case, for an FP store, if the SPARC Error Enable ceen bit is cleared (or PSTATE.ie is cleared when HPSTATE.hpriv is set), hardware will log an FRC error for the retries and then take an FRU when the three retries have completed unsuccessfully. For other FP operation, on a persistent single-bit error, hardware will log an FRC error for the retries and then take an FRU when the three retries have completed unsuccessfully.

(2) Due to the microtrap retry mechanism for correcting FRF single-bit errors, the *ECC_error* trap will be taken before the instruction that encountered the single-bit error has completed execution for FP stores only. For other FP operations, the *ECC_error* trap will be taken after the instruction has completed execution. This implies that a persistent single-bit error for a FP store source will need to be detected in the *ECC_error* trap handler when SPARC Error Enable ceen is set (and PSTATE.ie = 1 if HPSTATE.hpriv = 1), as the entry into the handler will occur before the three retries can cause the FRU error.

(3) If multiple FRF correctable ECC errors are encountered for the same instruction (for example, rs1 and rs2 both have correctable ECC errors), this will be treated as a single FRC error by the SPARC Error Status register, and the syndrome and source register will be logged for the correctable error, with the error in the rs1 operand given priority over the error in the rs2 operand.

(4) If both uncorrectable and correctable FRF ECC errors are encountered for the same instruction (for example, rs1 has a correctable error, while rs2 has an uncorrectable error), both an FRC and FRU error will be indicated to the SPARC Error Status register (and their corresponding traps will be taken if SPARC Error Enable ceen and/or nceen bits, respectively, are set).

12.3.13   FRF Uncorrectable ECC Error (FRU)

The floating-point register file is protected by SECDED ECC. ECC is checked for each register operand of an instruction. When an uncorrectable ECC error is detected the error information is captured in the SPARC Error Status and SPARC Error Address registers. In addition, if the SPARC Error Enable nceen bit is set, a precise *internal_processor_error* trap is generated to the strand that encountered the error. Note that block store operations perform eight reads of the FRF.

See "Warning" below, regarding FRC/MEC.

Notes (1) Software will need to initialize all floating-point registers after reset to prevent code which reads uninitialized registers from causing *data_access_error* traps.

(2) If multiple FRF uncorrectable ECC errors are encountered for the same instruction (for example, rs1 and rs2 both have uncorrectable ECC errors), this will be treated as a single FRU error by the SPARC Error Status register, and the syndrome and source register will be logged for the uncorrectable error, with the error in the rs1 operand given priority over the error in the rs2 operand.

(3) If both uncorrectable and correctable FRF ECC errors are encountered for the same instruction (for example, rs1 has a correctable error, while rs2 has an uncorrectable error), both a FRC and FRU error will be indicated to the SPARC Error Status register (and their corresponding traps will be taken if SPARC Error Enable ceen and/or nceen bits, respectively, are set).

Warning	There is a corner case where an FRC is detected, but the hardware corrects the wrong FP register (of the same strand), effectively corrupting that wrong register. This can be detected in the error handler (conservatively) if (1) FRC and MEC are both logged, or (2) FRC is logged, but the specified register is still uncorrected. If either of these cases are detected, this should be treated as an uncorrectable error.

12.3.14 Modular Arithmetic Memory (MAU)

The modular arithmetic memory is protected by parity. If a parity error is detected on any MA memory access (either an MA operation or an MA memory store operation), the MA unit immediately aborts its asynchronous operation and logs the error in the SPARC Error Status register. Once the abort completes, if the SPARC Error Enable nceen bit is set and the ASI_MA_CONTROL_REG int bit is set, a disrupting *data_error* is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt and the normal *modular_arithmetic_interrupt* completion trap is not generated.

If the SPARC Error Enable nceen bit is set and the ASI_MA_CONTROL_REG int bit is cleared, a precise *data_access_error* trap is generated to the strand that issues any load from an ASI_MA_* register.

If the SPARC Error Enable nceen bit is cleared and the ASI_MA_CONTROL_REG int bit is set, a normal *modular_arithmetic_interrupt* is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt.

If the SPARC Error Enable nceen bit is cleared and the ASI_MA_CONTROL_REG int bit is cleared, the load access to ASI_MA_SYNC_REG will complete normally without generating an error trap.

12.3.15 I/O Load/Instruction Fetch (NCU)

Loads and instruction fetches to noncacheable I/O space (physical addresses $80\ 0000\ 0000_{16}$–$FF\ FFFF\ FFFF_{16}$) can encounter uncorrectable errors. When an uncorrectable error is detected for an I/O load or instruction fetch the error information is captured in the SPARC Error Status and SPARC Error Address registers. Additional error logging may take place in the IOP as described in *JBUS Interface (JBI)* on page 169. For an I/O load, if the SPARC Error Enable nceen bit is set, a precise *data_access_error* trap is generated to the requesting strand. For an instruction fetch, if the SPARC Error Enable nceen bit is set, a precise *instruction_access_error* trap is generated to the requesting strand.

12.4　SPARC Error Registers

12.4.1　`ASI_SPARC_ERROR_EN_REG`

Each strand has a hyperprivileged `ASI_SPARC_ERROR_EN_REG` register at ASI $4B_{16}$, VA{63:0} = 0. This register controls the generation of traps for errors that are reported to the strand. The format of the `ASI_SPARC_ERROR_EN_REG` register is shown in TABLE 12-3.

TABLE 12-3　SPARC Error Enable Register – `ASI_SPARC_ERROR_EN_REG` (ASI $4B_{16}$, VA 0_{16})

Bit	Field	Initial Value	R/W	Description
63:2	—	0	R	*Reserved*
1	nceen	0	RW	If set to 1, trap on uncorrectable error
0	ceen	0	RW	If set to 1, trap on correctable error

Notes　(1) Errors are always logged in the `ASI_SPARC_ERROR_STATUS_REG` and `ASI_SPARC_ERROR_ADDRESS_REG` regardless of the setting of the nceen and ceen bits in `ASI_SPARC_ERROR_EN_REG`. `ASI_SPARC_ERROR_EN_REG` only controls whether or not a trap is generated for the error.

(2) For a few of the multiple simultaneous error cases, proper handling by UltraSPARC T1 requires the SPARC Error Enable nceen bit to be set to 1, so the nceen bit should only be cleared (set to 0) during carefully controlled situations. An example of multiple simultaneous errors that are not handled properly when the SPARC Error Enable nceen bit is zero occurs when an instruction cache data error on the switch bit (special-case of IDC) happens at the same time as a parity error (IMDU) on the translation for the instruction that encounters the IDC error. UltraSPARC T1 may hang for this case if the error in the TLB entry corrupts a critical address bit. When the SPARC Error Enable nceen bit is set, the trap taken for the IMDU error prevents the hang case from arising.

Sun-Internal Note　*Running with* **nceen** *= 0 is intened for use only during bringup;* **nceen** *should always be set to 1 in released software.*

If a thread is running with nceen = 0 and a load to a floating-point register occurs that encounters an uncorrectable error, the issuing thread will hang until a warm reset occors. The processor should not normally be run with nceen = 0; when it is, floating-point loads and LDBLOCKF instructions should not be executed.

12.4.2 ASI_SPARC_ERROR_STATUS_REG

Each virtual processor (strand) has a hyperprivileged
ASI_SPARC_ERROR_STATUS_REG register at ASI $4C_{16}$, VA{63:0} = 0. Each status bit in this register is cleared by writing a 1 to its bit position. The error register is not cleared on reset, so software can examine its contents after an error-induced reset.

| Programming Note | Since this register is not cleared on reset, after a power-on reset the contents of this register are undefined and the bits could be in an illegal state that could not possibly be generated by any error combination (for example, multiple ue bits set with all other bits cleared). Operation while in this illegal state leads to undefined behavior for the register, so *software should always clear this register after a power-on reset.* |

| Note | meu, mec, and priv are not preserved across resets, which means that those bits are also cleared after fatal error (which invokes a warm reset. |

TABLE 12-4 defines the format of the ASI_SPARC_ERROR_STATUS_REG register.

TABLE 12-4 SPARC Error Status Register – ASI_SPARC_ERROR_STATUS_REG
(ASI $4C_{16}$, VA 0_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—	0	R	*Reserved*
31	meu	0	RW1C	Multiple uncorrected errors; one or more uncorrected errors were not logged.
30	mec	0	RW1C	Multiple corrected errors; one or more corrected errors were not logged (due to either having both uncorrected and corrected errors, or by having multiple corrected errors).
29	priv	0	RW1C	Set to 1 if error occurred while in privileged or hyperprivileged mode.
28	—	1	R	*Reserved*
27:26	—	0	R	*Reserved*
25	imdu	Preserved	RW1C	Set to 1 if the error was an IMMU TLB data parity error.
24	imtu	Preserved	RW1C	Set to 1 if the error was an IMMU TLB tag parity error.
23	dmdu	Preserved	RW1C	Set to 1 if the error was a DMMU TLB data parity error on a load.
22	dmtu	Preserved	RW1C	Set to 1 if the error was a DMMU TLB tag parity error.
21	idc	Preserved	RW1C	Set to 1 if the error was an I-cache data parity error.

Bit	Field	Initial Value	R/W	Description
20	itc	Preserved	RW1C	Set to 1 if the error was an I-cache tag parity error.
19	ddc	Preserved	RW1C	Set to 1 if the error was a D-cache data parity error.
18	dtc	Preserved	RW1C	Set to 1 if the error was a D-cache tag parity error.
17	irc	Preserved	RW1C	Set to 1 if the error was an IRF ECC correctable error.
16	iru	Preserved	RW1C	Set to 1 if the error was an IRF ECC uncorrectable error.
15	frc	Preserved	RW1C	Set to 1 if the error was a FRF ECC correctable error.
14	fru	Preserved	RW1C	Set to 1 if the error was a FRF ECC uncorrectable error.
13	ldau	Preserved	RW1C	Set to 1 if the error was a L2/DRAM ECC uncorrectable error.
12	ncu	Preserved	RW1C	Set to 1 if the error was an Ifetch/Load from I/O space uncorrectable error.
11	dmsu	Preserved	RW1C	Set to 1 if the error was a DMMU TLB data parity error on a store.
10	—	0	R	*Reserved*
9	mau	Preserved	RW1C	Set to 1 if the error was a Modular Arithmetic Memory parity error.
8:0	RSVD4	0	R	*Reserved*

If multiple errors occur in the same cycle, the meu and/or mec bit is set and only the highest priority error is logged based on the following priority table (TABLE 12-5) (errors with the same priority are mutually exclusive).

TABLE 12-5   Priority for Simultaneous Errors

Error	Priority	Bit Set if Higher Priority Error
DMSU	1	
IMTU	2	meu
DMTU	3	meu
NCU	3	meu
IRU	3	meu
FRU	3	meu
LDAU	4	meu
IMDU	5	meu
DMDU	6	meu
MAU	7	meu
ITC	8	mec (see Note)
DTC	9	mec
IRC	10	mec
FRC	10	mec
IDC	11	mec (see Note)
DDC	12	mec

The syndrome, priv, and address are captured for the highest priority error in that cycle.

Note | If ITC and IDC are detected on the same instruction (will be same cycle), only the IDC is logged. However, hardware still cleans up both errors.

Implementation Note | The determination of which errors (and nonerror traps) happen on the same cycle is under control of the UltraSPARC T1 implementation. Operations that logically happen in sequence may report their errors on the same cycle.

For example, while instruction translation is required to issue an instruction fetch, both are considered to happen at the same time in UltraSPARC T1 for the case where there is an error on the instruction fetch, and an instruction fetch that encountered both a privilege violation and an LDAU from the L2 cache would take the *instruction_access_error* trap over the *instruction_access_exception* trap, even though a sequential sequence of operations would prevent the instruction from ever being fetched due to the privileged violation on the translation.

If errors occur in a cycle when an error status bit is already set (indicating a previous error exists that hasn't been cleared from the error status register), TABLE 12-6 applies.

TABLE 12-6 Errors Occurring in a Cycle With Error Status Bit Already Set

Existing Error	Error	Priority	Bit Set if Highest-Priority Error	Bit Set if Higher-Priority Error In Same Cycle
DMSU/IMTU/DMTU/NCU/LDAU/IRU/FRU/IMDU/DMDU/MAU	DMSU	1	meu	N/A
DMSU/IMTU/DMTU/NCU/LDAU/IRU/FRU/IMDU/DMDU/MAU	IMTU	2	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/IRU/FRU/IMDU/DMDU/MAU	DMTU	3	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	NCU	3	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	IRU	3	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	FRU	3	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	LDAU	4	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	IMDU	5	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	DMDU	6	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	MAU	7	meu	meu
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	ITC	8	mec	mec
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	DTC	9	mec	mec
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	IRC	10	mec	mec
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	FRC	10	mec	mec

TABLE 12-6 Errors Occurring in a Cycle With Error Status Bit Already Set *(Continued)*

Existing Error	Error	Priority	Bit Set if Highest-Priority Error	Bit Set if Higher-Priority Error In Same Cycle
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	IDC	11	mec	mec
DMSU/IMTU/DMTU/NCU/LDAU/ IRU/FRU/IMDU/DMDU/MAU	DDC	12	mec	mec
ITC/DTC/IRC/FRC/IDC/DDC	DMSU	1	dmsu	N/A
ITC/DTC/IRC/FRC/IDC/DDC	IMTU	2	imtu	meu
ITC/DTC/IRC/FRC/IDC/DDC	DMTU	3	dmtu	meu
ITC/DTC/IRC/FRC/IDC/DDC	NCU	3	ncu	meu
ITC/DTC/IRC/FRC/IDC/DDC	IRU	3	iru	meu
ITC/DTC/IRC/FRC/IDC/DDC	FRU	3	fru	meu
ITC/DTC/IRC/FRC/IDC/DDC	LDAU	4	ldau	meu
ITC/DTC/IRC/FRC/IDC/DDC	IMDU	5	imdu	meu
ITC/DTC/IRC/FRC/IDC/DDC	DMDU	6	dmdu	meu
ITC/DTC/IRC/FRC/IDC/DDC	MAU	7	mau	meu
ITC/DTC/IRC/FRC/IDC/DDC	ITC	8	mec	mec
ITC/DTC/IRC/FRC/IDC/DDC	DTC	9	mec	mec
ITC/DTC/IRC/FRC/IDC/DDC	IRC	10	mec	mec
ITC/DTC/IRC/FRC/IDC/DDC	FRC	10	mec	mec
ITC/DTC/IRC/FRC/IDC/DDC	IDC	11	mec	mec
ITC/DTC/IRC/FRC/IDC/DDC	DDC	12	mec	mec

For the cases above where the "Bit set" column contains a value besides mec and meu, the syndrome, priv, and address for the highest-priority error in that cycle will overwrite the existing syndrome, priv, and address.

Once set, error status bits are only cleared by software. Hardware will never clear a set status bit. If a software write of the error register happens on the same cycle as an error, the setting of bits by the error will be based on the register state before the write. The setting of fields by the error will take precedence over the same field being update by the write; however, fields that are not updated by the error will be updated by the write

For example, if the register has the itc bit set and software does a write to clear that bit on the same cycle as a integer register file uncorrectable error, the error register would end up with the iru bit set, the itc bit cleared, the mec bit set, and the priv field would contain the value for the IRU error.

As another example, if the irc bit was set, and software does a write to clear that bit on the same cycle as an integer register file correctable error, the error register would end up with the irc bit clear, the mec bit set, and the priv field would contain the value for the first IRC error.

Programming Note	To minimize the possibility of missing notification of an error, software should clear any multiple error indication as soon as possible, since UltraSPARC T1 provides no indication of the number of multiple errors represented by the multiple error bit.
	An example of clearing behavior for the case where an IRC error is followed closely by an DMDU error would be to first log that an IRC error was seen (which is indicated by the irc bit still being set in the error status register), and then to do a write to the error status register with bit 17 set to clear the irc bit. The virtual address for the translation would then be read from the error address register to memory or a register, and software could do a write to the error status register with bits 29 and 23 set to clear the priv and dmdu bits and put the error status register back in a state where it can capture full error information. Software would then invoke the code to force out the DTLB entry with bad parity and assuming that code was successful, the process that encountered the error could be restarted.
	If another correctable error happened after the DMDU but before the write that cleared the irc bit, that error would not be logged. If another correctable error happened simultaneous to or after the write that cleared the irc bit, but before or simultaneous to the write that cleared the priv and dmdu bits, that error would be captured by the mec bit being set after the priv/dmdu-clearing write. If another uncorrectable error happened after the dmdu, but before or simultaneous to the write that cleared the priv and dmdu bits, that error would be captured by the meu bit being set after the priv/dmdu-clearing write.

12.4.3 ASI_SPARC_ERROR_ADDRESS_REG

Each strand has a hyperprivileged ASI_SPARC_ERROR_ADDRESS_REG register at ASI $4D_{16}$, VA{63:0} = 0. This register contains the information on the address source of the error and syndrome to be used by software to correct and/or log the error. The error register is not cleared on reset so software can examine its contents after an error-induced reset.

TABLE 12-7 defines the format of the ASI_SPARC_ERROR_ADDRESS_REG register.

TABLE 12-7 SPARC Error Address Register – ASI_SPARC_ERROR_ADDRESS_REG
(ASI $4D_{16}$, VA 0_{16})

Bit	Field	Initial Value	R/W	Description
63:48	—	0	R	*Reserved*
47:4	address	X	R	Error address/syndrome.
3:0	—	0	R	*Reserved*

TABLE 12-8 lists the bits captured for each of the error types.

TABLE 12-8 Bits Captured for Each Error Type

Error	Address Bits	Address Contents
IMTU	9:4	TLB entry index
DMTU	9:4	TLB entry index
NCU	39:4	Physical address
LDAU	39:4	Physical address
IRU	23:16	Syndrome
	11:9	GL for globals, register window number otherwise
	8:4	Register number
FRU[1]	30:24	Syndrome even half of register
	22:16	Syndrome odd half of register
	9:4	Register number
IMDU	47:4	Program counter presented for translation, TLB entry index in bits 9:4 for ASI load
DMDU	47:4	Virtual address presented for translation, TLB entry index in bits 9:4 for ASI load
DMSU	47:4	Virtual address presented for translation
MAU		Not valid
ITC	ll:5	I_cache tag index
DTC	10:4	D-cache tag index
IDC	39:4	Physical address
DDC	39:4	Physical address
IRC	23:16	Syndrome
	11:9	GL for globals, register window number otherwise
	8:4	Register number
FRC	30:24	Syndrome even half of register
	22:16	Syndrome odd half of register
	9:4	Register number

1. For errors on single-precision operands, the syndrome will be captured in bits 30:24 for even registers (bit 4 is 0) or bits 22:16 for odd registers (bit 4 is 1). The other syndrome bits (for example, bits 22:16 for an even register) will be all zeros.

For a given error, the unused bits in the address field are not guaranteed to be zero, and should be masked by software. The FRU/FRC register number contains the floating-point register number (0-31 for single precision operations; 0, 2, 4, … 62 for double-precision operations).

12.4.4   `ASI_DMMU_SFSR_REG`

Each strand has a hyperprivileged `ASI_DMMU_SFSR_REG` register at ASI 58_{16}, VA{63:0} = 18_{16}. This register contains the synchronous fault status. The only difference between the UltraSPARC T1 SFSR and the UltraSPARC II SFSR is that UltraSPARC T1 does not support the pr bit.

TABLE 12-9 defines the format of the `ASI_DMMU_SFSR_REG` register.

TABLE 12-9   1 DMMU Synchronous Fault Status Register – `ASI_DMMU_SFSR` (ASI 58_{16}, VA 18_{16})

Bit	Field	Initial Value	R/W	Description
63:61	—	0	R	*Reserved*
60:48	asi	X	RW	ASI of faulting instruction
47:42	—	0	R	*Reserved*
41:0	ft	X	RW	Fault type for *data_access_exception* trap
6	e	X	RW	Side-effect bit
5:4	ct	X	RW	Context
3	—	0	R	*Reserved*
2	w	X	RW	Write
1	ow	X	RW	Overwrite
0	fv	0	RW	Valid

12.5   L2 Cache Error Descriptions

The L2 Cache protects its tag with SEC ECC. Each 32 bit data subline is protected with SECDED ECC. In the following L2 cache behavior descriptions, a partial store refers to a store of less than 32 bits.

12.5.1 L2 Cache Data Correctable ECC Error for Access (LDAC)

12.5.1.1 Load Hit/Instruction Fetch Hit

When a correctable ECC error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the requesting strand. Hardware corrects the error in the data being returned from the L2 cache, but does not correct the L2 cache data itself.

12.5.1.2 Prefetch Hit

When a correctable ECC error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers and no trap is generated. Hardware corrects the error in the data being returned from the L2 cache, but does not correct the L2 cache data itself.

12.5.1.3 Partial Store/Atomic Hit

When a correctable ECC error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the requesting strand. Hardware corrects the error in the L2 cache and returns the corrected data.

12.5.1.4 Modular Arithmetic Load Hit

When a correctable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware corrects the error on the data being returned from the L2 cache, but does not correct the L2 cache data itself. In addition, if the L2 Cache Error Enable ceen bit is set, the corrected error information is sent to the modular arithmetic unit.

Once the modular arithmetic unit completes, if it was sent an indication of a corrected error, a disrupting *ECC_error* trap is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt (if the ASI_MA_CONTROL_REG int bit is set this interrupt is in addition to the *modular_arithmetic_interrupt* completion trap). If the ASI_MA_CONTROL_REG int bit is cleared, a disrupting *ECC_error* trap is generated to the strand that issues the ASI_MA_SYNC. The *ECC_error* trap is generated only if the SPARC Error Enable ceen bit is set.

12.5.2 L2 Cache Data Correctable ECC Error for Writeback (LDWC)

When a correctable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware corrects the error on the data being written to memory. In addition, if the L2 Cache Error Enable ceen bit is set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer. The *ECC_error* trap is generated only if the SPARC Error Enable ceen bit is set to 1 for the strand specified in L2_CSR_REG.errorsteer.

12.5.3 L2 Cache Data Correctable ECC Error for DMA (LDRC)

12.5.3.1 DMA Read

When a correctable ECC error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware corrects the error on the data being returned from the L2 cache, but does not correct the L2 cache data itself. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set to 1, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

> **Note** | All DMA read operations will read full 64-byte lines and treat each 4-byte error as a separate error.

12.5.3.2 DMA Write Partial

When a correctable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware corrects the error in the L2 cache line. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set to 1, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

L2 Data ECC is only checked for partial DMA stores. Aligned 4B and 8B (and larger) DMA writes overwrite previous contents, so never check data ECC.

12.5.4 L2 Cache Data Correctable ECC Error for Scrub (LDSC)

When a correctable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware corrects the error in the L2 cache line by writing back the corrected data and parity (this rewrite

will be unable to correct a permanently failed bit). In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

12.5.5 L2 Cache Data Uncorrectable ECC Error for Access (LDAU)

12.5.5.1 Load Hit/Instruction Fetch Hit

When an uncorrectable ECC error is detected during a load operation or instruction fetch, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set to 1, the error information is also captured in the SPARC Error Status and SPARC Error Address Registers. If the L2 Error Enable nceen bit is set to 1, the erroneous data is loaded in the L1 cache with bad parity. In addition, if the L2 Error Enable nceen and SPARC Error Enable nceen bits are set to 1, a precise *data_access_error* trap is generated to the requesting strand for the load hit or a precise *instruction_access_error* trap is generated to the requesting strand for the instruction fetch hit.

> **Note** | If data loaded into the L1 cache with bad parity is accessed by any strand before the cache can be cleaned, the IDC/DDC error will be seen first, and then the hardware will try to refetch the line from the L2 as described in *I-cache Data Parity Error (IDC)* on page 117 and *D-cache Data Parity Error (DDC)* on page 118, causing another LDAU error.

12.5.5.2 Prefetch Hit

When an uncorrectable ECC error is detected during a prefetch operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers and no trap is generated.

12.5.5.3 Partial Store Hit

When an uncorrectable ECC error is detected during a partial store (STPARTIALF) operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, if the L2 Cache Error Enable nceen and

SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the requesting strand. The partial store will not complete its update, leaving the original data and the bad ECC unchanged.

Notes	(1) While the partial store does not update the data, the cache line is marked as Dirty (if not already marked as such) so that if the line is replaced it will be written back to memory to mark the memory data with bad ECC.
	(2) It is possible, but rare, that a partial store hitting an uncorrectable error will log both LDAU and meu. For this to occur, an (unrelated) fill operation must enter the L2 pipeline between the initial read and the subsequent write for the partial store operation.

12.5.5.4   Atomic Load-Store Hit

When an uncorrectable ECC error is detected during an atomic load-store operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set, the error information is also captured in the SPARC Error Status and SPARC Error Address Registers. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a precise *data_access_error* trap is generated to the requesting strand. The atomic operation will not complete its update, leaving the original data and the bad ECC unchanged.

Note	While the atomic load-store operation does not update the data, the cache line is marked as Dirty (if not already marked as such). Therefore, if the line is replaced, it will be written back to memory to mark the memory data with bad ECC.

12.5.5.5   Modular Arithmetic Load

When an uncorrectable ECC error is detected during a modular arithmetic operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, if the L2 Error Enable nceen bit is set, the uncorrected error information is sent to the modular arithmetic unit, which immediately aborts its asynchronous operation and logs the error in the SPARC Error Status and SPARC Error Address registers.

Once the abort completes, if the SPARC Error Enable nceen bit is set and the ASI_MA_CONTROL_REG int bit is set, a disrupting *data_error* is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt and the normal *modular_arithmetic_interrupt* completion trap is not generated. If the SPARC Error Enable nceen bit is set and the ASI_MA_CONTROL_REG int bit is cleared, a precise *data_access_error* trap is generated to the strand that issues any load from an ASI_MA_* register.

If the SPARC Error Enable nceen bit is cleared and the ASI_MA_CONTROL_REG int bit is set, a normal *modular_arithmetic_interrupt* is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt.

If the SPARC Error Enable nceen bit is cleared and the ASI_MA_CONTROL_REG int bit is cleared, the load access to ASI_MA_SYNC_REG will complete normally without generating an error trap.

12.5.6 L2 Cache Data Uncorrectable ECC Error for Writeback (LDWU)

When an uncorrectable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware indicates the error on the data being written to memory, where the DRAM controller will write back the data with poisoned ECC. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

12.5.7 L2 Cache Data Uncorrectable ECC Error for DMA (LDRU)

12.5.7.1 DMA Read

When an uncorrectable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Hardware returns the data with an error indicator back to the dma requestor. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

> **Note** | All DMA read operations will read full 64-byte lines and treat each 4-byte error as a separate error.

12.5.7.2 DMA Write Partial

When an uncorrectable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer. The DMA write partial will not complete its update, leaving the original data and the bad ECC unchanged.

> **Note** | While the DMA write partial does not update the data, the cache line is marked as Dirty (if not already marked as such) so that if the line is replaced it will be written back to memory to mark the memory data with bad ECC.

12.5.8 L2 Cache Data Uncorrectable ECC Error for Scrub (LDSU)

When an uncorrectable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

12.5.9 L2 Cache Tag Correctable ECC Error (LTC)

On every L2 access, ECC is checked for all 12 tags in the set. When a correctable ECC error is detected the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. Note that the Syndrome is *not* captured for a L2 cache tag ECC error. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

Hardware will clear the error, if and only if there was not a good tag that had a hit. If there is an error with a hit, the error will be logged and a trap will be generated, but software needs to clear the error to prevent additional traps.

> **Implementation Note** | Hardware does not generate the error trap until the tag error has been corrected. If the error is due to a hard failure in the tag, hardware will not be able to complete the correction and no further access will be able to be processed by the L2 (that is, the L2 bank is hung).

12.5.10 L2 Cache VAD Uncorrectable Parity Error (LVU)

On every L2 access, parity is checked for all 12 vad bits in the set (the "used" bit of vuad is not covered by parity since it only affects performance, not correctness). When an uncorrectable parity error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, a fatal error indication is issued across JBUS, to request a warm reset (WMR) trap to the entire chip.

12.5.11 L2 Cache Directory Uncorrectable Directory Parity (LRU)

During directory scrub, parity is checked for the directory entry. When an uncorrectable parity error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, a fatal error indication is issued across JBUS, to request a warm reset (WMR) trap to the entire chip.

12.6 L2 Error Registers

RegisterBaseAddress L2CSR Registers – A0 0000 0000$_{16}$.

12.6.1 L2 Error Enable Register

Each L2 bank has an error enable register that controls the reporting of L2 errors for that bank back to the initiator of an operation (or to the strand specified in L2_CSR_REG.errorsteer if no initiator exists or can be readily identified). The L2 Error Enable register, the format of which is shown in TABLE 12-11, is available at address offsets AA 0000 0000$_{16}$ or BA 0000 0000$_{16}$. Address bits 7:6 select the cache bank and address bits 31:8 and 5:3 are ignored (that is, the register aliases across the address range).

TABLE 12-10 Error Enable Register – L2_ERROR_EN_REG (A00000000$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:3	—	X	R	*Reserved*
2	debug_trig_en	0	RW	Trigger enable for the debug port.
1	nceen	0	RW	If set to 1, report uncorrectable errors.
0	ceen	0	RW	If set to 1, report correctable errors.

> **Note** | Errors are always logged in the L2_ERROR_STATUS_REG and L2_ERROR_ADDRESS_REG regardless of the setting of the nceen and ceen bits in L2_ERROR_EN_REG. L2_ERROR_EN_REG only controls whether or not the error is reported back to the appropriate strand (either the requestor, the strand specified in L2_CSR_REG.errorsteer, or the strand specified in ASI_MA_CONTROL_REG).

> **Programming Note** | This register can be used to get a debug trigger on a memory UE, by setting L2_ERROR_EN[DEBUG_TRIG_EN], since DRAM UEs tend to quickly cause L2 errors. This is the preferred method since the DRAM block does not assert debug triggers for UEs, only CEs.

12.6.2 L2 Error Status Register

Each L2 bank has an error status register which contains status on L2 errors for that bank. The status bits in this register are cleared by writing a 1 to the bit position. The error register is not cleared on reset so software can examine its contents after an error-induced reset. The L2 Error Status Register, the format of which is shown in TABLE 12-12, is available at address offsets AB 0000 0000$_{16}$ or BB 0000 0000$_{16}$.

> **Note** | Since this register is not cleared on reset, after a power-on reset the contents of this register are undefined, and the bits could be in an illegal state that could not possibly be generated by any error combination (for example, meu bit set with all other bits cleared). Operation while in this illegal state leads to undefined behavior for the register, so software should always clear this register after a power-on reset.

Address bits 7:6 select the cache bank, address bits 31:8 and 5:3 are ignored (that is, the register aliases across the address range).

TABLE 12-11   L2 Error Status Register – L2_ERROR_STATUS_REG (0B 0000 0000$_{16}$)

Bit	Field	Initial Value	R/W	Description
63	meu	Preserved	RW1C	Multiple uncorrected errors, one or more uncorrected errors were not logged.
62	mec	Preserved	RW1C	Multiple corrected errors, one or more corrected errors were not logged.
61	rw	Preserved	RW	Specifies whether the error access was a read or write. Set to 1 for a write, 0 for a read.
60	—	0	R	*Reserved*
59	moda	Preserved	RW	Set to 1 if the error was the result of an modular arithmetic read operation, 0 otherwise.
58:54	vcid	Preserved	RW	ID of virtual processor that encountered error.
53	ldac	Preserved	RW1C	Set to 1 if the error was a L2 cache data array access correctable error.
52	ldau	Preserved	RW1C	Set to 1 if the error was a L2 cache data array access uncorrectable error.
51	ldwc	Preserved	RW1C	Set to 1 if the error was a L2 cache data array writeback correctable error.

Bit	Field	Initial Value	R/W	Description
50	ldwu	Preserved	RW1C	Set to 1 if the error was a L2 cache data array writeback uncorrectable error.
49	ldrc	Preserved	RW1C	Set to 1 if the error was a L2 cache data array dma access correctable error.
48	ldru	Preserved	RW1C	Set to 1 if the error was a L2 cache data array dma access uncorrectable error.
47	ldsc	Preserved	RW1C	Set to 1 if the error was a L2 cache data array scrub correctable error.
46	ldsu	Preserved	RW1C	Set to 1 if the error was a L2 cache data array scrub uncorrectable error.
45	ltc	Preserved	RW1C	Set to 1 if the error was a L2 cache tag array correctable error.
44	lru	Preserved	RW1C	Set to 1 if the error was a L2 cache directory uncorrectable error.
43	lvu	Preserved	RW1C	Set to 1 if the error was a L2 cache VUAD array uncorrectable error.
42	dac	Preserved	RW1C	Set to 1 if the error was a DRAM access correctable error.
41	dau	Preserved	RW1C	Set to 1 if the error was a DRAM access uncorrectable error or JBUS uncorrectable error.
40	drc	Preserved	RW1C	Set to 1 if the error was a DRAM dma access correctable error.
39	dru	Preserved	RW1C	Set to 1 if the error was a DRAM dma access uncorrectable error.
38	dsc	Preserved	RW1C	Set to 1 if the error was a DRAM scrub correctable error. Setting this bit does not affect MEC nor VEC.
37	dsu	Preserved	RW1C	Set to 1 if the error was a DRAM scrub uncorrectable error. Setting this bit does not affect MEU nor VEU.
36	vec	Preserved	RW1C	Set to 1 if the register contains a valid correctable error.
35	veu	Preserved	RW1C	Set to 1 if the register contains a valid uncorrectable error.
34:32	—	0	R	*Reserved*
31:0	synd	Preserved	RW	Parity or ECC syndrome.

The vec bit is set on all cycles where a correctable error (except DSC) is encountered. The veu bit is set on all cycles where an uncorrectable error (except DSU) is encountered.

The synd field is only valid for LDAC, LDAU, LDSU, LDSC, LDRU, LDRC, and LVU errors.

The rw bit will be set to 1 for a partial store, DMA write, and atomic load/store operation if they detect an error while performing the read part of the L2 read-modify-write operation. For DAC and DAU errors, if an L2 fill happens before the data is returned to the requestor, the DAC or DAU error will be reported to the strand specified in L2_CSR_REG.errorsteer, and the rw bit will be 0.

The vcid field is valid only for LDAC, LDAU, DAC, and DAU errors where the error is detected synchronously with the load or store operation. If the error is reported at the time of the L2 cache fill operation, vcid will contain 0. For all other error types,

this field is not valid. Note that the dau bit is used for both DRAM and JBUS uncorrectable errors. A JBUS error can be distinguished from a DRAM error by examining the state of the dau bit in the DRAM_ERROR_STATUS register.

TABLE 12-13 summarizes the rw, vcid, moda, and synd fields.

TABLE 12-12 rw, vcid, moda, and synd Fields Summary

Error	rw	vcid	moda	synd
During L2 fill for DAC,DAU, DRC, DRU	X	L2_CSR_REG. errorsteer	X	X
LDAC, LDAU	1 for atomics and partial stores, else 0	vcid	1 for modular arithmetic load, else 0	0000, synd_127_96{6:0}, synd_95_64{6:0}, synd_63_32{6:0}, synd_31_0{6:0}
LDWC, LDWU	X	X	X	X
LDRC, LDRU	1 for write, else 0	X	X	0000, synd_127_96{6:0}, synd_95_64{6:0}, synd_63_32{6:0}, synd_31_0{6:0} for write, X for read
LDSC, LDSU	X	X	X	0000, synd_127_96{6:0}, synd_95_64{6:0}, synd_63_32{6:0}, synd_31_0{6:0}
LTC	X	X	X	X
LRU	X	X	X	X
LVU	X	X	X	zeros{31:4}, par_valid, par_dirty, par_used, par_alloc. Note that parity is not checked on *used*.
DAC, DAU, DBU	1 for atomics and partial stores, else 0	vcid	1 for modular arithmetic load, else 0	X
DRC, DRU	1 for write, else 0	X	X	X
DSC, DSU	N.A.	N.A.	N.A.	N.A.

The dsc and dsu bits are logged in L2 Error Status register to notify software to check the DRAM error registers, and are set regardless of the status of the other bits. Setting dsc and/or dsu does not cause any logging of the error address or syndrome in the L2 Error Address register.

Note | The syndrome is also *not* logged on a L2 tag correctable error.

With regard to the remaining bits, if multiple errors occur in the same cycle, the meu and/or mec bit is set and only the highest-priority error is logged based on the following priority shown in TABLE 12-14 (errors with the same priority are mutually exclusive).

TABLE 12-13 Priority for Simultaneous Errors

Error	Priority	Bit Set if Higher-Priority Error
LVU	1	
LRU	2	meu
LDAU	3	meu
LDSU	3	meu
LDWU	4	meu
LDRU	5	meu
DAU	6	meu
DRU	6	meu
LTC	7	mec
LDAC	8	mec
LDSC	8	mec
LDWC	9	mec
LDRC	10	mec
DAC	11	mec
DRC	11	mec
DSC	NA	dsc
Dsu	NA	dsu

The syndrome, rw, moda, vcid, and address are captured for the highest-priority error in that cycle.

If errors occur in a cycle when an error status bit is already set (indicating a previous error exists that hasn't been cleared from the error status register), the information in TABLE 12-15 applies.

TABLE 12-14 Errors Occurring in a Cycle Where Error Status Bit Already Set

Existing Error	Error	Priority	Bit Set if Highest-Priority Error	Bit Set if Higher-Priority Error in Same Cycle
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LVU	1	meu	N/A
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LRU	2	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDAU	3	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDSU	3	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDWU	4	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDRU	5	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	DAU	6	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	DRU	6	meu	meu
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LTC	7	mec	mec

Existing Error	Error	Priority	Bit Set if Highest-Priority Error	Bit Set if Higher-Priority Error in Same Cycle
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDAC	8	mec	mec
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDSC	8	mec	mec
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDWC	9	mec	mec
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	LDRC	10	mec	mec
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	DRC	11	mec	mec
LVU/LRU/LDAU/LDSU/LDWU/LDRU/DRU/DAU	DAC	12	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LVU	1	lvu	n/a
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LRU	2	lru	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDAU	3	ldau	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDSU	3	ldsu	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDWU	4	ldwu	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDRU	5	ldru	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	DAU	6	dau	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	DRU	6	dru	meu
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LTC	7	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDAC	8	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDSC	8	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDWC	9	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	LDRC	10	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	DRC	11	mec	mec
LTC/LDAC/LDSC/LDWC/LDRC/DRC/DAC	DAC	12	mec	mec
Any	DSC	N.A.	dsc	dsc
Any	Dsu	N.A.	dsu	dsu

For the cases above where the "Bit set" column contains a value besides mec and meu, the syndrome, rw, moda, vcid, and address for the highest-priority error in that cycle will overwrite the existing syndrome, rw, moda, vcid, and address.

Once set, error status bits are only cleared by software. Hardware will never clear a set status bit. If a software write of the error register happens on the same cycle as an error, the setting of bits by the error will be based on the register state before the write, following the rules of TABLE 12-15. The setting of fields by the error will take precedence over the same field being update by the write; however, fields that are not changed by the error will be updated by the write.

For example, if the register has the vec and ltc bits set and software does a write to clear those bit on the same cycle as a L2 cache data uncorrectable error, the error register would end up with the veu and ldau bits set, the vec and ltc bits cleared, and the rw, moda, vcid, and synd fields would contain the values for the LDAU error.

The rw, moda, vcid, and synd fields are always considered to be set by an error, even if the value they are being set to is undefined (that is, set to X in TABLE 12-13).

Notes (1) When writing the error status register on the same cycle that another error occurs, the error status register state before the register write is applied determines which bits will be updated in the register. If the error occurring on the same cycle as the write is lower priority than the error currently logged in the register that is being cleared by the write, this will result in the error status register having only either the vec and mec pair of bits set (for a correctable error) or veu and meu pair of bits set (for an uncorrectable error) after both the new error and the write which clears the old error bits are applied.

(2) The error priority implementation for preexisting errors is actually done by checking the values of the vec and veu bits, rather than the "sum of the bits that set vec or veu," as implied by the preceding table. This means that if sofware clears vec/veu, a new error will be logged, regardless of the state of the other error status bits. In other words, it is a good idea to clear the vec/veu bit(s) at the same time as the error status bit(s) that caused vec/veu to be set.

To minimize the possibility of missing notification of an error, software should clear any multiple error indication as soon as possible, since UltraSPARC T1 provides no indication of the number of multiple errors represented by the multiple error bit.

An example of clearing behavior for the case where an DAC error is followed closely by an DAU error would be to first log that an DAC error was seen (which is indicated by the dac bit still being set in the error status register), and then to do a write to the error status register with bits 42 and 36 set to clear the dac and vec bits. The vcid, moda, rw, and synd fields of error status would then be captured to memory or a register, the physical address for the DAU would be read from the error address register to memory or a register, and software could do a write to the error status register with bits 41 and 35 set to clear the dau and veu bits and put the error status register back in a state where it can capture full error information. Software would then invoke the code to shoot down all the TLB entries for the page with the bad cache line, force the bad cache line from L2 to memory, and kill all processes that had access to the bad cache line.

If another correctable error happened after the DAU but before the write that cleared the dac and vec bits, that error would not be logged. If another correctable error happened simultaneous to or after the write that cleared the dac and vec bits, but before or simultaneous to the write that cleared the dau and veu bits, that error would be captured by the mec and vec bits being set after the dau/veu-clearing write. If another uncorrectable error happened after the DAU, but before or simultaneous to the write that cleared the dau and veu bits, that error would be captured by the meu and veu bits being set after the dau/veu-clearing write.

12.6.3 L2 Error Address Register

Each L2 bank has an error address register that contains the address for the L2 error within that bank. The error register is not cleared on reset, so software can examine its contents after an error-induced reset. The L2 Error Address register is available at address offsets AC 0000 0000$_{16}$ or BC 0000 0000$_{16}$. Address bits 7:6 select the cache bank, address bits 31:8 and 5:3 are ignored (that is, the register aliases across the address range).

TABLE 12-16 shows the format of the L2 Error Address Register.

TABLE 12-15 L2 Error Address Register – L2_ERROR_ADDRESS_REG (C 0000 0000$_{16}$)

Bit	Field	Initial Value	R/W	Description
39:4	address	Preserved	R/W	Error address.
3:0	—	0	R	*Reserved*

TABLE 12-17 lists the bits captured for each of the error types.

TABLE 12-16 Bits Captured for Each Error Type

Error	Address Bits	AddresS Contents
LDAC	39:6	Physical address of cache line
LDAU	39:6	Physical address of cache line
LDWC	39:6	Physical address of cache line
LDWU	39:6	Physical address of cache line
LDRC (RW = 0)	39:6	Physical address of cache line
LDRC (RW = 1)	39:4	Physical address of quadword accessed
LDRU (RW = 0)	39:6	Physical address of cache line
LDRU (RW = 1)	39:4	Physical address of quadword accessed
LDSC	19:6	Cache index (19:16 way, 15:6 set). Note that this is shifted two bits from "expected," that set{15:6} corresponds to PA{17:8}.
LDSU	19:6	Cache index (19:16 way, 15:6 set). Note that this is shifted two bits from "expected," that set{15:6} corresponds to PA{17:8}.
LTC	17:6	Cache index (17:8 set, 7:6 bank)
LRU	16:6	Directory index (16:12 panel, 11:9 coreID, 8:7 L1way, 6 IcacheDir). If IcacheDir = 1, panel = address{10,9,8,5,11}, else panel = address{10,9,8,5,4}.
LVU	39:6	Physical address of cache line
DAC	39:6	Physical address of cache line
DAU	39:6	Physical address of cache line
DRC	39:6	Physical address of cache line
DRU	39:6	Physical address of cache line

For a given error, the unused bits in the address field are not guaranteed to be zero, and should be masked by software.

Programming Note It is not guaranteed that the L2 and corresponding DRAM controller will log errors in the same order. This means that on memory errors (DAC, DRC, DAU, DRU), if multiple errors are in close temporal proximity, the address in the LEAR and the syndrome in the DESR could be for two different errors. In these cases, there is an MEC or MEU logged in DRAM, since DRAM saw both errors.

12.7 L2 Software Error Scrubbing Support

Some errors will leave the L2 cache with a correctable error, which then needs to be scrubbed to prevent repetitive traps for effectively the same soft error. Flushing the data out of the cache, back to memory, will cause the error to be corrected.

With a 12-way set associative cache, with pseudo-LRU replacement, and no explicit flush instruction, flushing the error line is not entirely trivial. However, to make the flush provably workable, the L2 has a "Direct Mapped Replacement Mode" which forces the replacement algorithm to simulate a direct-mapped cache. This direct-mapped mode is safe to spuriously enable in a running system, since lines inserted in either mode can still be found (tag matched) normally.

In order to flush a line, software (hypervisor) would enable Direct-Mapped mode, fault in 12 cache lines with the same index as the error line (but is not the error line), then restore (disable?) the original state of the Direct-Mapped mode.

Scrubbing correctable main memory errors uses the same support. To scrub, fault the line into the L2, dirty it without modifying it (CAS), then use Direct-Mapped mode and 12 other fault-ins to force the error line out.

12.8 DRAM Error Descriptions

Each 128 bit data block in memory is protected by QEC/OEC (Quad Error Detect, Octal Error Detect) ECC. This ECC code supports ChipKill for x4 DRAM chips, where the complete failure of any aligned 4-bit block can be corrected, and any error where exactly two 4-bit blocks are in error is recognized as an uncorrectable error.

12.8.1 DRAM Correctable ECC Error for Access (DAC)

> **Note** If L2_CONTROL_REG.dis = 1 (the L2 cache is disabled), no logging or trap generation is performed for DRAM correctable ECC errors encountered on 32-bit and 64-bit stores.

12.8.1.1 Prefetch Miss

When a correctable ECC error is indicated in the DRAM reply during a prefetch operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. In addition, if the L2 Cache Error

Enable ceen and SPARC Error Enable ceen bits are set and the error was not in the block requested (what would be the 16B primary line), a disrupting *ECC_error* trap is delivered to the strand specified in L2_CSR_REG.errorsteer. Otherwise, no trap is generated to SPARC, but the error is still logged in L2 if L2 Cache Error Enable ceen is set. Hardware corrects the error before the data is placed into the L2 cache.

12.8.1.2 Load Miss/Instruction Fetch Miss/Atomic Miss

When a correctable ECC error is indicated in the DRAM reply during a load, atomic load-store, or instruction fetch operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated. If the error was in the block requested (the 16-byte primary line for loads, 32-byte primary line for instruction fetches), and the cache is not filled before the data is returned to the requestor, the trap is delivered to the requesting strand. Otherwise the trap is delivered to the strand specified in L2_CSR_REG.errorsteer. Hardware corrects the error before the data is placed into the L1 or L2 cache.

> **Note** | For a line from memory with correctable errors in both the block requested and outside the block requested, if the cache is not filled before the data is returned to the requestor, a *ECC_error* trap will be generated to both the requesting virtual processor, and to the virtual processor specified in L2_CSR_REG.errorsteer (with one exception, if the requesting virtual processor is the same as the one specified in L2_CSR_REG.errorsteer, it is possible for either one or two *ECC_error* traps to be generated to the requesting virtual processor depending on hardware timing).

12.8.1.3 Partial Store Miss

When a correctable ECC error is indicated in the DRAM reply during a partial store (STPARTIALF) operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer. Hardware corrects the error before the data is placed into the L2 cache (the new store data will also be correct).

12.8.1.4 Store Miss/Modular Arithmetic Store Miss

When a correctable ECC error is indicated in the DRAM reply during a store or modular arithmetic store operation, the error information is captured in the DRAM Error Status register. If the L2 Control Register dis bit is cleared, the error

information is also captured in the L2 Cache Error Status, and L2 Cache Error Address registers. In addition, if the L2 Control Register dis bit is cleared and the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer. Hardware corrects the error before the data is placed into the L2 cache (the new store data will also be correct).

12.8.1.5   Modular Arithmetic Load

When a correctable ECC error is indicated in the DRAM reply during a modular arithmetic load operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. Hardware corrects the error before the data is placed into the L2 cache and returned to the modular arithmetic unit. In addition, if the L2 Cache Error Enable ceen bit is set, the corrected error information is sent to the modular arithmetic unit. Once the modular arithmetic unit completes, if it was sent an indication of a corrected error, a disrupting *ECC_error* trap is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt if the ASI_MA_CONTROL_REG int bit is set (this interrupt is in addition to the *modular_arithmetic_interrupt* completion interrupt). If the ASI_MA_CONTROL_REG int bit is cleared, a disrupting *ECC_error* trap is generated to the strand that issues the ASI_MA_SYNC. The *ECC_error* trap is generated only if the SPARC Error Enable ceen bit is set.

12.8.1.6   DMA Read (DRC/DAC)

When a correctable ECC error is indicated in the DRAM reply during a DMA read operation, the information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. Hardware corrects the error and returns it to the DMA requestor. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

L2 Error status will report DRC (DMA correctable) for this error, while memory will report DAC.

12.8.1.7   DMA Write Partial (DRC/DAC)

When a correctable ECC error is indicated in the DRAM reply during a DMA write-partial operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. Hardware corrects the error in the L2 cache line. In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

L2 Error status will report DRC (DMA correctable) for this error, while memory will report DAC.

12.8.2 DRAM Correctable ECC Error for Scrub (DSC)

When a correctable ECC error is found during a scrub, the information is captured in the DRAM Error Status and DRAM Error Address registers, and the dsc bit is set in the L2 Cache Error Status register. Hardware corrects the error in memory by writing back the corrected data and parity (this rewrite will be unable to correct a permanently failed bit). In addition, if the L2 Cache Error Enable ceen and SPARC Error Enable ceen bits are set, a disrupting *ECC_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

12.8.3 DRAM Uncorrectable ECC Error for Access (DAU)

12.8.3.1 Load Miss/Instruction Fetch Miss/Atomic Miss

When an uncorrectable ECC error is indicated in the DRAM reply during an instruction fetch, load, or atomic load-store operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set, the error information is also captured in the SPARC Error Status and SPARC Error Address registers. For each 32-bit chunk with an error, the data is loaded into the L2 cache with poisoned ECC. For an atomic, if the error was in the word to be updated, the atomic operation will not complete its update, leaving the original data and the bad ECC unchanged.

In addition, if the L2 Error Enable nceen and SPARC Error Enable nceen bits are set, a trap is generated. If the error was not in the block requested (the 16-byte primary line for loads, 32-byte primary line for instruction fetches), a disrupting *data_error* is generated to the strand specified in L2_CSR_REG.errorsteer. Otherwise, one of two conditions will occur, depending on whether the cache is refilled before or after the miss is reissued to the cache (either option is possible during normal operation). If the miss is reissued to the cache before the fill, a precise *data_access_error* trap is generated for the load or atomic hit or a precise *instruction_access_error* trap is generated for the instruction fetch hit, and the data is loaded into the L1 cache with bad parity. Otherwise, if the fill is issued first, a disrupting *data_error* is generated to

the strand specified in L2_CSR_REG.errorsteer. The reissued miss will then encounter a separate LDAU error (see Section 12.5.5 for details), and as a result the L2 Cache Error Status will end up with the meu bit set.

> **Note** | While an atomic load-store operation that encounters an uncorrectable ECC error does not update the data, the cache line is marked as Dirty so that if the line is replaced it will be written back to memory to mark the memory data with bad ECC.

12.8.3.2 Prefetch Miss

When an uncorrectable ECC error is indicated in the DRAM reply during a prefetch operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. For each 32-bit chunk with an error, the data is loaded into the L2 cache with poisoned ECC. In addition, if the error was not in the block requested (what would be the 16-byte primary line), and the L2 Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer. Otherwise, no trap is generated, but if L2 Error Enable nceen is set, the error is still logged in the L2.

12.8.3.3 Partial Store Miss/Store Miss/Modular Arithmetic Store Miss

When an uncorrectable ECC error is indicated in the DRAM reply during a partial store (STPARTIALF), store, or modular arithmetic store operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. For each 32-bit chunk with an error, the data is loaded into the L2 cache with poisoned ECC. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

For a partial store, if there is an error in the block (16B) to be written, the partial store will not complete its update, leaving the original data and the poisoned ECC unchanged.

> **Notes** | (1) While the partial store does not update the data, the cache line is marked as Dirty so that if the line is replaced it will be written back to memory to mark the memory data with bad ECC.
>
> (2) If L2 caches are disabled, DAU errors on partial stores are logged correctly, but do not generate a trap. From a practical standpoint, L2 disabled mode is a bringup-only mode, intended until the caches are known to be working, and not appropriate for normal use.

12.8.3.4 Modular Arithmetic Load

When an uncorrectable ECC error is indicated in the DRAM reply during a modular arithmetic load operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. For each 32-bit chunk with an error, the data is loaded into the L2 cache with poisoned ECC. In addition, if the L2 Error Enable nceen bit is set, the uncorrected error information is sent to the modular arithmetic unit, which immediately aborts its asynchronous operation and logs the error in the SPARC Error Status and SPARC Error Address registers.

Once the abort completes, if the SPARC Error Enable nceen bit is set and the `ASI_MA_CONTROL_REG` int bit is set, a disrupting *data_error* is generated to the strand specified in the `ASI_MA_CONTROL_REG` as receiving the completion interrupt and the normal *modular_arithmetic_interrupt* completion trap is not generated.

If the SPARC Error Enable nceen bit is set and the `ASI_MA_CONTROL_REG` int bit is cleared, a precise *data_access_error* trap is generated to the strand that issues any load from an `ASI_MA_*` register.

If the SPARC Error Enable nceen bit is cleared and the `ASI_MA_CONTROL_REG` int bit is set, a normal *modular_arithmetic_interrupt* is generated to the strand specified in the `ASI_MA_CONTROL_REG` as receiving the completion interrupt.

If the SPARC Error Enable nceen bit is cleared and the `ASI_MA_CONTROL_REG` int bit is cleared, the load access to `ASI_MA_SYNC_REG` will complete normally without generating an error trap.

12.8.3.5 DMA Read (DRU/DAU)

When an uncorrectable ECC error is indicated in the DRAM reply during a DMA read operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. Hardware returns the data back to the DMA requestor, with an error indicator for each 128-bit chunk with an error. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

L2 Error status will report DRU (DMA uncorrectable) for this error, while memory will report DAU.

12.8.3.6 DMA Write Partial (DRU/DAU)

When an uncorrectable ECC error is indicated in the DRAM reply during a DMA write-partial operation, the error information is captured in the DRAM Error Status, L2 Cache Error Status, and L2 Cache Error Address registers. For each 32-bit chunk with an error, the data is loaded into the L2 cache with poisoned ECC. In addition, if

the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

If there is an error in the block (16B) to be written, the DMA write partial will not complete its update, leaving the original data and the poisoned ECC unchanged.

> **Note** | While the DMA write-partial operation that causes detection of an uncorrectable ECC error does not update the data, the cache line is marked as Dirty so that if the line is replaced it will be written back to memory to mark the memory data with bad ECC.

L2 Error status will report DRU (DMA uncorrectable) for this error, while memory will report DAU.

12.8.4 DRAM Uncorrectable ECC Error for Scrub (DSU)

When an uncorrectable ECC error is found during a scrub, the information is captured in the DRAM Error Status and DRAM Error Address registers, and the DSU bit is set in the L2 Cache Error Status register. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

12.8.5 DRAM Addressing Error (DBU)

12.8.5.1 Load Miss/Instruction Fetch Miss/Atomic Miss (DAU/DBU)

When an out-of-bounds error is indicated in the DRAM reply during an instruction fetch, load, or atomic load-store operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set, the error information is also captured in the SPARC Error Status and SPARC Error Address Registers. An out-of-bounds error is signaled as a cache line with all data marked with an uncorrectable error.

For each 32-bit chunk, the data is loaded into the L2 cache with poisoned ECC. For an atomic, if the error was in the word to be updated, the atomic operation will not complete its update, leaving the original data and the bad ECC unchanged. In addition, if the L2 Error Enable nceen and SPARC Error Enable nceen bits are set, a trap is generated. For this error, one of two conditions will occur, depending on whether the cache is refilled before or after the miss is reissued to the cache (either option is possible during normal operation). If the miss is reissued to the cache before the fill, a precise *data_access_error* trap is generated for the load or atomic hit or a precise *instruction_access_error* trap is generated for the instruction fetch

hit, and the data is loaded into the L1 cache with bad parity. Otherwise, if the fill is issued first, a disrupting *data_error* is generated to the strand specified in L2_CSR_REG.errorsteer. The reissued miss will then encounter a separate LDAU error (see Section 12.5.5 for details), and as a result the L2 Cache Error Status will end up with the meu bit set.

> **Note** | While an atomic load-store operation that causes an out-of-bounds error does not update the data, the cache line is marked as Dirty.

L2 Error status will report DAU (Uncorrectable error on Access) for this error, while memory will report DBU.

12.8.5.2 Prefetch Miss (DAU/DBU)

When an out-of-bounds error is indicated in the DRAM reply during a prefetch operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. An out-of-bounds error is signaled as a cache line with all data marked with an uncorrectable error. For each 32-bit chunk, the data is loaded into the L2 cache with poisoned ECC. In addition, if the L2 Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

L2 Error status will report DAU (Uncorrectable error on Access) for this error, while memory will report DBU.

12.8.5.3 Partial Store Miss/Store Miss/Modular Arithmetic Store Miss (DAU/DBU)

When an out-of-bounds error is indicated in the DRAM reply during a partial store (STPARTIALF), store, or modular arithmetic store operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. An out-of-bounds error is signaled as a cache line with all data marked with an uncorrectable error. For each 32-bit chunk, the data is loaded into the L2 cache with poisoned ECC. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer. For a partial store, the partial store will not complete its update, leaving the original data and the bad ECC unchanged.

> **Note** | While a partial store that causes an out-of-bounds error does not update the data, the cache line is marked as Dirty.

L2 Error status will report DAU (Uncorrectable error on Access) for this error, while memory will report DBU.

12.8.5.4　Modular Arithmetic Load (DAU/DBU)

When an out-of-bounds error is indicated in the DRAM reply during a modular arithmetic load operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set, the error information is also captured in the SPARC Error Status and SPARC Error Address registers. An out-of-bounds error is signaled as a cache line with all data marked with an uncorrectable error. For each 32 bit chunk, the data is loaded into the L2 cache with poisoned ECC. In addition, if the L2 Error Enable nceen bit is set, the uncorrected error information is sent to the modular arithmetic unit, which immediately aborts its asynchronous operation and logs the error in the SPARC Error Status and SPARC Error Address registers.

Once the abort completes, if the SPARC Error Enable nceen bit is set and the ASI_MA_CONTROL_REG int bit is set, a disrupting *data_error* is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt and the normal *modular_arithmetic_interrupt* completion trap is not generated. If the SPARC Error Enable nceen bit is set and the ASI_MA_CONTROL_REG int bit is cleared, a precise *data_access_error* trap is generated to the strand that issues any load from an ASI_MA_* register. If the SPARC Error Enable nceen bit is cleared and the ASI_MA_CONTROL_REG int bit is set, a normal *modular_arithmetic_interrupt* is generated to the strand specified in the ASI_MA_CONTROL_REG as receiving the completion interrupt. If the SPARC Error Enable nceen bit is cleared and the ASI_MA_CONTROL_REG int bit is cleared, the load access to ASI_MA_SYNC_REG will complete normally without generating an error trap.

L2 Error status will report DAU (Uncorrectable error on Access) for this error, while memory will report DBU.

12.8.5.5　DMA Read (DRU/DBU)

When an out-of-bounds error is indicated in the DRAM reply during a DMA read operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set, the error information is also captured in the SPARC Error Status and SPARC Error Address registers. An out-of-bounds error is signaled as a cache line with all data marked with an uncorrectable error. Hardware returns data with an error indicator back to the dma requestor. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

> **Note** In general, DBU errors from DMA should be prevented by the JBI_MEMSIZE register, and instead be detected as a JBUS Nonexistent Memory error.

L2 Error status will report DRU (DMA Uncorrectable) for this error, while memory will report DBU.

12.8.5.6 DMA Write Partial (DRU/DBU)

When an out-of-bounds error is indicated in the DRAM reply during a DMA write-partial operation, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. If the L2 Error Enable nceen bit is set, the error information is also captured in the SPARC Error Status and SPARC Error Address registers. An out-of-bounds error is signaled as a cache line with all data marked with an uncorrectable error. For each 32-bit chunk with an error, the data is loaded into the L2 cache with poisoned ECC. In addition, if the L2 Cache Error Enable nceen and SPARC Error Enable nceen bits are set, a disrupting *data_error* trap is generated to the strand specified in L2_CSR_REG.errorsteer.

If there is an error in the 16-byte block to be written, the DMA write partial will not complete its update, leaving the original data and the poisoned ECC unchanged.

> **Notes** (1) While the DMA write partial does not update the data, the cache line is marked as Dirty.
>
> (2) In general, DBU errors from DMA should be prevented by the JBI_MEMSIZE register, and instead be detected as a JBUS Nonexistent Memory error.

L2 Error status will report DRU (DMA Uncorrectable) for this error, while memory will report DBU.

12.9   DRAM Error Registers

This section describes the error control and log registers for the DRAM. Each DRAM channel has its own set of error registers.

RegisterBaseAddress DRAM CSR Registers – 97 0000 0000₁₆.

12.9.1 DRAM Error Status Register

This register contains status regarding DRAM errors. The status bits in this register are cleared by writing a 1 to the bit position. The error register is *not* cleared on reset, so that software can examine its contents after an error-induced reset.

> **Note** | Since this register is not cleared on reset, after a power-on reset the contents of this register are undefined, and the bits could be in an illegal state that could not possibly be generated by any error combination (for example, meu bit set with all other bits cleared). Operation while in this illegal state leads to undefined behavior for the register, so *software should always clear this register after a power-on reset.*

TABLE 12-19 shows the format of the DRAM Error Status Register.

TABLE 12-17 DRAM Error Status Register – DRAM_ERROR_STATUS_REG (00 0000 0280₁₆)

Bit	Field	Initial Value	R/W	Description
63	meu	Preserved	RW1C	Multiple uncorrected errors; one or more uncorrected errors were not logged.
62	mec	Preserved	RW1C	Multiple corrected errors; one or more corrected errors were not logged.
61	dac	Preserved	RW1C	Set to 1 if the error was a DRAM access correctable error.
60	dau	Preserved	RW1C	Set to 1 if the error was a DRAM access uncorrectable error.
59	dsc	Preserved	RW1C	Set to 1 if the error was a DRAM scrub correctable error.
58	dsu	Preserved	RW1C	Set to 1 if the error was a DRAM scrub uncorrectable error.
57	dbu	Preserved	RW1C	Set to 1 if the error was an access to a nonexistent DRAM address (address out of bounds).
56:16	—	0	R	*Reserved*
15:0	synd	Preserved	RW	ECC syndrome.

If both correctable and uncorrectable errors occur in the same cycle, the mec bit is set and only the appropriate bit for the uncorrectable error is set. The syndrome is captured for the highest-priority error in that cycle (DSU, DAU, are priority 1; DSC, DAC are priority 2). The address is loaded if the highest-priority error in the cycle is either a DSU or DSC.

If there are multiple errors, in different 16-byte chunks, in a single access, they are treated as multiple errors, since there is only logging state to describe a single 16-byte chunk error.

If errors occur in a cycle where an error status bit is already set, TABLE 12-20 applies.

TABLE 12-18 Priority When Error Status Bit Is Set

Existing Error	Error	Priority	Bit Set
DSU/DAU	DSU	1	meu
DSU/DAU	DAU	1	meu
DSU/DAU	DSC	2	mec
DSU/DAU	DAC	2	mec
DSC/DAC	DSU	1	dsu
DSC/DAC	DAU	1	dau
DSC/DAC	DSC	2	mec
DSC/DAC	DAC	2	mec

For the cases above where the "Bit set" column contains a value besides mec and meu, the syndrome and address (for DSU/DSC) for the highest-priority error will overwrite the existing syndrome and address.

Once set, error status bits are only cleared by software. Hardware will never clear a set status bit. If a software write of the error register happens on the same cycle as an error, the setting of bits by the error will be based on the register state before the write. The setting of fields by the error will take precedence over the same field being update by the write; however, fields that are not changed by the error will be updated by the write (for example, if the register has the dac bit set and software does a write to clear that bit on the same cycle as a DRAM scrub uncorrectable error, the error register would end up with the dac bit cleared, the dsu bit set, and the rw and synd fields would contain the values for the error).

12.9.2   DRAM Error Address Register

This register contains the physical address for the DRAM scrub error. DRAM load access address for errors are expected to be logged by L2 controller. The error register is not cleared on reset, so software can examine its contents after an error-induced reset.

TABLE 12-21 shows the format of the DRAM Error Address Register.

TABLE 12-19   DRAM Error Address Register – DRAM_ERROR_ADDRESS_REG (00 0000 0288$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:40	—	0	R	*Reserved*
39:4	address	Preserved	RW	Error address.
3:0	—	0	R	*Reserved*

<table>
<tr><td>**Programming Note**</td><td>This register is writable by software for register diagnostic reasons and isn't expected to be written during normal operation. However, in the event it is written on the same cycle that an error is reported, the update from the error will take precedence over the write.</td></tr>
</table>

12.9.3 DRAM Error Counter Register

Each DRAM channel has an error counter register for use in counting DRAM errors and generating an interrupt when the counter decrements to 0 and both the enb and valid bits are set. Each 16B chunk with an error will cause the count field to decrement by one.

When the count reaches zero, and enb and valid are set, an error interrupt is issued via INT_MAN[1] / INT_CTL[1]

TABLE 12-23 shows the format of the DRAM Error Counter Register.

TABLE 12-20   DRAM Error Counter Register – DRAM_ERROR_COUNTER_REG (00 0000 0298$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	X	R	*Reserved*
17	enb	Preserved	RW	Enables interrupt generation when the counter reaches 0.
16	valid	Preserved	RW	Valid bit for counter value. This bit is reset when count decrements to zero.
15:0	count	Preserved	RW	Counter that decrements with each error when the valid bit is set.

<table>
<tr><td>**Note**</td><td>To use this reg to cause a debug trigger, DRAM_DBG_TRG_EN_REG[EN] must be set, the DRAM_ERROR_COUNTER_REG must be 1$_{16}$ (to trigger on next CE detected), and CLK_CTL[CLKDIS] must be set to choose whether the trigger causes a clock-stop or just an assertion on J_ERR. Note that this register does not count UEs, and thus cannot trigger or interrupt on UEs (but UEs still tend to cause error traps).</td></tr>
</table>

12.9.4 DRAM Error Location Register

Each DRAM channel has an error location register for software to sample to locate a bad memory part.

TABLE 12-24 shows the format of the DRAM Error Location register.

TABLE 12-21 DRAM Error Location Register – DRAM_ERROR_LOCATION_REG (00 000 02A0$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:36	—	0	R	*Reserved*
35:0	location	Preserved	RW	DRAM ECC Error Location, contains the location of the bad nibble. Loaded with each DRAM correctable error. Bits 35:32 correspond to C3..C0, bit 31 corresponds to data{127:124}, …, and bit 0 corresponds to data{3:0}.

12.10 Block Loads and Stores

UltraSPARC T1 supports 64-byte block load and store access to ASI_BLK_P, ASI_BLK_S, ASI_BLK_PL, ASI_BLK_SL, ASI_BLK_AIUP, ASI_BLK_AIUS, ASI_BLK_AIUPL, and ASI_BLK_AIUSL. Loads for these operations consist of four 16-byte "helper" loads, while stores are composed of eight 8-byte "helper" stores.

Errors encountered on any of the helper stores will be logged the same as if it was a nonhelper store, and will not affect the issuing of subsequent helper stores. For the helper loads, errors will be accumulated on all four helpers and reported (and trapped if enabled) after the final helper completes. The error reported will be the uncorrectable error associated with the earliest helper if one or more uncorrectable error(s) were encountered. If no uncorrectable error was encountered, but one or more correctable error(s) were encountered, the error reported will be the correctable error associated with the earliest helper.

The block load is considered a single operation, and even if multiple uncorrectable or correctable errors (or a combination of the two) is encountered, it is treated as a single error of the appropriate type with respect to updating the SPARC Error Status register (that is, the mec and/or meu bits will not be set by the errors in subsequent helpers from the same block load). The floating-point register file will be updated with load results up to the point of the earliest helper that encountered an uncorrectable error. Loading of results into the floating-point register file will be suppressed for the helper that encountered the uncorrectable error and any subsequent helpers.

12.11 CMT Error Summary

TABLE 12-25 summarizes CMT error handling. Terms in the table are defined after the table.

TABLE 12-22 CMT Error Handling Summary *(1 of 3)*

Error Type	Error	SPARC Status	L2 Status	DRAM Status	PA	Syn	Trap	priv	Trap Type	RW	moda
I-TLB data Parity: Ifetch translation	UE	imdu			S	S	p	x	I	—	—
I-TLB data Parity: LDXA to Tag Read reg	UE	imdu			S	S	p	1	D	—	—
I-TLB cam Parity: LDXA to Tag Read reg	UE	imtu			S	S	p	1	D	—	—
D-TLB data parity error: atomic or load translation, LDXA to Tag Read reg	UE	dmdu			S	S	p	x	D	—	—
D-TLB data parity error: store translation	UE	dmsu			S	S	p	x	D	—	—
D-TLB cam Parity: LDXA to Tag Read reg	UE	dmtu			S	S	p	1	D	—	—
I-Cache Data Parity: Ifetch	CE	idc			S	S	di	0	C	—	—
I-cache Tag parity: Ifetch	CE	itc			S	S	di	0	C	—	—
D-cache data parity: cacheable LDs	CE	ddc			S	S	di	0	C	—	—
D-cache tag parity: cacheable LDs	CE	dtc			S	S	di	0	C	—	—
Integer Reg File (IRF) ecc	CE	irc			S	S	di	0	C	—	—
Integer Reg File (IRF) ecc	UE	iru			S	S	p	x	P	—	—
FP Reg File (FRF) ecc	CE	frc			S	S	di	0	C	—	—
FP Reg File (FRF) ecc	UE	fru			S	S	p	x	P	—	—
MA memory parity	UE	mau					p,di	x	D,E	—	—
MA memory parity	UE	mau					p,di	x	D,E	—	—
I/O error from Load from noncacheable address	UE	ncu			S	S	p	x	D	—	—
I/O error from Ifetch from noncacheable address	UE	ncu			S	S	p	x	I	—	—
L2$ data ecc: LD_h, If_h, ATOM_h	CE		ldac		L	L	di	—	C	x	0
L2$ data ecc: PF_h	CE		ldac		L	L	-	—	—	x	0
L2$ data ecc: LD_h, ATOM_h	UE	ldau	ldau		LS	L	p	x	D	x	0
L2$ data ecc: PF_h	UE		ldau		L	L	-	—	—	x	0
L2$ data ecc: If_h	UE	ldau	ldau		LS	L	p	x	I	x	0
L2$ data ecc: MA_ld	CE		ldac		L	L	di(MA)	—	C	x	1
L2$ data ecc: MA_ld_sync_chk	UE	ldau	ldau		LS	L	p	x	D	x	1

TABLE 12-22   CMT Error Handling Summary *(2 of 3)*

Error Type	Error	SPARC Status	L2 Status	DRAM Status	PA	Syn	Trap	priv	Trap Type	RW	moda
L2$ data ecc: MA_ld_interrupt	UE	ldau	ldau		LS	L	di(MA)	x	E	x	1
L2$ data ecc: PST_h	CE		ldac		L	L	di	—	C	x	0
L2$ data ecc: PST_h	UE		ldau		L	L	di	—	E	x	0
L2$ data ecc: PST_h_if	UE		ldau meu		L	L	di	—	E	x	0
L2$ data ecc: wb	CE		ldwc		L	L	di(ES)	—	C	—	—
L2$ data ecc: wb	UE		ldwu		L	L	di(ES)	—	E	—	—
L2$ data ecc:dma_read	CE		ldrc		L	L	di(ES)	—	C	x	—
L2$ data ecc:dma_read	UE		ldru		L	L	di(ES)	—	E	x	—
L2$ data ecc:dma_write_partial	CE		ldrc		L	L	di(ES)	—	C	x	—
L2$ data ecc:dma_write_partial	UE		ldru		L	L	di(ES)	—	E	x	—
L2$ data ecc: scrub	CE		ldsc		L	L	di(ES)	—	C	—	—
L2$ data ecc: scrub	UE		ldsu		L	L	di(ES)	—	E	—	—
L2$ tag ecc: all refs	CE		ltc		L	-	di(ES)	—	C	x	—
L2$ dir parity: scrub	FE		lru		L	L	di(ES)	—	r	—	—
L2$ vad parity: all refs	FE		lvu		L	L	di(ES)	—	r	x	—
Dram ECC error: LD_m_cc, If_m_c32B, ATOM_m_cc	CE		dac	dac	L	D	di	—	C	x	0
Dram ECC error: PF_m_cc	CE		dac	dac	L	D	—	—	-	x	0
Dram ECC error: LD_m_cc_mf, ATOM_m_cc_mf	UE	ldau	dau	dau	LS	D	p	x	D	x	0
Dram ECC error: LD_m_cc_ff, ATOM_m_cc_ff	UE	ldau	dau, meu	dau	LS	D	p, di(ES)	x	D, E	x	0
Dram ECC error: PF_m_cc	UE		dau	dau	L	D	—	—	-	x	0
Dram ECC error: If_m_c32B_mf	UE	ldau	dau	dau	LS	D	p	x	I	x	0
Dram ECC error: If_m_c32B_ff	UE	ldau	dau, meu	dau	LS	D	p, di(ES)	x	I, E	x	0
Dram address out of bounds: LD_m_mf, ATOM_m_mf	UE	ldau	dau	dbu	LS	D	p	x	D	x	0
Dram address out of bounds: LD_m_ff, ATOM_m_ff	UE	ldau	dau, meu	dbu	LS	D	p, d(ES)	x	D, E	x	0
Dram address out of bounds: PF_m	UE		dau	dbu	L	D	di(ES)	x	E	x	0

TABLE 12-22   CMT Error Handling Summary *(3 of 3)*

Error Type	Error	SPARC Status	L2 Status	DRAM Status	PA	Syn	Trap	priv	Trap Type	RW	moda
Dram address out of bounds: If_m_mf	UE	ldau	dau	dbu	LS	D	p	x	I	x	0
Dram address out of bounds: If_m_ff	UE	ldau	dau, meu	dbu	LS	D	p, d(ES)	x	I	x	0
Dram ECC error: MA_ld	CE		dac	dac	L	D	di(MA)	—	C	x	1
Dram ECC error: MA_ld_sync_chk	UE	ldau	dau	dau	LS	D	p	x	D	x	1
Dram address out of bounds: MA_ld_sync_chk	UE	ldau	dau	dbu	LS	D	p	x	D	x	1
Dram ECC error: MA_ld_interrupt	UE	ldau	dau	dau	LS	D	di(MA)	x	E	x	1
Dram address out of bounds: MA_ld_interrupt	UE	ldau	dau	dbu	LS	D	di(MA)	x	E	x	1
Dram ECC error: ST_m,PST_m, LD_m_ncc, If_m_nc32B, ATOM_m_ncc, MA_m_ncc, PF_m_ncc	CE		dac	dac	L	D	di(ES)	—	C	x	—
Dram ECC error: dma_read_req, dma_write_partial	CE		drc	dac	L	D	di(ES)	—	C	x	—
Dram ECC error: ST_m,PST_m, LD_m_ncc, If_m_nc32B, ATOM_m_ncc, MA_m_ncc, PF_m_ncc	UE		dau	dau	L	D	di(ES)	—	E	—	—
Dram ECC error: dma_read_req, dma_write_partial	UE		dru	dau	L	D	di(ES)	—	E	—	—
Dram address out of bounds: ST_m, PST_m, MA_st_m	UE		dau	dbu	L	D	di(ES)	—	E	—	—
Dram address out of bounds: dma_read_req, dma_write_partial	UE		dru	dbu	L	D	di(ES)	—	E	—	—
Dram scrub error	CE		dsc	dsc	D	D	di(ES)	—	C	—	—
Dram scrub error	UE		dsu	dsu	D	D	di(ES)	—	E	—	—

Abbreviations in the Error Type column in Table have the following meaning:

- ATOM_h: Atomic operation hit
- ATOM_m_cc_ff: Atomic operation miss critical 16B chunk, cache fill before miss replay
- ATOM_m_cc_mf: Atomic operation miss critical 16B chunk, miss replay before cache fill

- ATOM_m_ncc: Atomic operation miss noncritical chunk
- dma_read: DMA read any size
- dma_write_partial: subline DMA write
- If_h; I-fetch hit
- If_m_c32B_ff: Ifetch miss critical 32B chunk, cache refill before miss replay
- If_m_c32B_mf: Ifetch miss critical 32B chunk, miss replay before cache fill
- If_m_nc32B: Ifetch miss noncritical 32B chunk
- LD_h: Load hit
- LD_m_cc_ff: Load miss critical 16B chunk, cache fill before miss replay
- LD_m_cc_mf:- Load miss critical 16B chunk, miss replay before cache fill
- LD_m_ncc: Load miss noncritical chunk
- MA_ld: Modular arithmetic load
- MA_ld_interrupt: Modular arithmetic load where MA_CTL.int = 1.
- MA_ld_m_ncc: Modular arithmetic load miss noncritical chunk
- MA_ld_sync_chk: Modular arithmetic load where MA_CTL.int = 0, trap is delivered on load of any `ASI_MA_*` register (normally this would be a load of `ASI_MA_SYNC_REG`)
- MA_st_m: Modular arithmetic store miss
- PF_h: Prefetch hit
- PF_m_cc:Prefetch miss critical 16B chunk
- PF_m_ncc: Prefetch miss noncritical chunk
- PST_h: Partial Store hit
- PST_h_if: Partial Store hit with intervening fill
- PST_m: Partial Store miss
- PF_m_ncc: Prefetch miss noncritical chunk
- wb: writeback to memory

Terms under the other column headings are as follows:

- Error: FE – fatal error; UE – uncorrected error; CE – corrected error.

- SPARC status: entries in the column mark the bits that are set in the SPARC status register when that error occurs

- L2 Status: entries in the column mark the bits that are set in the L2 Status register when that error occurs

- DRAM Status: entries in the column mark the bits that are set in the DRAM Status register when that error occurs

- PA (logging): S – SPARC error address; L – L2 error address, D – DRAM error address

- SYN (logging): S – SPARC error status, L – L2 error status; D – DRAM error status

- Trap: p – precise to requestor: di – disrupting to requestor; di(ES) – disrupting to the strand specified in L2_CSR_REG.errorsteer: di(MA) – disrupting to the strand specified in `ASI_MA_CONTROL_REG`

- priv: x marks cases where priv bit will be set depending on whether strand is executing in privileged mode; 1 marks where the privileged bit will be set to 1 due to the operation only being permitted when PSTATE.priv = 1.

- Trap Type: I – *instruction_access_error*; D – *data_access_error*;
  P – *internal_processor_error*; E – *data_error*; C – *ECC_error*; R – *power_on_reset*.
- RW: x marks cases where L2_ERROR_STATUS.rw bit is set depending on
  whether access was a read or write, atomic operations are treated as writes; 1
  marks where bit will be set to 1 since the operation is a read; 0 marks where the
  bit will be set to 0 since the operation is a write.
- moda: 1 and 0 mark cases where the moda bit will be set and cleared, respectively,
  in L2_ERROR_STATUS.

12.12   JBUS Interface (JBI)

Each detectable JBUS error has individual "error detected" bits in each of two
registers, the JBI_ERROR_LOG and JBI_ERROR_OVF registers. The
JBI_ERROR_LOG logs the first worst error, and the JBI_ERROR_OVF logs any
error that cannot be logged in the JBI_ERROR_LOG (because there is already an
error logged of the same or higher severity). This leaves a reasonable trail in case a
single fault causes multiple error "events," since it shows which was first, plus all
other errors that came after it.

In addition, each detectable error has individual mask bits in each of two CSRs,
jbi_log_enb and jbi_sig_enb. If an error's log_enable bit is clear/off, the error is
completely ignored if it occurs. If an error's log_enable bit is set/on, the error is
logged in either JBI_ERROR_LOG or JBI_ERROR_OVF, plus error information will
be saved in the appropriate error information register(s), if this is the first error or
the first fatal error. Also, if the error can cause a transactional error indication (that
is, error cycle, or error return, or poison data), the log_enable bit enables that
transactional error indication.

When an error occurs, and both its log_enable and signal_enable bits are set, the
error causes either an asynchronous interrupt or a fatal error indication (DOK_ONx4
on JBUS), dependent on the severity of the error.

TABLE 12-26 defines JBUS interface error handling. Errors in the Error column are
described in TABLE 12-27.

TABLE 12-23   JBUS Interface Error Handling

Error	Master	Severity	Logs	log_enb	sig_enb
JBUS Address Parity, [or J_ADP[3] error on indeterminate cycles]	Any	Fatal	J_AD Group *, or J_LOG_DATA0/1 & J_LOG_CTRL **	Asynch Intr	DOK_ONx4***
JBUS Control Parity	N.A.	Fatal	J_AD Group *, or J_LOG_PAR **	Asynch Intr	DOK_ONx4***
Illegal adtype	N.A.	Fatal	J_AD Group *, or J_LOG_CTRL **	Asynch Intr	DOK_ONx4***
L2 Interface Flow Control Timeout	UltraSPA RC T1	Fatal	Just the bit	Asynch Intr	DOK_ONx4***
Arbitration Timeout	UltraSPA RC T1	Fatal	JBI_LOG_ARB	Asynch Intr	DOK_ONx4***
Reported Fatal Error [5:4]	N.A.	Uncorr	Just the bit; one bit per JPACK		Asynch Intr
JBUS Data Parity – DMA Write or INTR to us	Other	Uncorr	J_AD Group *	Poison Data	Asynch Intr
JBUS Data Parity – PIO Read return	UltraSPA RC T1	Uncorr	J_AD Group *	Error Return	Asynch Intr
JBUS Data Parity – Other	Any	Uncorr	J_AD Group *		Asynch Intr
Reported UE Error – DMA Write	Other	Uncorr	J_AD Group *	Poison Data	Asynch Intr
Illegal JBUS Command	Other	Uncorr	J_AD Group *		Asynch Intr
Unsupported JBUS Command	Other	Uncorr	J_AD Group *		Asynch Intr
Nonexistent Memory – DMA Write	Other	Uncorr	J_AD Group *		Asynch Intr
Nonexistent Memory – DMA Read	Other	Uncorr	J_AD Group *	Error Cycle - Unmapped	Asynch Intr
Transaction Timeout – PIO Read Req	UltraSPA RC T1	Uncorr	Just the bit; gets precise trap	Error Return	Asynch Intr
Unmapped Target – PIO Write Req	UltraSPA RC T1	Uncorr	J_AD Group *		Asynch Intr
Read Data Error Cycle – PIO Read	UltraSPA RC T1	Uncorr	J_AD Group *	Error Return	Asynch Intr
Unexpected Data Return	Other	Uncorr	J_AD Group *		Asynch Intr
INTACK Timeout	N.A.	Uncorr	jbi_log_nack		Asynch Intr

* J_AD Group includes JBI_LOG_ADDR, JBI_LOG_DATA0, JBI_LOG_DATA1, JBI_LOG_CTRL, and JBI_LOG_PAR

** These fatal errors log J_AD Group if they are the first error, otherwise they overwrite the specified registers if they are the first fatal error after an uncorrectable error.

*** Asserting the command "DOK_ON" four cycles in a row on JBUS is the defined JBUS protocol means to signal a fatal error.

12.12.1 JBUS Error Descriptions

The JBUS errors listed in TABLE 12-26 are described in TABLE 12-27.

TABLE 12-24 JBUS Error *(1 of 3)*

JBUS Error	Description
JBUS Address Parity Error	Detected when J_ADP{3:0} does not match the calculated parity of J_ADTYPE{7:0}/J_AD{127:0} for any address cycle, or J_ADP{3} does not match the calculated parity of J_ADTYPE{7:0}/J_AD{127:96} for the first cycle of any JBUS packet.
	[The separate J_ADP{3} case is to detect when adtype{7:6} may be the bits in error, converting an address cycle into what looks like a non-address cycle.]
JBUS Control Parity Error	Detected when the current value on J_PAR does not match the calculated parity of the JBUS control signals from 2 cycles ago.
Illegal ADTYPE Error	Detected whenever adtype has an illegal state, usually by having a non-data cycle when a data cycle is required.
L2 Interface Flow Control Timeout	Detected when one of the JBI → L2 interfaces has flow control asserted for longer than the time specified in the L2 Interface Timeout register. This should only happen if something is horribly wrong. A counter counts whenever UltraSPARC T1 has a transaction that it wants to issue, and the count is initialized/reset whenever it successfully issues a transaction.
Arbitration Timeout	Detected when UltraSPARC T1 is unable to issue a transaction (that it wants to issue) for a duration specified by the Arb Timeout register. A counter counts whenever UltraSPARC T1 has a transaction that it wants to issue, and the count is initialized/reset whenever it successfully issues a transaction.
JBUS Reported Fatal Error	Reported when JBI detects some module asserting DOK_ON for four consecutive cycles, indicating that this other module has a fatal error
JBUS Write Data Parity Error	Detected when J_ADP{3:0} does not match the calculated parity of J_AD{127:0} for any write or INTR payload cycle, targeted to us.
JBUS Read Data Parity Error	Detected when J_ADP{3:0} does not match the calculated parity of J_AD{127:0} for any read return data cycle targeted to us.
JBUS Other Data Parity Error	Detected when J_ADP{3:0} does not match the calculated parity of J_AD{127:0} for any data cycle for which UltraSPARC T1 is not the intended recipient of the data.
	NOTE: This error is expected to be enabled only for HW bringup
JBUS Reported UE Error	Reported when JBI receives a write data cycle with a UE (Uncorrected Error) indication.
Illegal JBUS Command	Detected when JBI detects an invalid (reserved) JBUS command.
	If log_enb is zero, illegal JBUS commands are still dropped.
Unsupported JBUS Command	Detected when JBI detects a valid JBUS command that is not supported by UltraSPARC T1, where UltraSPARC T1 is an intended target, but was never expected to be seen in an UltraSPARC T1 system.
	If log_enb is zero, unsupported JBUS commands are still dropped.

TABLE 12-24   JBUS Error *(2 of 3)*

JBUS Error	Description
Nonexistent Memory Error – Write	Detected when UltraSPARC T1 receives a write transaction that is out of the range of installed memory, or to any of UltraSPARC T1's (nonaliased) NC (noncoherent) spaces. In addition, this error will be logged if a coherent write is directed to UltraSPARC T1's aliased NC space, or if a noncoherent write is directed to UltraSPARC T1's coherent memory range.
	If log_enb is zero, the above checks will not be performed, and such transactions will be forwarded to the L2/memory to be handled there.
Nonexistent Memory Error – Read	Detected when UltraSPARC T1 receives a read transaction that is out of the range of installed memory, or to any of UltraSPARC T1's (nonaliased) NC spaces. In addition, this error will be logged if a coherent read is directed to UltraSPARC T1's aliased NC space, or if a non-coherent read is directed to UltraSPARC T1's coherent memory range. JBI will issue an Error Cycle – Unmapped as a JBUS response, and will (if sig_enb) issue an Asynchronous Error Interrupt to the processor.
	NOTE: If a coherent read is issued to any of UltraSPARC T1's 8MB NC spaces, and log_enb is on, the NONEX_RD is logged, but no error cycle is issued. For this case, the issuer should recover be timing out.
	If log_enb is zero, the above checks will not be performed, and such transactions will be forwarded to the L2/memory to be handled there.
Transaction Timeout	Detected when an UltraSPARC T1-launched NCRD transaction is outstanding while the free-running Transaction Timer "wraps" twice. This creates a timeout between 1X and 2X the timer's maximum value. The max/rollover value of the transaction timer is specified in the software-programmable Transaction Timeout Value register.
	NOTE: Transaction timeout detection is completely disabled when JBI_LOG_ENB.read_to = 0, which means that an NCRD transaction that doesn't get a response will patiently wait forever (and thus hang the requestor strand). If JBI_ERR_CONFIG.erren = 0 and JBI_LOG_ENB.read_to = 1, an NCRD transaction will time out based on JBI_TRANS_TIMEOUT.timeval, generating a read_nack error return (and NCU error) to the requesting strand, but no error will be logged in the JBI error logs.
Unmapped Target Error – Write	Detected when UltraSPARC T1 tries to issue a NCWR transaction to a nonexistent module on JBUS. JBI uses the port_pres field in the JBI_CONFIG register, specifying which modules are mapped on a JPACK basis. The transaction will actually be issued on JBUS before being checked, so that the J_AD Group can be used to log the error. • If port_pres[0] is set, transactions to module 0_{16} are allowed. • If port_pres[1] is set, transactions to module 1_{16} are allowed. • If port_pres[4] is set, transactions to modules $1C_{16}$ and $1D_{16}$ are allowed. • If port_pres[5] is set, transactions to modules $1E_{16}$ and $1F_{16}$ are allowed.

TABLE 12-24 JBUS Error *(3 of 3)*

JBUS Error	Description
JBUS Read Data Error Cycle	Detected when JBI receives a Read Data Error Return.
	If log_eng is zero, any Read Data Error Return is dropped. This means that the corresponding NCRD transaction will eventually time out (if enabled).
Unexpected Data Return	Detected when UltraSPARC T1 sees a Data16 matching one of UltraSPARC T1's AIDs, but where the TransID doesn't match an outstanding transaction, or when UltraSPARC T1 sees a Data64 Return matching one of UltraSPARC T1's AIDs. The likely problem is that a TransID or ADTYPE got corrupted.
	If log_enb is zero, unexpected data returns will not be recognized as unexpected, and the data will be returned to some virtual processor, based on the current contents of JBI's Trans_ID translation tables.
INTACK Timeout	Detected whenever an INTR has been sent, but not INTACKed, while the free-running INTACK Timer "wraps" twice. Note that the timer gets reset only on the first INTR, and not on later retries of the same INTR. This creates a timeout between 1X and 2X the timer's maximum value. The max/rollover value of the INTACK timer is specified in the software-programmable INTACK Timeout Value Register.
	NOTE: This timeout mechanism is tracking up to 32 interrupts simultaneously, with a single timer plus two bits per interrupt. A detected error indicates that the OS interrupt mechanism is having a problem servicing interrupts in a timely fashion, or that a strand has become non-communicative. With a 32-bit counter, the max timeout is 20-40 seconds, which working software should be able to guarantee. If not, this check can be disabled.

12.12.2   JBI Error Registers

12.12.2.1 Error Log Registers

Each detectable error has individual "error detected" bits in each of two registers, the JBI_ERROR_LOG and JBI_ERROR_OVF registers. JBI_ERROR_LOG logs the first worst error, and JBI_ERROR_OVF logs any error that cannot be logged in JBI_ERROR_LOG (because there is already an error logged of the same or higher severity). This leaves a reasonable trail in case a single fault causes multiple error "events," since it shows which was first, plus what other errors came after it.

Note that it is possible to have multiple errors logged in the JBI_ERROR_LOG, either by having multiple errors of the same severity detected simultaneously, or by detecting an Uncorrectable error followed by, or in the same cycle as, a fatal error. Also, note that a second occurrence of the same error will always log in the JBI_ERROR_OVF register. Third and fourth occurrences are also "logged" but setting a bit that is already set doesn't make a visible difference.

12.12.2.2 Error Masking

Each detectable error has individual mask bits in each of two CSRs, JBI_LOG_ENB and JBI_SIG_ENB.

> **Note** | JBI_ERR_CONFIG has a global enable bit, erren, that is **and**ed with each individual enable bit, and that erren is cleared (set to 0) on reset. This allows the state of the error enable bits to be preserved through reset, but effectively turned off after an error reset.

If an error's log_enable bit is clear/off, the error is completely ignored if it occurs. No bit is set in JBI_ERROR_LOG or in JBI_ERROR_OVF, no error information is saved in error information registers, and no error response nor indication is generated. A bad transaction may still be dropped regardless of log_enable, as indicated in the error descriptions, particularly if the error prevents a reasonable handling of the transaction.

If an error's log_enable bit is set/on, the error is logged in either JBI_ERROR_LOG or JBI_ERROR_OVF, plus error information will be saved in the appropriate error information register(s), if those registers are not already in use by a prior error of equal or greater severity. Also, if the error can cause a transactional error indication (that is, error cycle, or error return, or poison data), the LOG_ENABLE bit enables that transactional error indication.

When an error occurs, and both its log_enable and signal_enable bits are set, the error causes either an asynchronous interrupt or a fatal error indication (DOK_ONx4), depending on the severity of the error. Of course, software is expected to turn on a signal_enable bit only if the corresponding log_enable bit is also set (but not vice versa).

12.12.2.3 Error Information Registers

In addition, JBI has other error registers, containing information pertinent to the error, such as what was on JBUS when the error occurred (for example, J_AD, J_ADP, J_ADTYPE). These registers get loaded with pertinent data of the first error. In addition, if we get an unconditional error followed by a fatal error, the fatal error will log information most pertinent to the fatal error, as described in TABLE 12-26 on page 170, potentially overwriting a portion of the log from the uncorrectable error.

Most of the errors use the same error information registers, called the "J_AD Group", which consists of JBI_LOG_ADDR, JBI_LOG_DATA0, JBI_LOG_DATA1, JBI_LOG_CTRL, and JBI_LOG_PAR. These registers contain what was on JBUS at the time of the error (DATA0, DATA1, and CTRL) including who owned the bus to put it there, the most recent address cycle before the error (JBI_LOG_ADDR), plus the six previous cycles of adtype (also in JBI_LOG_CTRL)

Control Parity Error uses the J_AD Group, and JBI_LOG_PAR is in the J_AD Group, to simplify the debugging should we need to debug a Control Parity Error during bringup. By taking a snapshot of what was on J_AD (JBI_LOG_DATA*) at the time of the Control Parity Error, and a snapshot of the previous address cycle (JBI_LOG_ADDR), it will be much easier to find the cycle with the Control Parity Error on the logic analyzer trace (assuming one was hooked up).

12.12.2.4 JBUS Error Configuration Register

RegisterBaseAddress 19 JBI – 80 0000 0000$_{16}$. TABLE 12-28 defines the format of the JBUS Error Configuration register.

TABLE 12-25 JBUS Error Configuration – JBI_ERR_CONFIG (0001_{16}–0000_{16})

Bit	Field	Initial Value	R/W	Description
63:5	—	X	R	*Reserved*
4	fe_enb	0	RW	Fatal error enable. This bit controls the assertion of DOK_ON*4 for non-JBI fatal errors.
3	erren	0	RW	Global error enable. This is logically **and**ed with all of the individual log_en bits, such that when 0, error detection and reaction is pretty much off. Exception: see *Transaction Timeout* in TABLE 12-27, page 172.
2	—	0	RW	Not for normal use. This bit should *always* be set to 0.
1:0	—	X	R	*Reserved*

12.12.2.5 JBUS Error Log Registers

TABLE 12-29 defines the format of the JBUS Error Log register.

TABLE 12-26 JBUS Error Log Register – JBI_ERROR_LOG (0001_{16}-0020_{16})

Bit	Field	Initial Value	R/W	Description
63:29	—	0	R	*Reserved*
28	apar	Preserved	RW1C	Address parity error
27	cpar	Preserved	RW1C	Control parity error
26	adtype	Preserved	RW1C	Illegal adtype
25	l2_to	Preserved	RW1C	L2 interface flow control timeout
24	arb_to	Preserved	RW1C	Arbitration timeout
23:18	—	0	R	*Reserved*
17:16	fatal	Preserved	RW1C	Reported fatal error – jpack5 and jpack4
15	dpar_wr	Preserved	RW1C	Data parity error – write to JBI
14	dpar_rd	Preserved	RW1C	Data parity error – read return
13	dpar_o	Preserved	RW1C	Data parity error - other

Bit	Field	Initial Value	R/W	Description
12	rep_ue	Preserved	RW1C	Reported UE error - write
11	illegal	Preserved	RW1C	Illegal JBUS command
10	unsupp	Preserved	RW1C	Unsupported JBUS command
9	nonex_wr	Preserved	RW1C	Nonexistent memory - write
8	nonex_rd	Preserved	RW1C	Nonexistent memory - read
7:6	—	0	R	*Reserved*
5	read_to	Preserved	RW1C	Transaction timeout – CPU read
4	unmap_wr	Preserved	RW1C	Unmapped target – CPU write
3	—	0	R	*Reserved*
2	err_cycle	Preserved	RW1C	Read data error cycle
1	unexp_dr	Preserved	RW1C	Unexpected data return
0	intr_to	Preserved	RW1C	INTACK timeout

This register logs the first and worst error. If an error is detected, and its log_enb bit is set, the corresponding error log bit is set, unless there is another bit of equal or greater severity already logged in this register. More than one bit can be set in this register if either (1) an uncorrectable error is detected first, followed by some fatal error, or (2) two errors are detected on exactly the same cycle.

TABLE 12-30 defines the format of the JBUS Error Overflow register.

TABLE 12-27 JBUS Error Overflow – JBI_ERROR_OVF (000$_{16}$-0028$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:29	—	0	R	*Reserved*
28	apar	Preserved	RW1C	Address parity error
27	cpar	Preserved	RW1C	Control parity error
26	adtype	Preserved	RW1C	Illegal adtype
25	l2_to	Preserved	RW1C	L2 Interface flow control timeout
24	arb_to	Preserved	RW1C	Arbitration timeout
23:18	—	0	R	*Reserved*
17:16	fatal	Preserved	RW1C	Reported fatal error – jpack5 and jpack4
15	dpar_wr	Preserved	RW1C	Data parity error – write to JBI
14	dpar_rd	Preserved	RW1C	Data parity error – read return
13	dpar_o	Preserved	RW1C	Data parity error – other
12	rep_ue	Preserved	RW1C	Reported UE error – write
11	illegal	Preserved	RW1C	Illegal JBUS command
10	unsupp	Preserved	RW1C	Unsupported JBUS command
9	nonex_wr	Preserved	RW1C	Nonexistent memory – write

Bit	Field	Initial Value	R/W	Description
8	nonex_rd	Preserved	RW1C	Nonexistent memory – read
7:6	—	0	R	*Reserved*
5	read_to	Preserved	RW1C	Transaction timeout – CPU read
4	unmap_wr	Preserved	RW1C	Unmapped target – CPU write
3	—	0	R	*Reserved*
2	err_cycle	Preserved	RW1C	Read data error cycle
1	unexp_dr	Preserved	RW1C	Unexpected data return
0	intr_to	Preserved	RW1C	INTACK timeout

This register logs errors that occur after the first/worst error. If an error is detected, and its log_enb bit is set, and there is already an error of equal or greater severity logged in the JBI_ERROR_LOG register, the appropriate error overflow bit is set in this register.

Note that this register is identical in format to JBI_ERROR_LOG, JBI_LOG_ENB, and JBI_SIG_ENB.

12.12.2.6 JBUS Error Mask Registers

TABLE 12-31 defines the format of the JBUS Error Log Enable register.

TABLE 12-28 JBUS Error Log Enable – JBI_LOG_ENB (0001$_{16}$-0030$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:29	—	0	R	*Reserved*
28	apar	Preserved	RW	Address parity error
27	cpar	Preserved	RW	Control parity error
26	adtype	Preserved	RW	Illegal adtype
25	l2_to	Preserved	RW	L2 interface flow control timeout
24	arb_to	Preserved	RW	Arbitration timeout
23:18	—	0	R	*Reserved*
17:16	fatal	Preserved	RW	Reported fatal error – jpack5 and jpack4
15	dpar_wr	Preserved	RW	Data parity error – write to JBI
14	dpar_rd	Preserved	RW	Data parity error – read return
13	dpar_o	Preserved	RW	Data parity error – other
12	rep_ue	Preserved	RW	Reported UE error – write
11	illegal	Preserved	RW	Illegal JBUS command
10	unsupp	Preserved	RW	Unsupported JBUS command
9	nonex_wr	Preserved	RW	Nonexistent memory - write

TABLE 12-28 JBUS Error Log Enable – JBI_LOG_ENB (0001₁₆-0030₁₆) *(Continued)*

Bit	Field	Initial Value	R/W	Description
8	nonex_rd	Preserved	RW	Nonexistent memory - read
7:6	—	0	R	*Reserved*
5	read_to	Preserved	RW	Transaction timeout – CPU read
4	unmap_wr	Preserved	RW	Unmapped target – CPU write
3	—	0	R	*Reserved*
2	err_cycle	Preserved	RW	Read data error cycle
1	unexp_dr	Preserved	RW	Unexpected data return
0	intr_to	Preserved	RW	INTACK timeout

This register controls which errors are detected and logged. If an error's log_enb bit is 0, the error detection circuit is disabled (except for illegal and unsupp, which still drop the transaction), so errors have no effect. Note that the erren bit in the JBI_ERR_CONFIG register is a single overriding disable, so that if erren is 0, it is as though all log_enb bits are 0.

Note that this register is identical in format to JBI_ERROR_LOG, JBI_ERROR_OVF, and JBI_SIG_ENB.

TABLE 12-32 defines the format of the JBUS Error Signal Enable register.

TABLE 12-29 JBUS Error Signal Enable – JBI_SIG_ENB (0001₁₆-0038₁₆)

Bit	Field	Initial Value	R/W	Description
63:29	—	0	R	*Reserved*
28	apar	Preserved	RW	Address parity error
27	cpar	Preserved	RW	Control parity error
26	adtype	Preserved	RW	Illegal adtype
25	l2_to	Preserved	RW	L2 Interface flow control timeout
24	arb_to	Preserved	RW	Arbitration timeout
23:18	—	0	R	*Reserved*
17:16	fatal	Preserved	RW	Reported fatal error – jpack5 and jpack4
15	dpar_wr	Preserved	RW	Data parity error – write to JBI
14	dpar_rd	Preserved	RW	Data parity error – read return
13	dpar_o	Preserved	RW	Data parity error - other
12	rep_ue	Preserved	RW	Reported UE error - write
11	illegal	Preserved	RW	Illegal JBUS command
10	unsupp	Preserved	RW	Unsupported JBUS command
9	nonex_wr	Preserved	RW	Nonexistent memory - write
8	nonex_rd	Preserved	RW	Nonexistent memory - read

Bit	Field	Initial Value	R/W	Description
7:6	—	0	R	*Reserved*
5	read_to	Preserved	RW	Transaction timeout – CPU read
4	unmap_wr	Preserved	RW	Unmapped target – CPU write
3	—	0	R	Reserved
2	err_cycle	Preserved	RW	Read data error cycle
1	unexp_dr	Preserved	RW	Unexpected data return
0	intr_to	Preserved	RW	INTACK timeout

This register controls which errors are proactively reported. If an error's log_enb bit and sig_enb are both 1, the error will be reported as specified in the Error Handling table.

Note that this register is identical in format to JBI_ERROR_LOG, JBI_ERROR_OVF, and JBI_LOG_ENB.

12.12.2.7 JBUS Error Information Registers

TABLE 12-33 defines the format of the JBUS Error Log Address register.

TABLE 12-30 JBUS Error Log Address – JBI_LOG_ADDR (0001$_{16}$-0040$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:61	owner	Preserved	R	Bus owner for most recent address cycle:
				100 = UltraSPARC T1; 010 = REQ5; 001 = REQ4
60:56	—			*Reserved*
55:48	adtype	Preserved	R	adtype from most recent address cycle.
47:43	ttype	Preserved	R	J_AD{47:43} from most recent address cycle.
42:0	addr	Preserved	R	J_AD{42:0} from most recent address cycle.

This register contains transaction address information from the most recent address cycle coincident with or before the detected error. This means that if a loggable error is detected on an address cycle, and that error logs information into the J_AD Group registers (which includes this one), the information logged in the JBI_LOG_ADDR register will be the same as the corresponding fields in the JBI_LOG_DATA0, JBI_LOG_DATA1, and JBI_LOG_CTRL registers.

The intent of this register is to capture the address cycle of a transaction, in case it gets an error on a later data cycle, in which case we can see the address and the transaction ID of the affected transaction.

TABLE 12-34 defines the format of the JBUS Error Log Data0 register.

TABLE 12-31 JBUS Error Log Data0 – JBI_LOG_DATA0 (0001_{16}-0050_{16})

Bit	Field	Initial Value	R/W	Description
63:0	addr	Preserved	R	J_AD{127:64} from cycle with error.

This register is part of the J_AD Group, and is always loaded simultaneously with JBI_LOG_DATA0 and JBI_LOG_CTRL.

TABLE 12-35 defines the format of the JBUS Error Log Detail register.

TABLE 12-32 JBUS Error Log Data1 – JBI_LOG_DATA1 (0001_{16}-0058_{16})

Bit	Field	Initial Value	R/W	Description
63:0	addr	Preserved	R	J_AD{63:0} from cycle with error.

This register is part of the J_AD Group, and is always loaded simultaneously with JBI_LOG_DATA0 and JBI_LOG_CTRL.

TABLE 12-36 defines the format of the JBUS Error Log Control register.

TABLE 12-33 JBUS Error Log Control – JBI_LOG_CTRL (0001_{16}-0048_{16})

Bit	Field	Initial Value	R/W	Description
63:61	owner	Preserved	R	Bus owner for cycle with error
				100 = UltraSPARC T1; 010 = REQ5; 001 = REQ4.
60	—	X	R	*Reserved*
59:56	parity	Preserved	R	ADP{3:0} from cycle with error.
55:48	adtype0	Preserved	R	adtype from cycle with error.
47:40	adtype1	Preserved	R	adtype from 1st cycle before error.
39:32	adtype2	Preserved	R	adtype from 2nd cycle before error.
31:24	adtype3	Preserved	R	adtype from 3rd cycle before error.
23:16	adtype4	Preserved	R	adtype from 4th cycle before error.
15:8	adtype5	Preserved	R	adtype from 5th cycle before error.
7:0	adtype6	Preserved	R	adtype from 6th cycle before error.

This register is part of the J_AD Group, and logs the bus owner (as determined by the arbitration state machine), the parity field for J_AD, and the current cycle and six previous cycles of adtype. For data errors, saving the adtype history will give which data cycle had the error, and the Transaction ID of the failed data return (by seeing the adtype of the first data cycle of the return, and how many cycles ago it was). For illegal adtype errors, since adtype is only illegal in the context of previous adtype cycles, seven cycles of adtype is useful for understanding why the current cycle is illegal.

TABLE 12-37 defines the format of the JBUS Error Log Parity register.

TABLE 12-34 JBUS Error Log Parity – JBI_LOG_PAR (0001_{16}-0060_{16})

Bit	Field	Initial Value	R/W	Description
63:33	—	X	R	*Reserved*
32	jpar	Preserved	R	Received jpar that had parity error.
31:26	—	X	R	*Reserved*
25:23	jpack5	Preserved	R	jpack5 bits used to calculate jpar.
22:20	jpack4	Preserved	R	jpack4 bits used to calculate jpar.
23:14	—	X	R	*Reserved*
13:11	jpack1	Preserved	R	jpack1 bits used to calculate jpar.
10:8	jpack0	Preserved	R	jpack0 bits used to calculate jpar.
7	rsvd4	X	R	*Reserved*
6:0	jreq_l	Preserved	R	j_req_l bits used to calculate jpar; only bits 5,4,0 are implemented. Note that j_req_l is active low.

This register is part of the J_AD Group, and logs information pertaining to a JBUS Control Parity Error. This register contains the bits that went into the JPAR parity calculation.

TABLE 12-38 defines the format of the JBUS Error Log Interrupt Nack register.

TABLE 12-35 JBUS Error Log Interrupt Nack – JBI_LOG_NACK (0001_{16}-0070_{16})

Bit	Field	Initial Value	R/W	Description
63:0	—	X	R	*Reserved*
31:0	intr	Preserved	RW1C	Bit vector of which interrupt(s) time out on not getting an INTACK.

This register logs INTACK Timeout Errors, with one bit per intr that timed out. Unlike most logs, this one doesn't freeze when it first logs a bug, but each subsequent INTACK Timeout error sets another bit in this register.

TABLE 12-39 defines the format of the JBUS Error Log Arb register.

TABLE 12-36 JBUS Error Log Arb – JBI_LOG_ARB (0001_{16}-0078_{16})

Bit	Field	Initial Value	R/W	Description
63:35	—			
34:32	myreq	Preserved	R	Queue of transaction JBI is trying to issue at arb timeout:
				000 = None, or PioReqQ held off by ~AOK/DOK
				001 = PioReqQ 010 = PioAckQ
				011 = SCT0RdQ 100 = SCT1RdQ
				101 = SCT2RdQ 110 = SCT3RdQ
				111 = DbgQ
31:27	—	X	R	*Reserved*

TABLE 12-36 JBUS Error Log Arb – JBI_LOG_ARB (0001_{16}-0078_{16}) *(Continued)*

Bit	Field	Initial Value	R/W	Description
26:24	reqtype	Preserved	R	Type of PioReqQ transaction JBI is trying to issue at arb timeout: 000 = none　　　　　001 = NCRD 100 = NCWR to aid0　101 = NCWR to aid4 110 = NCWR to aid5　111 = NCWR to aid-other
23	—	X	R	*Reserved*
22:16	aok	Preserved	R	AOK states at Arb timeout; only bits 5,4,0 are implemented.
15	rsvd4	X	R	*Reserved*
14:8	dok	Preserved	R	DOK states at Arb timeout; only bits 5,4,0 are implemented.
7	rsvd5	X	R	*Reserved*
6:0	jreq	Preserved	R	State of J_REQ lines at Arb timeout; only bits 5,4,0 are implemented.

The register logs information relating to an Arbitration Timeout error, saving state that may help explain why our arbitration is wedged.

> **Note** | There is a "feature" in JBI_LOG_ARB, such that reqtype field will be "none" if the timeout is caused by ~aok or ~dok, since UltraSPARC T1 backs off requesting the bus. It would be more useful logging whatever is at the front of the PIO_REQ queue, but we only log that if we are actively arbitrating for it when the timeout occurs.

12.12.2.8 JBUS Error Control Registers

Most of the nonlogging error CSRs are timeout value registers, to more accurately specify a timeout hierarchy. To get best failure isolation, timeouts that are dependent on other operations that can timeout, need to be longer than the dependent timeout. In this design, the timeout hierarchy for JBI detected timeouts is as follows:

- L2 Flow Control - Shortest
- Arbitration
- Transaction (PIO Read)
- INTACK – Longest

TABLE 12-40 through TABLE 12-44 define the formats of the JBUS Error Control registers.

TABLE 12-37 JBI L2 Interface Timeout Register – JBI_L2_TIMEOUT (0001_{16}-0080_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—	X	R	*Reserved*
31:0	timeval	Preserved	RW	Number of CPU cycles before a stalled JBI-to-L2 queue invokes a timeout error.

TABLE 12-38 JBI Arbitration Timeout Register – JBI_ARB_TIMEOUT (0001_{16}-0088_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—	X	R	*Reserved*
31:0	timeval	Preserved	RW	Number of JBUS cycles before an unsuccessful arb causes a timeout error.

The JBI Arb timeout counter (not software visible) counts when any JBI transaction is in a queue, regardless of whether it is eligible to be sent to JBUS. The counter clears (resets) when (1) all the queues are empty, or (2) PioQ transaction is sent to JBUS, or (3) PioQ is empty and any (non-PioQ) transaction is sent to JBUS.

TABLE 12-39 JBI Transaction Timeout Register – JBI_TRANS_TIMEOUT (0001_{16}-0090_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—	X	R	*Reserved*
31:0	timeval	Preserved	RW	Minimum number of JBUS cycles before an unreceived read reply causes a timeout error.
				This register *must* be configured to a value that is large enough that it is impossible(!) for a transaction to timeout unless there has been some kind of hardware failure, or the address target is nonexistent.

TABLE 12-40 JBI INTACK Timeout Register – JBI_INTR_TIMEOUT (0001_{16}-0098_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—			*Reserved*
31:0	timeval	Preserved	RW	Minimum number of JBUS cycles before an unacknowledged INTR causes a timeout error.

TABLE 12-41 JBI Memory Size Register – JBI_MEMSIZE (0001_{16}-$00A0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:0	—	X	R	*Reserved*
37:30	size	Preserved	RW	Size of memory, in Gbytes. Used for detecting nonexistent memory errors. The largest legal value for this field is 80_{16} (128), which corresponds to the largest supported memory configuration.
29:0	—	X	R	*Reserved*

12.13 IOB Error

The only errors the IOB detects are accesses to nonexistent modules. For these, it discards writes and NACK reads. The error has no other effect.

12.14 Boot ROM Interface (SSI)

TABLE 12-46 describes the SSI's handling of errors. The error indication on read returns is delivered regardless of the erren bit, where it is up to the strand to ignore the error or receive it. Logging the error and sending an error interrupt are controlled by the erren bit. Note that returning zeros on an I-fetch timeout will tend to cause an illegal instruction trap.

TABLE 12-42 SSI Error Handling

Error	Type	Severity	Logs	Returns	erren
SSI Parity Error	Read	Uncorrectable	Just the bit	Data, with error indication	Asynch Intr
SSI Parity Error	Write	Uncorrectable	Just the bit	N.A.	Asynch Intr
SSI Timeout	Read	Uncorrectable	Just the bit	All zeros, with error indication	Asynch Intr
SSI Timeout	Write	Uncorrectable	Just the bit	N.A.	Asynch Intr

12.14.1 SSI Parity Error

SSI has serial parity on all requests and responses. Odd parity on any response will be treated as a parity error.

On reads, the SSI block will return the data, but marked with an error indication, which will tend to cause an NCU error at the requesting SPARC. For both reads and writes, the SSI block will issue an error interrupt via INT_MAN[1] ÷ INT_CTL[1] (if erren).

12.14.2 SSI Timeout

SSI only supports a single transaction outstanding at any time, and write transactions receive a positive acknowledgement to inform UltraSPARC T1 of their completion. Whenever UltraSPARC T1 issues a transaction, it starts a timer to the

value specified in SSI_TIMEOUT.timeval, which then decrements by one every JBUS cycle. If the time underflows before the transaction completes, it is treated as a timeout.

On reads, the SSI block will return zeros, but marked with an error indication, which will tend to cause an NCU error at the requesting SPARC. For both reads and writes, the SSI block will issue an error interrupt via INT_MAN[1] ÷ INT_CTL[1] (if erren).

12.14.3 SSI Error Registers

RegisterBaseAddress 14 BOOT – FF 0000 0000$_{16}$.   The serial bus interface to the Boot ROM is called SSI; hence the registers dealing with errors on this interface are SSI registers.

TABLE 12-47 and TABLE 12-48 define the format of the SSI Timeout and Log registers, respectively.

TABLE 12-43   SSI Timeout Register– SSI_TIMEOUT (0001$_{16}$-0088$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:25	—	X	R	*Reserved*
24	erren	0	RW	Enables error logging and error interrupt generation in the SSI.
23:0	timeval	800000$_{16}$	RW	Number of JBUS cycles before an unacknowledged request causes a timeout error.

The default value for timeout is about 40 msec.

TABLE 12-44   SSI Log Register – SSI_LOG (0000$_{16}$-0018$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:2	—	X	R	*Reserved*
1	parity	Preserved	RW1C	Parity error detected on response
0	tout	Preserved	RW1C	No response before timeval

12.15   IOP Error Summary

TABLE 12-49 summarizes JBUS error handling. Column abbreviations are described after the table.

TABLE 12-45 JBUS and SSI Error Handling Summary

Error Type	Error	JBI or SSI Status	SPARC Status	L2 Status	DRAM Status	PA	Log/Syn	Trap	Trap Type
JBUS address parity	FE	apar				J	J	WMR	
JBUS control parity	FE	cpar					J	WMR	
Illegal ADTYPE	FE	adtype					J	WMR	
L2 interface flow control timeout	FE	l2_to						WMR	
JBUS arbitration timeout	FE	arb_to					J	WMR	
Reported fatal error	UE	fatal						WMR	
DRAM ECC: RDx	CE			DRC	DAC	L	D	di(ES)	C
DRAM ECC: WRM	CE			DRC	DAC	L	D	di(ES)	C
DRAM ECC: RDx	UE			DRU	DAU	L	D	di(ES), int(IOB)	E
DRAM ECC: WRM	UE			DRU	DAU	L	D	di(ES)	E
L2 tag ECC: any DMA	CE			LTC		L		di(ES)	C
L2 data ECC: any DMA	CE			LDRC		L	L	di(ES)	C
L2 data ECC: RDx	UE			LDRU		L	L	di(ES), int(IOB)	E
L2 data ECC: WRM	UE			LDRU		L	L	di(ES)	E
Reported UE: WRIS to UltraSPARC T1	UE	rep_ue		LDWU	poisoned	JL	JL	int(ERR)	
Reported UE: WRM to UltraSPARC T1	UE	rep_ue			poisoned	J	J	int(ERR)	
JBUS data parity: WRIS to UltraSPARC T1	UE	dpar_wr		LDWU	poisoned	JL	JL	int(ERR)	
JBUS data parity: WRM to UltraSPARC T1	UE	dpar_wr			poisoned	J	J	int(ERR)	
JBUS data parity: INTR payload to UltraSPARC T1	UE	dpar_wr				J	J	int(ERR)	
JBUS data parity: NCRD return to UltraSPARC T1	UE	dpar_rd	NCU			S	J	p, int(ERR)	D
Transaction timeout – PIO Read	UE	read_to	NCU			S		p, int(ERR)	D
JBUS error cycle – PIO Read	UE	err_cycle	NCU			S	J	p, int(ERR)	D
JBUS data parity: other	UE	dpar_o					J	int(ERR)	
Unsupported JBUS command	UE	unsupp				J	J	int(ERR)	
Illegal JBUS command	UE	illegal				J	J	int(ERR)	

TABLE 12-45 JBUS and SSI Error Handling Summary *(Continued)*

Error Type	Error	JBI or SSI Status	SPARC Status	L2 Status	DRAM Status	PA	Log/Syn	Trap	Trap Type
Nonexistent memory – DMA write	UE	nonex-wr				J	J	int(ERR)	
Nonexistent memory – DMA read	UE	nonex_rd				J	J	int(ERR), int(IOB)	
Unmapped target – PIO write	UE	unmap_wr				J	J	int(ERR)	
Unexpected JBUS data return	UE	unexp_dr					J	int(ERR)	
JBUS INTACK timeout	UE	intr_to					J	int(ERR)	
SSI parity – Ifetch	UE	parity	NCU			S		p, int(ERR)	I
SSI parity – Load	UE	parity	NCU			S		p, int(ERR)	D
SSI parity – Store	UE	parity				S		int(ERR)	
SSI timeout – Ifetch	UE	tout	NCU			S		p, int(ERR)	I
SSI timeout – Load	UE	tout	NCU			S		p, int(ERR)	D
SSI timeout – Store	UE	tout				S		int(ERR)	

- Error: FE – fatal error; UE – uncorrected error; CE – corrected error

- PA (logging): J - JBUS error address; S – SPARC error address; L – L2 error address; D – DRAM error address

- LOG/SYN: J - JBUS error logs; S – SPARC error status; L – L2 error status; D – DRAM error status

- Trap: WMR - warm reset; p – precise; di – disrupting; di(ES) – disrupting to the strand specified in L2_CSR_REG.errorsteer; int(ERR) - interrupt to strand and vector specified in INT_MAN[1]; int(IOB) - interrupt to strand specified in external IO-Bridge chip

- Trap Type: D – *data_access_error*; E - *data_error*; C – *ECC_error*

Memory Management Unit

This chapter provides detailed information about the UltraSPARC T1 Memory Management Unit. It describes the internal architecture of the MMU and how to program it.

13.1 Translation Table Entry (TTE)

The Translation Table Entry (TTE) 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.

13.1.1 TTE Tag Format

UltraSPARC T1 supports both the UltraSPARC Architecture 2005 TTE tag format (as described in the *UltraSPARC Architecture 2005* specification; also known as the "sun4v" TTE format) and the older sun4u TTE tag format.

Note that UltraSPARC T1 only supports 13-bit context IDs; therefore, the most significant 3 bits of the (16-bit) context field are always zero.

UltraSPARC T1 supports 48-bit virtual addresses in hardware. When hardware writes a 48-bit virtual address into a 64-bit register, it sign-extends (copies) the most significant address bit (bit 47) into bits 63:48 of the register.

13.1.2 TTE Data Format

For the data portion of the TTE, both the sun4v and sun4u formats are supported by UltraSPARC T1. The sun4v TTE data format is described in the *UltraSPARC Architecture 2005* specification.

UltraSPARC Architecture 2005 specifies a 4-bit size field for TTE entries. Since UltraSPARC T1 only supports a 3-bit size field, the most significant bit of TTE (bit 3) is ignored when written to a TLB and always reads as zero when read from a TLB.

In the sun4u TTE virtual address tag, bits 63:22 are used. Bits 21 through 13 are not maintained in the tag, since these bits are used to index the smallest direct-mapped TSB of 512 entries.

The sun4u TTE data format is shown in TABLE 13-1.

TABLE 13-1 Format 16 Sun4u TTE Data Format

Bit	Field	Description
63	v	Valid
62:61	szl	size{1:0}
60	nfo	No-fault-only
59	ie	Invert endianness
58:49	soft2	Soft2
48	szh	size{2}
47:40	diag	Diagnostic
39:13	pa	PA{39:13}
12:8	soft	Soft
7	—	Reserved
6	l	Locked
5	cp	Cacheable in physically indexed cache
4	cv	Cacheable in virtually indexed cache
3	e	Side effect
2	p	Privileged
1	w	Writable
0	—	Reserved

TABLE 13-2 provides UltraSPARC T1-specific information regarding sun4u TTE data fields.

TABLE 13-2 *TTE Field Description*

Field	Description
nfo	No-Fault-Only. For the IMMU, the nfo bit in the TTE is written into the ITLB, but ignored during ITLB operation. The value of the nfo bit written into the ITLB will be read out on an ITLB Data Access read. **Note:** If the nfo bit is set before loading the TTE into the ITLB, the ITLB miss handler software should generate an error.

TABLE 13-2   *TTE Field Description (Continued)*

Field	Description
ie	Invert Endianness. See *Context and Endianness Selection for Translation* on page 196. For the IMMU, the ie bit in the TTE is written into the ITLB, but ignored during ITLB operation. The value of the ie bit written into the ITLB will be read out on an ITLB Data Access read.
soft, soft2	Software-defined fields, provided for use by the operating system. Software fields are not implemented in UltraSPARC T1 TLB hardware. soft and soft2 fields may be written with any value; they read as zero.
diag	Used by diagnosticsto access the redundant information held in the TLB structure. diag{7} = Used bit. diag{6} = TTE data parity for a valid entry, undefined for an invalid entry[1]. diag{5:3}: Mux selects based on page size: 111 – 256 Mbytes; 011 – 4 Mbytes; 001 – 64 Kbytes; 000 – 8 Kbytes. diag{7:3} are read-only. All other diag bits are reserved.
	[1] Parity is generated across PA{39:13}, Diag{5:3}, v, nfo, ie, l, cp, cv, e, p, and w only. Parity is calculated when loading the TLB, but is not recalculated when demapping an entry. Thus, an invalid data entry with no bit errors will have a correct parity bit if it was loaded via a write to the Data-In or Data-Access register with v = 0, and will have an inverted parity bit if it was loaded via a write to the Data-In or Data-Access register with v = 1 and then demapped.
pa	The physical page number. Page offset bits for larger page sizes (PA{15:13}, PA{21:13}, and PA{27:13}for 64-Kbyte, 4-Mbyte, and 256-Mbyte 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.
w	Writable. For the IMMU, the w bit in the TTE is written into the ITLB, but ignored during ITLB operation. The value of the w bit written into the ITLB will be read out on an ITLB Data Access read.

> **Note** There is a corner case where entry 63 of the UltraSPARC T1 TLB can be replaced, even when locked, when there are other entries unlocked. When all the "used" bits are set, it takes several cycles to clear all the unlocked used bits. During this period, an insert will use entry 63, regardless of whether it is locked.

13.2   Translation Storage Buffer

A Translation Storage Buffer (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 hardware support for TSB access, although the operating system is not required to make use of this support hardware.

Inclusion of the TLB entries in a TSB is not required; that is, translation information may exist in the TLB that is not present in the TSB.

A TSB is arranged as a direct-mapped cache of TTEs. The UltraSPARC T1 MMU provides precomputed pointers into the TSB(s) for both zero and nonzero contexts for two different page sizes: PS0 and PS1, as specified in the following registers:

- `ASI_IMMU_CONTEXT_ZERO_CONFIG_REG`,
- `ASI_IMMU_CONTEXT_NONZERO_CONFIG_REG`,
- `ASI_DMMU_CONTEXT_ZERO_CONFIG_REG`, and
- `ASI_DMMU_CONTEXT_NONZERO_CONFIG_REG`.

In each case, the n least significant bits of a virtual page number is used as the offset from the respective TSB base address, where n equals $\log_2$ of the number of TTEs in the TSB.

A bit in the TSB register allows the PS0 and PS1 pointers to be computed for the case of separate or split PS0/PS1 TSB(s).

No hardware TSB indexing support is provided for TTEs of pages other than PS0 and PS1. Since the TSB is entirely software managed, however, the operating system may choose to place these different page TTEs in the TSB by forming the appropriate pointers. In addition, simple modifications to the PS0 and PS1 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 13-1 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 entries to 16 M entries.

Tag1 (8 bytes)		Data1 (8 bytes)
0000_{16} n Lines in Common TSB	0008_{16}	
Tag$n$ (8 bytes)		Data$n$ (8 bytes)
Tag1 (8 bytes)		Data1 (8 bytes)
$2n$ Lines in Split TSB		
Tag$n$ (8 bytes)		Data$n$ (8 bytes)
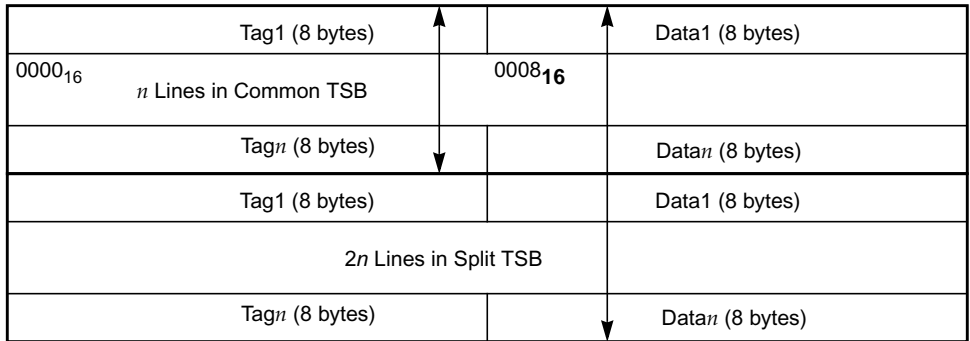
FIGURE 13-1 TSB Organization

13.3 Hardware Support for Hypervisor

To support hypervisor, a number of additions to the MMU are included. First, a 3-bit PID (partition ID) field is included in each TLB entry. This field is loaded with the context of the Partition Identifier register when a TLB entry is loaded. In addition, the PID entry of a TLB is compared against the Partition Identifier register to determine if a TLB hit occurs.

A single r (real translation) bit is included in the TLB entry. This field is loaded with bit 9 from the VA used by the store to data-in or data-access. The real bit distinguishes between VA → PA translations (r = 0) and RA → PA translations (r = 1). If the real bit is 1, the context ID is ignored when determining a TLB hit. Bit 10 of the VA is used on stores of data-in and data-access to select between the sun4u format (VA{10} = 0) and sun4v format (VA{10} = 1). Only the sun4u format is available for data-access loads.

TLB misses on real-to-physical translations generate a *data_real_translation_miss* or *instruction_real_translation_miss* trap instead of the *fast_data_access_MMU_miss* and *fast_inst_access_MMU_miss* traps, respectively. If the hpriv bit of HPSTATE is 0, all privileged ASIs that would normally bypass the TLB, as well as accesses performed while the LSU_CONTROL_REG has the TLBs disabled, use the TLB and need to match against an entry with the r bit set. The final support for hypervisor is that if HPSTATE.hpriv = 1, the MMU is bypassed for all translating ASIs except ASI_*REAL* and ASI_*AS_IF_USER*.

When the MMU is bypassed (bypass occurs under certain LSU_CONTROL_REG settings or in red mode), TABLE 13-3 specifies the physical page attribute bits. This table does not apply to real ASIs; if a real 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 real ASI.

When disabled, both the I-MMU and D-MMU correctly perform all LDXA and STXA operations to internal registers, and traps based on the page attribute bits are signaled just as if the MMU were enabled.

TABLE 13-3 summarizes the page attribute bits.

TABLE 13-3   Physical Page Attribute Bits

Address[39]	Physical Page Attribute Bits							
	cp	ie	cv	e	p	w	nfo	size
0	1	0	0	0	0	1	0	8 KB
1	0	0	0	1	0	1	0	8 KB

13.3.1 Hardware Support for TSB Access

The MMU hardware provides services to allow the TLB miss handler to efficiently reload a missing TLB entry for a PS0 or PS1 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

A typical TLB miss and refill sequence is as follows:

1. A TLB miss causes either a *fast_instruction_access_MMU_miss* or a fast_*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 *MMU Internal Registers and ASI Operations* on page 206 for details. (Note that there are no separate physical registers in

UltraSPARC T1 hardware for the Pointer registers, but rather they are implemented through a dynamic reordering of the data stored in the Tag Access and the TSB registers.)

Pointers are provided by hardware for the most common cases of PS0 and PS1 page miss processing. These pointers give the virtual addresses where the PS0 and PS1 TTEs would be stored if either is present in the TSB.

n is defined to be the tsb_size field of the appropriate zero/nonzero context and PS0/PS1 BASE register; it ranges from 0 to 15. Note that when the TSB is split, tsb_size refers to the size of each TSB. *TSB_Base_PS0* refers to the tsb_base field of the appropriate zero/nonzero context BASE register. tsb_base_ps1 refers to the tsb_base field of the appropriate zero/nonzero context BASE register. PS0 refers to the ps0 field of the Config register. PS1 refers to the ps1 field of the Config register.

Programming Notes	(1) For the split TSB mode, tsb_base_ps0 and tsb_base_ps1 should be set to the same values, or the pair of TSB tables will not be contiguous in virtual memory. Also, note that for a "common" TSB, tsb_base_ps0 and tsb_base_ps1 should be set to the same values with split set to 0. A nonsplit TSB mode can be used with differing tsb_base_ps0 and tsb_base_ps1 values to provide separate TSBs for PS0 and PS1, without requiring them to be contiguous in virtual memory.
	As this latter use of the TSB_BASE_PS{0,1}.split = 0 with differing TSB base pointers can also generate a pair of contiguous TSBs for PS0 and PS1 with the proper loading of the base pointers, the setting of TSB_BASE_PS{0,1}.split to 1 is *deprecated*.
	For the "common" TSB or separate TSBs (BASE register split field = 0):
| | ```
PS0_POINTER = TSB_Base_PS0{YC63:13+N} ::
VA{21+N+3*PS0:13+3*PS0} :: 0000
PS1_POINTER = TSB_Base_PS1{63:13+N} ::
VA{ 21+N+3*PS1:13+3*PS1} :: 0000
``` |
| | For a split TSB (BASE register split field = 1): |
| | ```
PS0_POINTER = TSB_Base_PS0{63:14+N} :: 0 ::
VA{21+N+3*PS0:13+3*PS0} :: 0000
PS1_POINTER = TSB_Base_PS1{63:14+N} :: 1 ::
VA{21+N+3*PS1:13+3*PS1} :: 0000
``` |
| | (2) The TSB pointers assume that the TSB is aligned on a boundary equal to its size. For example, with $n = 12$, the TSB is assumed to be aligned to a 32-Mbyte boundary. Any bits in the tsb_base field from appropriate BASE register that are below the TSB-size boundary are ignored in the pointer calculation. |

The TSB Tag Target (described in *MMU Internal Registers and ASI Operations* on page 206) 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 use of an XOR instruction for TSB hit detection.

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.

13.4   MMU-Related Faults and Traps

TABLE 13-4 lists the traps recorded by the MMU.

TABLE 13-4   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		ı		
instruction_real_translation_miss	iTLB miss		x		
instruction_access_exception	Several (see below)	x	x[1]		
fast_data_access_MMU_miss	dTLB miss				x
data_real_translation_miss	dTLB miss				x
data_access_exception	Several (see below)			x	x
fast_data_access_protection	Protection violation			x	x
privileged_action	Use of privileged ASI			x	
VA_watchpoint	Watchpoint hit			x	
mem_address_not_aligned	Misaligned mem op			x	

1. Contents undefined if *instruction_access_exception* is due to virtual address out of range.

Note	(1) 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.
Programming Note	The Tag Access, SFSR, and SFAR registers can be updated if one of the traps listed in TABLE 13-4 is generated by UltraSPARC T1 on the same cycle as a *trap_level_zero, pic_overflow,* or *instruction_breakpoint* trap (the trap in TABLE 13-4 is suppressed in those cases by UltraSPARC T1, but the update of Tag Access, SFSR, and SFAR is not). If UltraSPARC T1 is configured where it can generate any of those three traps, software cannot assume that Tag Access, SFSR, and SFAR will remain unchanged between the traps listed in TABLE 13-4.

MMU traps are described in TABLE 13-5.

TABLE 13-5   MMU Trap Description

Trap	Description
data_access_exception	Occurs when one of the following events (the D-MMU does not prioritize these and may set multiple bits) occurs: • The D-MMU detects a privilege violation for a data access; that is, an attempted access to a privileged page when PSTATE.priv = 0. • A speculative (nonfaulting) load instruction issued to a page marked with the side-effect (e) bit = 1. • An atomic instruction issued to an I/O address (that is, VA{39} = 1). • 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 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 and PSTATE.am is not set. See *48-bit Virtual Address Space* on page 65 for details.
data_real_translation_miss	Occurs when the MMU is unable to find a translation for a data access that is using a real-to-physical translation; that is, when the appropriate TTE is not present in the data TLB with the r bit set for a memory operation.
fast_data_access_protection	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.
fast_data_access_MMU_miss	Occurs when the MMU is unable to find a translation for a data access that is using a virtual-to-physical translation; that is, when the appropriate TTE is not present in the data TLB with the r bit cleared for a memory operation.

TABLE 13-5   MMU Trap Description *(Continued)*

Trap	Description
fast_instruction_access_MMU_miss	Occurs when the I-MMU is unable to find a translation for an instruction access that is executing using a virtual address; that is, when the appropriate TTE is not present in the iTLB with the r bit cleared.
instruction_access_exception	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 *48-bit Virtual Address Space* on page 65. 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.
instruction_real_translation_miss	Occurs when the I-MMU is unable to find a translation for an instruction access that is executing using a real address; that is, when the appropriate TTE is not present in the iTLB with the r bit set.
mem_address_not_aligned	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.
privileged_action	Occurs when an access is attempted using a *restricted* ASI while in nonprivileged mode (PSTATE.priv = 0 and HPSTATE.hpriv = 0).
VA_watchpoint	Occurs when virtual watchpoints are enabled and the D-MMU detects a load or store to the virtual address specified by the VA Data Watchpoint register.

13.5   MMU Operation Summary

TABLE 13-8 summarizes the behavior of the D-MMU; TABLE 13-10 summarizes the behavior of the I-MMU for normal (non-UltraSPARC T1-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 13-6. TABLE 13-7 lists abbreviations for ASI types.

TABLE 13-6   Abbreviations for MMU Behavior

Abbreviation	Meaning
ok	Normal Translation
dmiss	*fast_data_access_MMU_miss* trap
dreal	*data_real_translation_miss* trap
dexc	*data_access_exception* trap
dprot	*fast_data_access_protection* trap

TABLE 13-6   Abbreviations for MMU Behavior

Abbreviation	Meaning
imiss	*fast_instruction_access_MMU_miss* trap
ireal	*instruction_real_translation_miss* trap
iexc	*instruction_access_exception* trap
pact	*privileged_action* trap

TABLE 13-7   Abbreviations for ASI Types

Abbreviation	Meaning
NUC	ASI_NUCLEUS*
PRIM	Any ASI with PRIMARY translation, except *NO_FAULT and *AS_IF_USER*
SEC	Any ASI with SECONDARY translation, except *NO_FAULT and *AS_IF_USER*
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*
REAL	ASI_*_REAL_* and also other ASIs that require the MMU to perform a bypass operation

> **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 nonexistent ASIs; for example, ASI_PRIMARY_NO_FAULT for a store or atomic; also, access to UltraSPARC T1 internal registers other than LDXA or STXA; the MMU signals a *data_access_exception* trap (SFSR.ft = 08_{16}) for this case.

- Attempted access using a restricted ASI in nonprivileged mode; the MMU signals a *privileged_action* exception for this case.

- An atomic instruction issued to an IO address (PA[39] =1); the MMU signals a *data_access_exception* trap (SFSR.ft = 04_{16}) for this case.

- A data access 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
  (SFSR.ft = 10_{16}) for this case.

- Virtual address out of range and PSTATE.am is not set; the MMU signals a
  data_access_exception trap (SFSR.ft = 20_{16}) for this case.

TABLE 13-8   D-MMU Operations for Normal ASIs

Condition				Behavior				
Instruction	Privilege Mode	ASI	W	TLB Miss	e = 0 p = 0	e = 0 p = 1	e = 1 p = 0	e = 1 p = 1
	nonprivileged	PRIM, SEC	—	dmiss	ok	dexc	ok	dexc
		PRIM_NF, SEC_NF	—	dmiss	ok	dexc	dexc	dexc
		REAL	—	pact	pact	pact	pact	pact
	privileged	PRIM, SEC, NUC	—	dmiss	ok	ok	ok	ok
		PRIM_NF, SEC_NF	—	dmiss	ok	ok	dexc	dexc
Load		U_PRIM, U_SEC	—	dmiss	ok	dexc	ok	dexc
		REAL	—	dreal	ok	ok	ok	ok
	hyperprivileged	PRIM, SEC, NUC	1	—	ok	—	ok	—
		PRIM_NF, SEC_NF	1	—	ok	—	dexc	—
		U_PRIM, U_SEC	—	dmiss	ok	dexc	ok	dexc
		REAL	—	dreal	ok	ok	ok	ok
	nonprivileged		—	ok	ok	ok	ok	ok
FLUSH	privileged		—	ok	ok	ok	ok	ok
	hyperprivileged		—	ok	ok	ok	ok	ok
	nonprivileged	PRIM, SEC	0	dmiss	dprot	dexc	dprot	dexc
			1	dmiss	ok	dexc	ok	dexc
		REAL	—	pact	pact	pact	pact	pact
	privileged	PRIM, SEC, NUC	0	dmiss	dprot	dprot	dprot	dprot
			1	dmiss	ok	ok	ok	ok
		U_PRIM, U_SEC	0	dmiss	dprot	dexc	dprot	dexc
Store or			1	dmiss	ok	dexc	ok	dexc
Atomic		REAL	0	dreal	dprot	dprot	dprot	dprot
			1	dreal	ok	ok	ok	ok
	hyperprivileged	PRIM, SEC, NUC	1	—	ok	—	ok	—
		U_PRIM, U_SEC	0	dmiss	dprot	dexc	dprot	dexc
			1	dmiss	ok	dexc	ok	dexc
		REAL	0	dreal	dprot	dprot	dprot	dprot
			1	dreal	ok	ok	ok	ok

TABLE 13-9   I-MMU Operations for Normal ASIs

Condition	Behavior		
Privilege Mode	**TLB Miss**	**p = 0**	**p = 1**
user	imiss	ok	iexc
privileged	imiss	ok	
hyperprivileged	—	ok	—

13.6   Context and Endianness Selection for Translation

This information is described in the Memory Management chapter of the *UltraSPARC Architecture 2005* specification.

13.7   Translation

The translation operation of MMU is determined by the LSU_CONTROL_REG and the HPSTATE registers. TABLE 13-12 describes the operation of the I-MMU.

TABLE 13-10 IMMU Translation

Control State			
LSU.im	**HPSTATE.hpriv**	**HPSTATE.red**	**IMMU Translation**
Don't care	Don't care	1	Bypass[1]
Don't care	1	0	Bypass[1]
0	0	0	RA $\rightarrow$ PA[2]
1	0	0	VA $\rightarrow$ PA[3]

1. VA{39:0} passed directly to PA{39:0}

2. VA{63:0} passed directly to RA{63:0}, RA{63:0} translated via the IMMU.

TABLE 13-14 describes the operation of the D-MMU.

TABLE 13-11 DMMU Translation

Control State			
LSU.dm		**HPSTATE.hpriv**	**DMMU Translation**
0		0	RA $\rightarrow$ PA
0		1	Follows TABLE 13-16
1		See TABLE 13-16	Follows TABLE 13-16

In TABLE 13-16, which describes DMMU address translation behavior, the following abbreviations are used:

Abbreviation	Meaning
DAE	*data_access_exception* exception is triggered
priv_action	*privileged_action* exception is triggered
—	"nontranslating" (as defined in *UltraSPARC Architecture 2005*)

TABLE 13-12 DMMU Translation when (LSU_CONTROL_REG.dm = 1 or HPSTATE.hpriv = 1) *(1 of 5)*

ASI Value	ASI NAME	Translation		
		Nonprivileged	Privileged	Hyperprivileged
Mandatory SPARC V9 ASIs				
04_{16}	ASI_NUCLEUS	*priv_action*	VA $\rightarrow$ PA	bypass
$0C_{16}$	ASI NUCLEUS_LITTLE	*priv_action*	VA $\rightarrow$ PA	bypass
10_{16}	ASI_AS_IF_USER_PRIMARY	*priv_action*	VA $\rightarrow$ PA	VA $\rightarrow$ PA
11_{16}	ASI_AS_IF_USER_SECONDARY	*priv_action*	VA $\rightarrow$ PA	VA $\rightarrow$ PA
18	ASI_AS_IF_USER_PRIMARY_LITTLE	*priv_action*	VA $\rightarrow$ PA	VA $\rightarrow$ PA
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE	*priv_action*	VA $\rightarrow$ PA	VA $\rightarrow$ PA
80_{16}	ASI_PRIMARY	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
81_{16}	ASI_SECONDARY	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
82_{16}	ASI_PRIMARY_NO_FAULT	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
83_{16}	ASI_SECONDARY_NO_FAULT	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
88_{16}	ASI_PRIMARY_LITTLE	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
89_{16}	ASI_SECONDARY_LITTLE	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass
$8B_{16}$	ASI_SECONDARY_NO_FAULT_LITTLE	VA $\rightarrow$ PA	VA $\rightarrow$ PA	bypass

ASI Value	ASI NAME	Translation		
		Nonprivileged	Privileged	Hyperprivileged
	Additional ASIs			
14_{16}	ASI_REAL	*priv_action*	RA → PA	RA → PA
15_{16}	ASI_REAL_IO	*priv_action*	RA → PA	RA → PA
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY	*priv_action*	VA → PA	VA → PA
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY	*priv_action*	VA → PA	VA → PA
$1C_{16}$	ASI_REAL_LITTLE	*priv_action*	RA → PA	RA → PA
$1D_{16}$	ASI_REAL_IO_LITTLE	*priv_action*	RA → PA	RA → PA
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE	*priv_action*	VA → PA	VA → PA
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE	*priv_action*	VA → PA	VA → PA
20_{16}	ASI_SCRATCHPAD	*priv_action*	—	—
21_{16}	ASI_MMU	*priv_action*	—	—
22_{16}	ASI_TWINX_AS_IF_USER_PRIMARY (ASI_LDTX_AIUP) ASI_ST_BLKINIT_AS_IF_USER_PRIMARY (ASI_STBI_AIUP)	*priv_action*	VA → PA	VA → PA
23_{16}	ASI_TWINX_AS_IF_USER_SECONDARY (ASI_LDTX_AIUS) ASI_ST_BLKINIT_AS_IF_USER_SECONDARY (ASI_STBI_AIUS)	*priv_action*	VA → PA	VA → PA
24_{16}	ASI_QUAD_LDD	*priv_action*	VA → PA	bypass
25_{16}	ASI_QUEUE	*priv_action*	—	—
26_{16}	ASI_QUAD_LDD_REAL	*priv_action*	RA → PA	RA → PA
27_{16}	ASI_TWINX_NUCLEUS (ASI_LDTX_N) ASI_ST_BLKINIT_NUCLEUS (ASI_STBI_N)	*priv_action*	VA → PA	bypass
$2A_{16}$	ASI_TWINX_AS_IF_USER_PRIMARY_LITTLE (ASI_LDTX_AIUP_L) ASI_ST_BLKINIT_AS_IF_USER_PRIMARY_LITTLE (ASI_STBI_AIUP_L)	*priv_action*	VA → PA	VA → PA
$2B_{16}$	ASI_TWINX_AS_IF_USER_SECONDARY_LITTLE (ASI_LDTX_AIUS_L) ASI_ST_BLKINIT_AS_IF_USER_SECONDARY_LITTLE (ASI_STBI_AIUS_L)	*priv_action*	VA → PA	VA → PA
$2C_{16}$	ASI_TWINX_LITTLE (ASI_LDTX_L), ASI_QUAD_LDD_LITTLE[D], ASI_NUCLEUS_QUAD_LDD_LITTLE[D]	*priv_action*	VA → PA	bypass
$2E_{16}$	ASI_TWINX_REAL_LITTLE (ASI_LDTX_REAL_L) ASI_QUAD_LDD_REAL_LITTLE[D]	*priv_action*	RA → PA	RA → PA

TABLE 13-12 DMMU Translation when (LSU_CONTROL_REG.dm = 1
or HPSTATE.hpriv = 1) *(3 of 5)*

ASI Value	ASI NAME	Translation		
		Nonprivileged	Privileged	Hyperprivileged
$2F_{16}$	ASI_TWINX_NUCLEUS_LITTLE (ASI_LDTX_NL) ASI_ST_BLKINIT_NUCLEUS_LITTLE (ASI_STBI_NL)	*priv_action*	$VA \rightarrow PA$	bypass
31_{16}	ASI_DMMU_CTXT_ZERO_TSB_BASE_PS0	*priv_action*	*DAE*	—
32_{16}	ASI_DMMU_CTXT_ZERO_TSB_BASE_PS1	*priv_action*	*DAE*	—
33_{16}	ASI_DMMU_CTXT_ZERO_CONFIG	*priv_action*	*DAE*	—
35_{16}	ASI_IMMU_CTXT_ZERO_TSB_BASE_PS0	*priv_action*	*DAE*	—
36_{16}	ASI_IMMU_CTXT_ZERO_TSB_BASE_PS1	*priv_action*	*DAE*	—
37_{16}	ASI_IMMU_CTXT_ZERO_CONFIG	*priv_action*	*DAE*	—
39_{16}	ASI_DMMU_CTXT_NONZERO_TSB_BASE_PS0	*priv_action*	*DAE*	—
$3A_{16}$	ASI_DMMU_CTXT_NONZERO_USB_BASE_PS1	*priv_action*	*DAE*	—
$3B_{16}$	ASI_DMMU_CTXT_NONZERO_CONFIG	*priv_action*	*DAE*	—
$3D_{16}$	ASI_IMMU_CTXT_NONZERO_TSB_BASE_PS0	*priv_action*	*DAE*	—
$3E_{16}$	ASI_IMMU_CTXT_NONZERO_USB_BASE_PS1	*priv_action*	*DAE*	—
$3F_{16}$	ASI_IMMU_CTXT_NONZERO_CONFIG	*priv_action*	*DAE*	—
40_{16}	ASI_STREAM_MA	*priv_action*	*DAE*	—
42_{16}	ASI_SPARC_BIST_CONTROL	*priv_action*	*DAE*	—
42_{16}	ASI_INST_MASK_REG	*priv_action*	*DAE*	—
42_{16}	ASI_LSU_DIAG_REG	*priv_action*	*DAE*	—
44_{16}	ASI_STM_CTL_REG	*priv_action*	*DAE*	—
45_{16}	ASI_LSU_CONTROL_REG	*priv_action*	*DAE*	—
46_{16}	ASI_DCACHE_DATA	*priv_action*	*DAE*	—
47_{16}	ASI_DCACHE_TAG	*priv_action*	*DAE*	—
48_{16}	ASI_INTR_DISPATCH STATUS	*priv_action*	*DAE*	*DAE*
49_{16}	ASI_INTR_RECEIVE	*priv_action*	*DAE*	*DAE*
$4A_{16}$	ASI_UPA_CONFIG_REGISTER	*priv_action*	*DAE*	*DAE*
$4B_{16}$	ASI_SPARC_ERROR_EN_REG	*priv_action*	*DAE*	—
$4C_{16}$	ASI_SPARC_ERROR_STATUS_REG	*priv_action*	*DAE*	—
$4D_{16}$	ASI_SPARC_ERROR_ADDRESS_REG	*priv_action*	*DAE*	—
$4E_{16}$	ASI_ECACHE_TAG_DATA	*priv_action*	*DAE*	*DAE*
$4F_{16}$	ASI_HYP_SCRATCHPAD	*priv_action*	*DAE*	—
50_{16}	ASI_IMMU	*priv_action*	*DAE*	—
51_{16}	ASI_IMMU_TSB_PS0_PTR_REG	*priv_action*	*DAE*	—
52_{16}	ASI_IMMU_TSB_PS1_PTR_REG	*priv_action*	*DAE*	—
54_{16}	ASI_ITLB_DATA_IN_REG	*priv_action*	*DAE*	—
55_{16}	ASI_ITLB_DATA_ACCESS_REG	*priv_action*	*DAE*	—

ASI Value	ASI NAME	Translation		
		Nonprivileged	Privileged	Hyperprivileged
56_{16}	ASI_ITLB_TAG_READ_REG	*priv_action*	*DAE*	—
57_{16}	ASI_IMMU_DEMAP	*priv_action*	*DAE*	—
58_{16}	ASI_DMMU	*priv_action*	*DAE*	—
59_{16}	ASI_DMMU_TSB_PS0_PTR_REG	*priv_action*	*DAE*	—
$5A_{16}$	ASI_DMMU_TSB_PS1_PTR_REG	*priv_action*	*DAE*	—
$5B_{16}$	ASI_DMMU_TSB_DIRECT_PTR_REG	*priv_action*	*DAE*	—
$5C_{16}$	ASI_DTLB_DATA_IN_REG	*priv_action*	*DAE*	—
$5D_{16}$	ASI_DTLB_DATA_ACCESS_REG	*priv_action*	*DAE*	—
$5E_{16}$	ASI_DTLB_TAG_READ_REG	*priv_action*	*DAE*	—
$5F_{16}$	ASI_DMMU_DEMAP	*priv_action*	*DAE*	—
60_{16}	ASI_TLB_INVALIDATE_ALL	*priv_action*	*DAE*	—
66_{16}	ASI_ICACHE_INSTR	*priv_action*	*DAE*	—
67_{16}	ASI_ICACHE_TAG	*priv_action*	*DAE*	—
72_{16}	ASI_SWVR_INTR_RECEIVE	*priv_action*	*DAE*	—
73_{16}	ASI_SWVR_UDB_INTR_W	*priv_action*	*DAE*	—
74_{16}	ASI_SWVR_UDB_INTR_R	*priv_action*	*DAE*	—
76_{16}	ASI_ECACHE_W	*priv_action*	*DAE*	*DAE*
77_{16}	ASI_UDB_INTR_W	*priv_action*	*DAE*	*DAE*
$7E_{16}$	ASI_ECACHE_R	*priv_action*	*DAE*	*DAE*
$7F_{16}$	ASI_UDB_INTR_R	*priv_action*	*DAE*	*DAE*
$C0_{16}$–$C5_{16}$	ASI_PST*	*DAE*	*DAE*	*DAE*
$C8_{16}$–CD_{16}	ASI_PST*	*DAE*	*DAE*	*DAE*
$D0_{16}$–$D3_{16}$	ASI_FL*	*DAE*	*DAE*	*DAE*
$D8_{16}$, $D9_{16}$	ASI_FL*	*DAE*	*DAE*	*DAE*
DA_{16}, DB_{16}	ASI_FL*	*DAE*	*DAE*	*DAE*
$E0_{16}$	ASI_BLK_COMMIT_P	*DAE*	*DAE*	*DAE*
$E1_{16}$	ASI_BLK_COMMIT_S	*DAE*	*DAE*	*DAE*
$E2_{16}$	ASI_TWINX_PRIMARY (ASI_LDTX_P) ASI_ST_BLKINIT_PRIMARY (ASI_STBI_P)	VA → PA	VA → PA	bypass
$E3_{16}$	ASI_TWINX_SECONDARY (ASI_LDTX_S) ASI_ST_BLKINIT_SECONDARY (ASI_STBI_S)	VA → PA	VA → PA	bypass

ASI Value	ASI NAME	Translation		
		Nonprivileged	Privileged	Hyperprivileged
EA_{16}	`ASI_TWINX_PRIMARY_LITTLE (ASI_LDTX_PL)` `ASI_ST_BLKINIT_PRIMARY_LITTLE (ASI_STBI_PL)`	VA → PA	VA → PA	bypass
EB_{16}	`ASI_TWINX_SECONDARY_LITTLE (ASI_LDTX_SL)` `ASI_ST_BLKINIT_SECONDARY_LITTLE (ASI_STBI_SL)`	VA → PA	VA → PA	bypass
$F0_{16}$	`ASI_BLK_P`	VA → PA	VA → PA	bypass
$F1_{16}$	`ASI_BLK_S`	VA → PA	VA → PA	bypass
$F8_{16}$	`ASI_BLK_PL`	VA → PA	VA → PA	bypass
$F9_{16}$	`ASI_BLK_SL`	VA → PA	VA → PA	bypass

13.8 MMU Behavior During Reset and Upon Entering `RED_state`

During global reset of the UltraSPARC T1 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 (nonsuspended) operation.
- The I-MMU and D-MMU Enable bits in the LSU Control Register (see *ASI_LSU_CONTROL_REG Register* on page 236) 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.

> **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 to the `ASI_TLB_INVALIDATE_ALL`. 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.

See the Memory Management chapter in the *UltraSPARC Architecture 2005* for more information regarding MMU behavior during resets and transitions to `RED_state`.

13.9   MMU Internal Registers and ASI Operations

This section describes aspects of access to MMU internal registers that are specific to the UltraSPARC T1 processor. For general information on this topic, consult the *UltraSPARC Architecture 2005*

13.9.1   Accessing MMU Registers

All internal MMU registers can be accessed directly by the CPU through UltraSPARC T1-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. Use of %g0 for the address when accessing an internal ASI register reduces the number of instructions needed to perform the access to the alternate space (by eliminating address formation).

If the three least-significant bits of the virtual address are nonzero (VA{2:0} = 0) in an LDXA or STXA instruction to or 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 (SFSR.ft = 08_{16}). (UltraSPARC T1 hardware detects VA violations in only an unspecified lower portion of the virtual address (impl. dep. #414-S10.))

> **Programming Note**   UltraSPARC T1 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{47}. Software must guarantee that the VA is within range.

Writes to the TSB register, Tag Access register, and VA Watchpoint Address registers are not checked for out-of-range virtual addresses. No matter what is written to any of those register, VA{63:48} will always be identical to VA{47} on a read.

TABLE 13-17 describes the internal registers and ASI operations specific to UltraSPARC T1.

TABLE 13-13   UltraSPARC T1 MMU Internal Registers and ASI Operations

I-MMU ASI	D-MMU ASI	VA{63:0}	Access (R/W)	Replication Level	Register or Operation Name
—	21_{16}	8_{16}	RW	Strand	(UA-2005) Primary Context Register
—	21_{16}	10_{16}	RW	Strand	(UA-2005) Secondary Context Register
35_{16}	31_{16}	0_{16}	RW	Strand	I-/D-Context Zero TSB Base PS0
36_{16}	32_{16}	0_{16}	RW	Strand	I-/D-Context Zero TSB Base PS1

TABLE 13-13 UltraSPARC T1 MMU Internal Registers and ASI Operations *(Continued)*

I-MMU ASI	D-MMU ASI	VA{63:0}	Access (R/W)	Replication Level	Register or Operation Name
37_{16}	33_{16}	0_{16}	RW	Strand	I-/D-Context Zero Config
$3D_{16}$	39_{16}	0_{16}	RW	Strand	I-/D-Context Nonzero TSB Base PS0
$3E_{16}$	$3A_{16}$	0_{16}	RW	Strand	I-/D-Context Nonzero TSB Base PS1
$3F_{16}$	$3B_{16}$	0_{16}	RW	Strand	I-/D-Context Nonzero Config
50_{16}	58_{16}	0_{16}	R	Strand	(UA-2005) I-/D-TSB Tag Target registers
50_{16}	58_{16}	18_{16}	RW	Strand	I-/D-TLB Synchronous Fault Status register
—	58_{16}	20_{16}	R	Strand	(UA-2005) D-TLB Synchronous Fault Address register
50_{16}	58_{16}	30_{16}	RW	Strand	(UA-2005) I-/D-TLB Tag Access registers
—	58_{16}	38_{16}	RW	Strand	Virtual Watchpoint Address
—	58_{16}	80_{16}	RW	Strand	(UA-2005) Partition Identifier
51_{16}	59_{16}	0_{16}	R	Strand	I-/D-TSB PS0 Pointer registers
52_{16}	$5A_{16}$	0_{16}	R	Strand	I-/D-TSB PS1 Pointer registers
—	$5B_{16}$	0_{16}	R	Strand	D-TSB Direct Pointer register
54_{16}	$5C_{16}$	0_{16}	W	Physical Core	(UA-2005) I-/D-TLB Data In registers
55_{16}	$5D_{16}$	$0_{16}..1F8_{16}$	RW	Physical Core	(UA-2005) I-/D-TLB Data Access registers
56_{16}	$5E_{16}$	$0_{16}..1F8_{16}$	R	Physical Core	(UA-2005) I-/D-TLB Tag Read register
57_{16}	$5F_{16}$	See 13.10	W	Strand	(UA-2005) I-/D-MMU Demap Operation
60_{16}	—	0_{16}	W	Physical Core	I-TLB Invalidate All
—	60_{16}	8_{16}	W	Physical Core	D-TLB Invalidate All

13.9.2 Context ID Registers

The Primary and Secondary context registers are shared by the I- and D-MMUs, and the Nucleus Context register is hardwired to a zero value, as defined in the *UltraSPARC Architecture 2005* specification.

On UltraSPARC T1, the context ID fields (pcontextid and scontextid) are implemented as 13-bit fields (bits 12:0) in the context registers (impl. dep. #415-S10).

13.9.3 I-/D- TSB Base Registers

The following UltraSPARC T1-specific TSB registers provide information for the hardware formation of TSB pointers and tag target, to assist software in handling TLB misses quickly:

- I/DMMU_CONTEXT_ZERO_TSB_BASE_PS0
- I/DMMU_CONTEXT_ZERO_TSB_BASE_PS1
- I/DMMU_CONTEXT_NONZERO_TSB_BASE_PS0
- I/DMMU_CONTEXT_NONZERO_TSB_BASE_PS1

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. Separate pointers exist for the I-MMU and D-MMU, as well as for two page sizes (PS0 and PS1, specified by the I-/D-TSB Control Registers), and for zero (nucleus) and nonzero contexts.

FIGURE 13-2 illustrates the TSB register. The fields are described in TABLE 13-18.

tsb_base{63:13} (virtual)	split	—	tsb_size
63	13 12	11 4	3 0

FIGURE 13-2 I-/D-TSB Register Format

TABLE 13-14 Bit Description of TSB Register Fields

Bit	Field	Description
63:13	tsb_base	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{47}.
12	split	When split = 1, the TSB PS1 Pointer address is calculated assuming separate (but abutting and equally sized) TSB regions for the PS0 and the PS1 TTEs. In this case, tsb_size refers to the size of each TSB. **Programming Note:** When split = 1, TSB_BASE_PS0 and TSB_BASE_PS1 must be identical to get the pair of TSB regions to be abutting and equally sized. Since UltraSPARC T1 provides separate TSB base pointers for PS0 and PS1, the use of split = 1 is deprecated. When split = 0, the TSB PS0 and PS1 Pointer addresses are calculated based on their base pointer. By setting the TSB_BASE_PS0 and TSB_BASE_PS1 pointers to different values, separate TSB can be used for PS0 and PS1, without the requirement that they be abutting and equally sized. By setting TSB_BASE_PS0 equal to TSB_BASE_PS1, a "common TSB" configuration can be created that assumes that the same lines in the TSB are shared by PS0 and PS1 TTEs. **Programming Note:** In the "common TSB" configuration (TSB.split = 0 with TSB_BASE_PS0 and TSB_BASE_PS1 identical), PS0 and PS1 page TTEs can conflict, unless the TLB miss handler explicitly checks the TTE for page size. Therefore, do not use the "common TSB" configuration in an optimized handler (for example, by setting the TSB Base pointers for PS0 and PS1 to differing non-overlapping locations in memory). 　　For example, suppose an PS0 = 8-Kbyte page at VA = 2000_{16} and a PS1 = 64-Kbyte 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.

TABLE 13-14 Bit Description of TSB Register Fields *(Continued)*

Bit	Field	Description
3:0	tsb_size	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-Kbyte common/ separate TSB, 16-Kbyte split TSB), to 16-Mbyte entries at tsb_size = 15 (256 Mbyte common/separate TSB, 512 MB; split TSB). **Note:** When the page size for a tsb_base is set to 5 (256-Mbyte pages), setting tsb_size to a value greater than 11 is larger than the 48-bit VA range and is a programming error. **Note:** Any update to the TSB register immediately affects the data that is returned from later reads of the TSB Pointer registers.

13.9.4   I-/D- TSB Config Registers

The TSB config registers (I/DMMU_CONTEXT_ZERO_CONFIG, I/DMMU_CONTEXT_NONZERO_CONFIG) provide the values for PS1 and PS0 for the TSB Registers.

FIGURE 13-6 illustrates the TSB Config register; the fields are described below.

—	ps1	—	ps0
63	11 10 8	7	3 2 0

Bit	Field	Description
10:8	ps1	Page size for matching *TSB_BASE_PS1.
2:0	ps0	Page size for matching TSB_BASE_PS0.

Programming Note | If a write is performed that attempts to write an unsupported page size into ps1 (and/or ps0), the value of 5_{16} (256 Mbyte page size) will instead be loaded into ps1 (and/or ps0), and no error trap will be generated by the hardware.

13.9.5   I-/D-TSB Tag Target Registers

The UltraSPARC T1 I- and D-TSB Tag Target registers are bit-shifted versions of the data stored in the I- and D-Tag Access registers. See APpendix E of the *UltraSPARC Architecture 2005* for a description of these registers.

13.9.6   I-/D-MMU Synchronous Fault Status Registers (SFSR)

The UltraSPARC T1 I- and D-MMU maintain their own SFSR register, as illustrated in FIGURE 13-3 and defined in TABLE 13-19.

—	asi	—	ft	e	ct	—	w	ow	fv

63   24 23   16 15 14 13 7 6 5   4 3 2 1 0

FIGURE 13-3 I- and D-MMU Synchronous Fault Status Register Format

TABLE 13-15   SFSR Bit Description

Bit	Field	Description
63:24	—	*Reserved*
23:16	asi	ASI. 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. For the D-SFSR, a trapping nonalternate load or store sets the default ASI, namely, ASI_PRIMARY or ASI_PRIMARY_LITTLE when SFSR.tl = 0 or ASI_NUCLEUS or ASI_NUCLEUS_LITTLE when SFSR.tl > 0. JMPL and RETURN *mem_address_not_aligned* traps ignore SFSR.tl and always set ASI_PRIMARY (for PSTATE.cle = 0) or ASI_PRIMARY_LITTLE (for PSTATE.cle = 1).
15:14	—	*Reserved*
13:7	ft	Fault type. Indicates the exact condition that caused the recorded fault, according to TABLE 13-20, following this table. 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 *fast_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.
6	e	Side effect. Reports the side-effect bit (e) associated with the faulting data access; set by translating ASI accesses (see *Translation* on page 196) mapped by the TLB with the e bit set, translating ASI accesses with PA{39} set when translation is bypassed (when translation is bypassed the default attributes of TABLE 13-3 on page 187 apply) and accesses to the ASI_REAL_IO{_LITTLE} ASIs (15_{16} and $1D_{16}$). Other cases that update the SFSR (that is, internal ASI accesses) set the e bit to 0. It always reads as 0 in the I-MMU.

TABLE 13-15 SFSR Bit Description *(Continued)*

Bit	Field	Description
5:4	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 13.7).

Context ID	I-MMU Context	D-MMU Context
00	Primary	Primary
01	*Reserved*	Secondary
10	Nucleus	Nucleus
11	*Reserved*	*Reserved*

Bit	Field	Description
		Implementation Note: JMPL and RETURN *mem_address_not_aligned* traps set ct to either Primary or Nucleus, but in a nondeterministic way (that is, the context value set in the SFSR does not reflect the context value used by the JMPL/RETURN).
3	—	*Reserved*
2	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.
1	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.
0	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

TABLE 13-16 MMU Synchronous Fault Status Register ft (Fault Type) Field

ft{6:0}	Fault Type
01_{16}	Privilege violation.
02_{16}	Speculative load from page marked with e bit. This bit is zero for internal ASI accesses.
04_{16}	Atomic to I/O address (PA{39} = 1).
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 nonfaulting 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).

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 13-4 on page 190.

> **Note** A fast_{instruction,data}_access_MMU_miss or *real_{instruction,data}_translation_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.

13.9.7 I-/D-MMU Synchronous Fault Address Registers (SFAR)

13.9.7.1 I-MMU Fault Address

There is no I-MMU Synchronous Fault Address register. Instead, software must read the TPC register appropriately as described here.

For *fast_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 48-bit virtual address, which is sign-extended based on bit va{47} 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 least-significant 48 bits of the virtual address that is out of range. Because the hardware sign-extends a read of the TPC register based on VA{47}, the contents of the TPC register **xor**ed with FFFF 0000 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.

13.9.7.2 D-MMU Synchronous Fault Address

The Data Synchronous Fault Address register contains the virtual memory address associated with the fault recorded in the D-MMU Synchronous Fault Status register. See Appendix E of the *UltraSPARC Architecture 2005* for a full description of this register.

13.9.8   I-/D-TLB Tag Access Registers

In each UltraSPARC T1 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 13-4 on page 190 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 UltraSPARC T1 TLB Tag Access Registers are defined in FIGURE 13-4; the fields are defined below the table.

> **Note** | When PSTATE.am = 1, the most significant 32 bits of the VA captured in this register will be zero.

va	context_id
63 13	12 0

FIGURE 13-4 I/D MMU TLB Tag Access Registers

Bit	Field	Descriptions
63:13	va	The 51-bit virtual page number (VA{63:13}. Note that writes to this field are not checked for out-of-range violation, but sign-extended based on va{47}.
12:0	context_id	The 13-bit context identifier. This field reads as zero when there is no associated context with the access.

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{47}.

13.9.9   Partition Identifier

A partition identifier register is provided to allow multiple operating systems to share the same TLB. See the *UltraSPARC Architecture 2005* for details.

13.9.10   I-/D-TSB PS0/PS1 Pointer and Direct Pointer Registers

These UltraSPARC T1 registers are provided to help the software determine the location of the missing or trapping TTE in the software-maintained TSB. The TSB PS0 and PS1 Pointer registers provide the possible locations of the PS0 and PS1 TTE, respectively, where PS0 and PS1 are selected from the appropriate configuration register based on zero/non-zero context (zero context uses I/DMMU_CONTEXT_ZERO_CONFIG; non-zero context uses I/DMMU_CONTEXT_NONZERO_CONFIG).

The Direct Pointer register is mapped by hardware to either the PS0 or PS1 Pointer register in the case of a *fast_data_access_protection* exception according to the known size of the trapping TTE, based on the PS1 value of the appropriate configuration register based on nucleus/non-nucleus context (nucleus context uses I/DMMU_CONTEXT_ZERO_CONFIG; non-nucleus context uses I/DMMU_CONTEXT_NONZERO_CONFIG). In the case of a PS1 page-sized miss, the Direct Pointer register returns the pointer as if the miss were from the PS1 page size; otherwise, it returns the pointer as if the miss were from the PS0 page size.

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 an STXA instruction), then the Pointer registers values will be updated as well.

The bit that controls selection of PS0 or PS1 address formation for the Direct Pointer register is a state bit in the D-MMU that is updated during a *fast_data_access_protection* exception. It records whether the page that hit in the TLB was an PS1 page or a non-PS1 page, in which case PS0 is assumed.

Note	When UltraSPARC T1 sets the direct pointer state bit, it does not select the PS1 value to compare against the trapping TTE size based on whether the context value of the access is zero or nonzero. Instead, it selects the PS1 value to compare against the trapping TTE size based on whether the access is from a nucleus or non-nucleus ASI. If the access is from a nucleus ASI, the PS1 value from `ASI_DMMU_CONTEXT_ZERO_CONFIG_REG` will be used. If the access is from a primary or secondary context ASI, the PS1 value from `ASI_DMMU_CONTEXT_NONZERO_CONFIG_REG` will be used, even if the primary or secondary context contains a value of zero. If it is possible that software has set the primary or secondary context to a value of zero, the direct pointer register should not be used.

The I-/D-TSB PS0/PS1 Pointer registers are defined in FIGURE 13-5, where va is the full virtual address of the TTE in the TSB ({63:0}, as determined by the MMU hardware. Described in *Hardware Support for TSB Access* on page 187. Note that this field is sign-extended based on VA{47}.

va
63   0

FIGURE 13-5 I-/D-MMU TSB PS0/PS1 Pointer and D-MMU Direct Pointer Register

13.9.11   I-/D-TLB Data-In/Data-Access/Tag-Read Registers

See the *UltraSPARC Architecture 2005* for general information regarding the TLB Tag Access, Data In, Data Access, and Tag Read registers.

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 registers can be written in either sun4u or sun4v TTE format, under control of bit 10 of the VA. Reads of the Data Access register are always done in sun4u format. Refer to the description of the TTE data in *Translation Table Entry (TTE)* on page 182 for a complete description of the sun4u and TTE formats. The real bit of the TLB is under the control of bit 9 of the VA. If this bit is 1, the real bit of the TLB entry is set to 1; otherwise, the real bit of the TLB entry is set to 0.

The hardware supports an autodemap function to handle the case where two strands sharing a TLB try to enter the same translation into the TLB (for example, due to near-simultaneous TLB misses on the same page). A TLB replacement that attempts

to add an already existing translation will cause the existing translation to be removed from the TLB. Note that this autodemapping of existing translations does not require that the pages be the same size. For example, an insertion of a 8-Kbyte page that sits inside the virtual address range of a 64-Kbyte page will cause the 64-Kbyte page to be autodemapped. Likewise, an insertion of a 4-Mbyte page that overlaps the virtual address of one or more 64-Kbyte pages will cause the overlapping 64-Kbyte pages to be autodemapped. Note that the PIDs and real bits on the pages must match for autodemap to take place. If the real bit is 0, the context IDs must match as well.

Notes | (1) Autodemap will demap locked pages. It is up to software to make sure that a locked TLB entry does not get autodemapped (or only gets autodemapped by an identical locked TLB entry).

(2) If a TLB replacement is attempted using a reserved page size value, a *data_access_exception* trap will be taken.

(3) If a TLB replacement is attempted with the valid bit (v) equal to 0, the MMU will treat that the same as if the valid bit was 1 for purposes of allocating and overwriting a TLB entry and autodemapping matching pages, and the entry will be written into the TLB with the v bit set to 0.

(4) If a TLB Data-In replacement is attempted with all TLB entries locked and valid, the last TLB entry (entry 63) is replaced.

The format of the TLB Data In register virtual address is shown in FIGURE 13-6.

—	sun4v	real	—	000
63	11 10	9 8	3 2	0

FIGURE 13-6 MMU TLB Data In Virtual Address Format

Bit	Field	Description
10	sun4v	The TTE being written is in sun4v format if 1, sun4u format if 0.
9	real	Written to the real bit of the TLB entry

The format of the TLB Data Access register virtual address is shown in FIGURE 13-7.

—	sun4v	real	TLB_entry	000
63	11 10	9 8	3 2	0

FIGURE 13-7 MMU TLB Data Access Virtual Address Format

Bit	Field	Description
10	sun4v	The TTE being written is in sun4v format if 1, sun4u format if 0.
9	real	Written to the real bit of the TLB entry
8:3	tlb_entry	The TLB Entry number to be accessed, in the range 0..63.

The write data format for TLB Data In and TLB Data Access Registers is shown in Appendix E of the *UltraSPARC Architecture 2005* specification.

> **Programming Note** The sun4u write data format, also supported by UltraSPARC T1UltraSPARC T1, is shown in TABLE 13-1 on page 183 and TABLE 13-2 on page 184.. Reads of the TLB Data Access Register always return data in the sun4u format.

The format of the Tag Read register virtual address is shown in FIGURE 13-8, where **TLB Entry** is the number to be accessed, in the range 0..63.

—	TLB Entry	000
63 9	8 3	2 0

FIGURE 13-8 MMU Tag Read Virtual Address Format

The format for the Tag Read register is shown in FIGURE 13-9; the bits are described in TABLE 13-21.

pid	real	parity	mux	va	context_id
63 61	60	59	58 56	55 13	12 0

FIGURE 13-9 I-/D-MMU TLB Tag Read Registers

TABLE 13-17 Tag Read Register Bit Description

BIt	Field	Description
63:61	pid	The 3-bit partition identifier.
60	real	If set, identifies an RA-to-PA translation instead of a VA-to-PA.
59	parity	Parity for the tag entry. Parity is generated across pid, real, mux, va{47:13}, context, and a copy of the l (locked) bit from the data entry.
58:56	mux	Multiplexor encodings of size: 000_2 for 256-Mbyte page, 100 for 4-Mbyte page, 110 for 64-Kbyte page, and 111 for 8-Kbyte page.
56:13	va	The least significant 43 bits of the 51-bit virtual page number (va{63:13}). 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{21:13}, and va{27:13} for 64-Kbyte, 4-Mbyte and 256-Mbyte pages, respectively. **Note:** va{55:48} are sign-extended based on va{47}.
12:0	context_id	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.

Programming Note	Stores to the Data In register are not guaranteed to replace the previous TLB entry causing a fault.

Compatibility Note	UltraSPARC T1 has an autodemap feature which will demap all matching entries before placing an entry in the TLB. This allows software to change an entry's attribute bits by simply storing the new entry into the TLB (although the new entry will not necessarily be placed in the same TLB entry as the old entry). In prior UltraSPARC systems, software needed to explicitly demap the old entry before writing the new entry; otherwise, a multiple match error condition could 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 and generate a *data_access_exception* trap.

13.10 I/D-MMU Demap

The *UltraSPARC Architecture 2005* MMU chapter provides a general description of the Demap operation. This section provides details on the Demap implementation in UltraSPARC T1.

UltraSPARC T1 provides three types of Demap operation: Demap Page, Demap Context, and Demap All. All demap operations only demap pages whose PID matches the PID specified in the Partition Identifier register.

Demap is initiated by a STXA instruction with ASI = 57_{16} for I-MMU demap or $5F_{16}$ for D-MMU demap. It removes TLB entries from an on-chip TLB. FIGURE 13-10 shows the Demap format; TABLE 13-22 defines the fields.

FIGURE 13-10 MMU Demap Operation Format

TABLE 13-18 MMU Demap Operation Bits

Bit	Field	Description
63:13	page_no	The virtual page number (VA{63:12}) of the TTE to be removed from the TLB. This field is not used by the MMU for the Demap Context or Demap All operations. **Note:** The virtual address for demap is *not* checked for out-of-range violations; instead, VA{63:48} is ignored.
9	r	Valid for Demap Page only. Selects between demapping a real translation ($r = 1$) or virtual translation ($r = 0$).
7:6	type	The type of demap operation, as described below.

type Field	Demap Operation
00_2	Demap Page
01_2	Demap Context
10_2	Demap All
11_2	*Reserved*

Use of type = 11_2 in a Demap operation will cause undefined behavior.

Bit	Field	Description
5:4	context	Context register selection, as described below. Use of context = 11_2 causes the demap to be ignored for Demap Page and Demap Context, but is a valid value for a Demap All operation.

context	Context Used in Demap
00_2	Primary
01_2	Secondary
10_2	Nucleus
11_2	*Reserved*

For an IMMU Demap Context operation, context = 01_2 (referencing the Secondary Context 0 register) is not supported; using context = 01_2 in that situation causes the demap operation to be ignored.
Address bits 12:10 and 8 are ignored by hardware.

A demap operation does not invalidate the TSB in memory. It is the responsibility of the software to modify the appropriate TTEs in the TSB before initiating any demap operation.

The demap operation only affects TLB contents; it does not modify any other virtual processor registers.

13.10.1   I-/D-Demap Page (Type = 0)

Demap Page removes the TTE (from the specified TLB) matching the specified virtual page number, "real" bit (r), partition identifier register, and context register.

Virtual page offset bits 15:13, 21:13, and 27:13, for 64-Kbyte, 4-Mbyte, and 256-Mbyte 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.

13.10.2   I-/D-Demap Context (Type = 1)

Demap Context removes all TTEs having the specified context, r = 0, a real bit of 0, and matching the partition identifier register from the specified TLB.

13.10.3   I-/D-Demap All (Type = 2)

Demap Context removes all TTEs with their locked bit clear and matching the partition identifier register from the specified TLB.

If a real page is being demapped, the context identifier is ignored when determining a match.

13.11   I-/D-TLB Invalidate All

I-/DTLB Invalidate All are write-only registers that invalidate the entire I or D TLB (regardless of any TLB state like lock bits). The data written to these registers is ignored. Reads of these registers return a *data_access_exception* trap.

13.12 TLB Hardware

13.12.1 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 *I-/D-TLB Data-In/Data-Access/Tag-Read Registers* on page 215.)
- **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 *TLB Replacement Policy*, below.
- **No operation.** The TLB performs no operation.

13.12.2 TLB Replacement Policy

UltraSPARC T1 uses a 1-bit LRU scheme. Each TLB entry has an associated valid, used, and lock bit. Used bits are set on each TLB translation and also on the initial TLB write of an entry. When setting the used bit for a translation or TLB write would result in all used bits being set, the used bits for all TLB entries that are unlocked are cleared instead.

Implementation Note	The used bits are updated on all TLB translations, including translations for PREFETCH instructions.

On an automatic write to the TLB initiated through an ASI store to the TLB Data In register, the TLB replaces the first invalid or unused entry. Arbitrary entries may have their lock bit set; however, if all entries have their lock bit set, a Data In replacement will replace the final TLB entry (63).

Implementation Dependencies

14.1   SPARC V9 General Information

14.1.1   Level-2 Compliance (Impl. Dep. #1)

UltraSPARC T1 is designed to meet Level-2 SPARC V9 compliance. It does the following:

■ Correctly interprets all nonprivileged operations, and

■ Correctly interprets all privileged elements of the architecture.

> **Note** | System emulation routines (for example, for quad-precision floating-point operations) shipped with UltraSPARC T1 also must be Level-2 compliant.

14.1.2   Unimplemented Opcodes, ASIs, and ILLTRAP

SPARC V9 unimplemented instructions, *reserved* instructions, ILLTRAP opcodes, and instructions with invalid values in *reserved* fields (other than *reserved* FPops and the *reserved* field in the Tcc instruction) encountered during execution cause an *illegal_instruction* trap. Reserved FPops 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 (Imp. Dep. #37, 38, 39, 40, 101, 114, 115)

UltraSPARC T1 supports privileged and six total trap levels; that is, *MAXPTL* = 2 and *MAXTL* = 6. Normal execution is at TL = 0. 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 strand normally executes at trap level 0 (execute_state, TL = 0). In SPARC V9, a trap makes the CPU enter the next higher trap level, which is a 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, NPC, 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 *Machine State after Reset and in RED_State* on page 104.

> **Note** | The RED_state trap vector address (*RSTVADDR*) is 256 Mbytes below the top of the virtual address space; that is, at virtual address FFFF FFFF F000 0000$_{16}$, which is passed through to physical address FF F000 0000$_{16}$ in RED_state.

14.1.4 Trap Handling (Imp. Dep. #16, 32, 33, 35, 36, 44)

UltraSPARC T1 supports precise trap handling for all operations except for disrupting traps from hardware failures and interrupts. UltraSPARC T1 implements precise traps, interrupts, and exceptions for all instructions, including long latency floating-point operations. Six traps levels are supported (*MAXTL* = 6), which allows graceful recovery from faults. The first three trap levels (0 through 2) are provided for application and operating system use. The remaining three levels (3 through 5) are provided for hyperprivileged and RED_state use.

UltraSPARC T1 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. Four sets of global registers are provided (*MAXPGL* = 2 and *MAXGL* = 3), for use at TL = 0, TL = 1 TL = 2, and TL = 3-5.

This further increases OS performance, providing fast trap execution by avoiding the need to save and restore registers while processing exceptions.

All traps supported in UltraSPARC T1 are listed in the "Traps" chapter of this document.

14.1.5 SIR Support (Impl. Dep. #116)

UltraSPARC T1 initiates a software-initiated reset (SIR) by executing a SIR instruction while in privileged or hyperprivileged mode. When in nonprivileged mode, SIR behaves as a NOP. See also *Watchdog Reset (WDR) and error_state* on page 99.

14.1.6 Population Count Instruction (POPC)

The population count instruction, POPC, generates an *illegal_instruction* exception and is emulated in software rather that being executed in hardware.

14.1.7 Secure Software

To establish an enhanced security environment, it may be necessary to initialize certain strand states between contexts. Examples of such states are the contents of integer and floating-point register files, condition codes, and state registers. See also *Clean Window Handling (Impl. Dep. #102)*.

14.1.8 Address Masking (Impl. Dep. #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. Traps also transmit zero in the high-order 32-bits of the NPC to the TNPC. Branch target addresses sent to the NPC and the updating of NPC with NPC+4 for a non-control-transferring instruction do not zero the high-order 32-bits. Restoration of PC and NPC from TPC and TNPC on a DONE or RETRY instruction do not mask the high-order 32-bits.

> **Note** | When PSTATE.am = 1, address masking applies to all VAs, even those that immediately do a VA-to-RA bypass or a VA-to-PA bypass. This implies that with PSTATE.am = 1, RA{63:32} will be zeros after a VA-to-RA bypass, and PA{39:32> will be zeros after a VA-to-PA bypass.

14.2 SPARC V9 Integer Operations

14.2.1 Integer Register File and Window Control Registers (Impl. Dep. #2)

UltraSPARC T1 implements an eight-window 64-bit integer register file; that is, *N_REG_WINDOWS* = 8. UltraSPARC T1 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 most-significant two bits of these registers read as zero.

14.2.2 SAVE Instruction

Upon a SAVE instruction, UltraSPARC T1 initializes the values of the local registers in the new window to the same values as the local registers in the old window and initializes the values of the *out* registers in the new window to the same values as the *in* registers in the old window (that is, the new window matches the old window with the *ins* and *outs* swapped). Since this implies that they contain values from the executing process, V9 compliance is maintained. In this sense, the behavior of the SAVE instruction on UltraSPARC T1 differs from most other SPARC V9 implementations.[1]

14.2.3 Clean Window Handling (Impl. Dep. #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. Note that the behavior on a *clean_window* trap for UltraSPARC T1 is the same as for a SAVE instruction, namely, the *local* registers for the new window remain the same as the *local* registers from the old window, while the *out* registers in the new window contain the contents of the *in* registers from the old window. Thus, while UltraSPARC T1 generates a *clean_window* trap, the new window is automatically cleaned by hardware. System software only needs to increment CLEANWIN before returning to the requesting context.

[1.] Most SPARC V9 processors do not initialize the *local* and *out* registers on a save instruction; instead, the values in the *local* and *out* registers are those left there from the last time the window was used.

14.2.4   Integer Multiply and Divide

Integer multiplications (MULScc, SMUL{cc}, MULX) and divisions (SDIV{cc}, UDIV{cc}, UDIVX) are executed directly in hardware.

14.2.5   MULScc

SPARC V9 does not define the value of xcc and R[rd]{63:32} for MULScc. UltraSPARC T1 sets xcc and rd based on the results of adding either (32 copies of R[rs1]{63}:: CCR.icc.n **xor** CCR.icc.v, R[rs1]{31:1}) or 0 (depending on Y{0}) to either R[rs2]{63:0} or the immediate operand.

14.2.6   Hyperprivileged Version Register (Impl. Dep. #2, 13, 101, 104)

Consult the product data sheet for the contents of the Version register for a specific UltraSPARC T1 implementation. The format of the Hyperprivileged Version register is described in *Hyperprivileged Version Register (HVER)* on page 22.

14.3   SPARC V9 Floating-Point Operations

14.3.1   Subnormal Operands and Results: Nonstandard Operation

UltraSPARC T1 handles all cases of subnormal operands or results directly in hardware.

Because there is no trapping on subnormal operands, UltraSPARC T1 does not support the nonstandard result option of the SPARC V9 architecture, and the FSR.ns bit ignores any value written to it and always returns zero on a read.

14.3.2 Overflow, Underflow, and Inexact Traps (Impl. Dep. #3, 55)

UltraSPARC T1 implements precise floating-point exception handling. Underflow is detected before rounding.

| Note | Major performance degradation may be observed while running with the inexact exception enabled. |

14.3.3 Quad-Precision Floating-Point Operations (Impl. Dep. #3)

All quad-precision floating-point instructions, listed in TABLE 14-1, cause an *fp_exception_other* (with FSR.ftt = 3, unimplemented_FPop) trap. These operations are emulated in system software.

TABLE 14-1 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 Square Root

The three floating-point square root instructions: FSQRTS, FSQRTD, FSQRTQ are unimplemented. Execution of any of these instructions results in an *fp_exception_other* exception, with FSR.ftt= unimplemented_FPop.

14.3.5　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 issued. 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 due to a trap. This includes the case where the floating-point instruction itself takes a *fp_disabled* trap.

14.3.6　Floating-Point State Register (FSR) (Impl. Dep. #13, 19, 22, 23, 24)

UltraSPARC T1 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 11-11 on page 106. TABLE 14-2 defines the register bits.

TABLE 14-2　Floating-Point Status Register Format

Bits	Field	RW	Description
63:38	—	R	*Reserved*
37:36	fcc3	RW	Floating-point condition code (set 3). One of 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.
35:34	fcc2	RW	Floating-point condition code (set 2). See fcc3.
33:32	fcc1	RW	Floating-point condition code (set 1). See fcc3.

TABLE 14-2   Floating-Point Status Register Format *(Continued)*

Bits	Field	RW	Description
31:30	rd	RW	IEEE Std. 754-1985 rounding direction. Rounding modes are shown below.

rd	Round Toward
0	Nearest (even if tie)
1	0
2	$+\infty$
3	$-\infty$

Bits	Field	RW	Description
29:28	—	R	*Reserved*
27:23	tem	RW	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.
22	ns	R	Nonstandard floating-point results. Always 0: UltraSPARC T1 produces IEEE-754 compatible results.
21:20	—	R	*Reserved*
19:17	ver	R	FPU version number. Identifies a particular implementation of the UltraSPARC T1 FPU architecture.
16:14	ftt	R	Floating-point trap type. 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. Trap types are listed in TABLE 14-4, below.
13:	qne	R	Floating-point deferred-trap queue (FQ) not empty. Not used, because UltraSPARC T1 implements precise floating-point exceptions.
12	—	R	*Reserved*
11:10	fcc0	RW	Floating-point condition code (set 0). See fcc3. Note: fcc0 is the same as fcc in SPARC V8.
9:5	aexc	RW	5-bit accrued exception field. Accumulates IEEE 754 exceptions while floating-point exception traps are disabled (that is, FSR.tem = 0).
4:0	cexc	RW	5-bit current exception field indicates the most recently generated IEEE 754 exceptions.

Note | fcc0 is the same as the fcc in SPARC V8.

TABLE 14-3   Floating-Point Rounding Modes

rd	Round Toward
0	Nearest (even if tie)

TABLE 14-3 Floating-Point Rounding Modes

rd	Round Toward
1	0
2	$+\infty$
3	$-\infty$

TABLE 14-4 Floating-Point Trap Type Values

ftt	Floating-Point Trap Type	Trap Signaled
0	None	—
1	IEEE_754_exception	*fp_exception_ieee_754*
2	unfinished_FPop	—
3	unimplemented_FPop	*fp_exception_other*
4	sequence_error	—
5	hardware_error	—
6	invalid_fp_register	—
7	reserved	—

Notes | (1) UltraSPARC T1 neither detects nor generates the unfinished_FPop, sequence_error, hardware_error or invalid_fp_register trap types directly in hardware.

(2) UltraSPARC T1 does not contain an FQ. An attempt to read the FQ with a RDPR instruction causes an *illegal_instruction* trap.

14.4   SPARC V9 Memory-Related Operations

14.4.1   Load/Store Alternate Address Space (Impl. Dep. #5, 29, 30)

Supported ASI accesses are listed in *Alternate Address Spaces* on page 69.

14.4.2 Read/Write ASR (Impl. Dep. #6, 7, 8, 9, 47, 48)

Supported ASRs are discussed in *Ancillary State Registers (ASRs)* on page 12.

14.4.3 MMU Implementation (Impl. Dep. #41)

UltraSPARC T1 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 13, *Memory Management Unit* for more details.

14.4.4 FLUSH and Self-Modifying Code (Impl. Dep. #122)

FLUSH is needed to synchronize code and data spaces after code space is modified during program execution. FLUSH is described in *Supported Memory Models* on page 58. On UltraSPARC T1, the FLUSH effective address is ignored, and as a result, FLUSH can not cause a *data_access_exception* or a *fast_data_access_MMU_miss* trap.

| **Note** | SPARC V9 specifies that the FLUSH instruction has no latency on the issuing strand. 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 multiprocessing system when the memory model is not TSO. When a MEMBAR #StoreStore, FLUSH sequence is performed, UltraSPARC T1 guarantees that earlier code modifications will be visible across the whole system. |

14.4.5 PREFETCH{A} (Impl. Dep. #103, 117)

For UltraSPARC T1, PREFETCH{A} instructions follow TABLE 14-5 based on the fcn value. All prefetches in UltraSPARC T1 are weak (on an MMU miss or when the MMU is bypassed the prefetch is dropped). The only trap that a prefetch can generate on UltraSPARC T1 is *illegal_instruction* (for fcn = 5_{16}–F_{16}).

TABLE 14-5 PREFETCH{A} Variants

fcn	Prefetch Function	Action
0_{16}	Weak prefetch for several reads	Weak prefetch into Level 2 cache
1_{16}	Weak prefetch for one read	
2_{16}	Weak prefetch for several writes	
3_{16}	Weak prefetch for one write	

TABLE 14-5 PREFETCH{A} Variants

fcn	Prefetch Function	Action
4_{16}	Prefetch Page	No operation
5_{16}–F_{16}	-	*Illegal_instruction* trap.
10_{16}	Invalidate read-once prefetch	Weak prefetch into Level 2 cache
11_{16}	Prefetch for read to nearest unified cache	Weak prefetch into Level 2 cache
12_{16}–13_{16}	Strong prefetches	Weak prefetch into Level 2 cache
14_{16}	Strong prefetch for several reads	Weak prefetch into Level 2 cache
15_{16}	Strong prefetch for one read	
16_{16}	Strong prefetch for several writes	
17_{16}	Strong prefetch for one write	
18_{16}-$1F_{16}$	—	No operation

PREFETCHA is legal for all implemented ASIs in UltraSPARC T1 and will prefetch into the Level 2 cache from memory using the context listed in TABLE 14-6. Prefetching is done regardless of privilege level (for example, user mode can use ASI 10_{16} to prefetch into the L2 cache), as long as the virtual address presented has a valid TLB entry.

TABLE 14-6 PREFETCH{A} ASIs

Context	ASIs (hexadecimal)
Primary	10, 16, 18, 1E, 22, 2A, 80, 82, 88, 8A, C0, C2, C4, C8, CA, CC, D0, D2, D8, DA, E0, E2, EA, F0, F8
Secondary	11, 17, 19, 1F, 23, 2B, 81, 83, 89, 8B, C1, C3, C5, C9, CB, CD, D1, D3, D9, DB, E1, E3, EB, F1, F9
Nucleus	04, 0C, 14, 15, 1C, 1D, 20, 21, 24, 25, 26, 27, 2C, 2E, 2F, 31, 32, 33, 35, 36, 37, 39, 3A, 3B, 3D, 3E, 3F, 40, 42, 43, 44, 45, 46, 47, 4B, 4C, 4D, 4F, 50, 51, 52, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 5E, 5F, 60, 66, 67, 72, 73, 74

Implementation Note | Although it would have been desirable to treat PREFETCHA to restricted ASIs by underprivileged code as NOPs, PREFETCH only moves data between main memory and the L2 cache, so UltraSPARC T1's implementation causes no security issues.

14.4.6   Instruction Prefetch

UltraSPARC T1 does not implement an instruction prefetch. No prefetching is performed from the effective address of the BPN instruction.

14.4.7 Nonfaulting Load and MMU Disable (Impl. Dep. #117)

When the data MMU is disabled, accesses are assumed to be noncacheable (TTE.cp = 0) and with side effect (TTE.e = 1). Nonfaulting 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).

14.4.8 LDTW/STTW Handling (Impl. Dep. #107, 108)

LDTW and STTW instructions are directly executed in hardware.

> **Note** | LDTW/STTW were deprecated in SPARC V9. In UltraSPARC
> T1, it is more efficient to use LDX/STX for accessing 64-bit data.
> LDTW/STTW take longer to execute than two 32-/64-bit loads/
> stores.

14.4.9 Floating-Point *mem_address_not_aligned* (Impl. Dep. #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.

14.4.10 Supported Memory Models (Impl. Dep. #113, 121)

UltraSPARC T1 supports only the TSO memory model, although certain specific operations such as block loads and stores operate under the RMO memory model. See *Supported Memory Models* on page 58.

14.4.11 I/O Operations (Impl. Dep. #118, 123)

I/O spaces and their accesses are specified in *Physical Address Spaces* on page 64.

14.4.12 Implicit ASI when TL > 0 (Impl. Dep. #124)

UltraSPARC T1 matches all UltraSPARC Architecture implementations, and makes the implicit ASI for instruction fetching `ASI_NUCLEUS` when TL > 0, while the implicit ASI for loads and stores when TL > 0 is `ASI_NUCLEUS` if PSTATE.cle = 0 or `ASI_NUCLEUS_LITTLE` if PSTATE.cle = 1.

Compatibility Note	With an implicit ASI for instruction fetching of `ASI_NUCLEUS`, if software was to set the strand in a state where PSTATE.priv = 0 but TL > 0, an instruction fetch will generate an *instruction_access_exception,* because user-level code is accessing `ASI_NUCLEUS`. UltraSPARC I/II overrides this *instruction_access_exception* and allows instruction fetching when PSTATE.priv = 0 and TL > 0. UltraSPARC T1 is compatible with UltraSPARC I/II and does the same override of *instruction_access_exception* when PSTATE.priv = 0 and TL > 0.

14.5 Non-SPARC V9 Extensions

14.5.1 Cache Subsystem

UltraSPARC T1 contains one or more levels of cache. The cache subsystem architecture is described in Appendix F, *Caches and Cache Coherency*.

14.5.2 Memory Management Unit

UltraSPARC T1 implements a multilevel memory management scheme. The MMU architecture is described in Chapter 13, *Memory Management Unit*.

14.5.3 Error Handling

UltraSPARC T1 implements a set of programmer-visible error and exception registers. These registers and their usage are described in Chapter 12, *Error Handling*.

14.5.4 Block Memory Operations

UltraSPARC T1 supports 64-byte block memory operations utilizing a block of eight double-precision floating point registers as a temporary buffer. See *Block Load and Store Instructions* on page 27.

14.5.5 Partial Stores

UltraSPARC T1 does not support 8-/16-/32-bit partial stores to memory.

14.5.6 Short Floating-Point Loads and Stores

UltraSPARC T1 does not supports 8-/16-bit loads and stores to the floating-point registers.

14.5.7 Interrupt Vector Handling

CPUs and I/O devices can interrupt a selected CPU by assembling and sending an interrupt packet. 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 7, *Interrupt Handling*.

14.5.8 Power-Down Support

UltraSPARC T1 supports the ability to power down virtual processors and I/O devices to reduce power requirements during idle periods.

14.5.9 UltraSPARC T1 Instruction Set Extensions (Impl. Dep. #106)

The UltraSPARC T1 CPU supports a subset of the VIS 1.0 and 2.0 instructions; see *UltraSPARC Architecture 2005 Instructions Not Directly Implemented by UltraSPARC T1 Hardware* on page 22.

Unimplemented IMPDEP1 and IMPDEP2 opcodes encountered during execution cause an *illegal_instruction* trap.

14.5.10 Performance Instrumentation

UltraSPARC T1 performance instrumentation is described in *Performance Control Register* on page 81 and *SPARC Performance Instrumentation Counter* on page 83.

Configuration and Diagnostics Support

15.1   `ASI_LSU_CONTROL_REG` Register

Each hardware strand has a hyperprivileged `ASI_LSU_CONTROL_REG` register at ASI 45_{16}, VA{63:0} = 0 which contains fields that control several memory-related hardware functions. The format of the register is shown in TABLE 15-1.

TABLE 15-1   LSU Control Register – `ASI_LSU_CONTROL_REG` (ASI 45_{16}, VA 0_{16})

Bit	Field	Initial Value	R/W	Description
63:41	—	0	R	*Reserved*
40:33	pm	0	R	*Reserved* (Normally Physical Address Data Watchpoint Byte Mask.)
32:25	vm	0	R/W	Virtual Address Data Watchpoint Byte Mask. The `ASI_DMMU_VA_WATCHPOINT` register described on page 239 contains the virtual address of a 64-bit word to be watched. The 8-bit vm controls which byte(s) within the 64-bit word should be watched. If all 8 bits are cleared, the virtual watchpoint is disabled. If the watchpoint is enabled and a data reference overlaps any of the watched bytes in the watchpoint mask, a *VA_watchpoint* trap is generated. Examples are shown below.

Watchpoint Mask	Address of Bytes Watched 7654 3210
00_{16}	Watchpoint disabled
01_{16}	0000 0001
32_{16}	0011 0010
FF_{16}	1111 1111

Bit	Field	Initial Value	R/W	Description
24	pr	0	R	*Reserved* (Physical Address Data Watchpoint Read Enable.)
23	pw	0	R	*Reserved* (Physical Address Data Watchpoint Write Enable.)

TABLE 15-1 LSU Control Register – `ASI_LSU_CONTROL_REG` (ASI 45_{16}, VA 0_{16}) *(Continued)*

Bit	Field	Initial Value	R/W	Description
22	vr	0	R/W	Virtual Address Data Watchpoint Read/Write Enable. If vr/vw is set, a read/write that matches the address in
21	vw	0	R/W	`ASI_DMMU_VA_WATCHPOINT` and the vm byte masks causes a *VA_watchpoint* trap. Both vr and vw may be set to place a watchpoint for either a read or write access. Atomic operations are considered both a read and a write, and watchpoints for atomics are enabled if either vr or vw or both are set.
20:4	—	0	R	*Reserved*
3	dm	0	R/W	DMMU Enable. If cleared, the DMMU is either bypassed, always does a Virtual-to-Real bypass (when HPSTATE.hpriv = 0), or behaves normally (when HPSTATE.hpriv = 1). See *Translation* on page 196 for more details.
2	im	0	R/W	IMMU Enable. If cleared, the IMMU does a Virtual-to-Real bypass. See *Translation* on page 196 for more details.
1	dc	0	R/W	D-cache Enable. If cleared, the primary data cache does not allocate a line on a miss. See **Note**.
0	ic	0	R/W	I-cache Enable. If cleared, the primary instruction cache does not allocate a line on a miss. See **Note**.

Note: The D-cache and I-cache are still kept coherent by UltraSPARC T1 when the dc and ic bits are set to 0. This includes updating the D-cache when stores from a strand hit in the D-cache.

Nonprivileged access to this register causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap.

15.1.1 Watchpoint Support

UltraSPARC T1 includes support for generating an *instruction_breakpoint* trap for an instruction or class of instructions, as described in *ASI_INST_MASK_REG Register*. UltraSPARC T1 also supports VA watchpoints, but does not support PA watchpoints. The VA watchpoint support is discussed in *ASI_DMMU_VA_WATCHPOINT Register*. Also see the `ASI_LSU_CONTROL_REG` definition in *ASI_LSU_CONTROL_REG Register*, which also includes controls for the VA watchpoint support.

15.1.2 `ASI_INST_MASK_REG` Register

Each physical processor core has a hyperprivileged `ASI_INST_MASK_REG` register at ASI 42_{16}, VA{63:0} = 8_{16}. This register is used to disable execution of a particular instruction or class of instructions for diagnostic or debug purposes, under the control of HPSTATE.ibe. If HPSTATE.ibe = 1 and any of the enable fields are set to 1,

execution of any instruction which matches all the enabled bits in the inst field of this register results in an *instruction_breakpoint* trap. Nonprivileged access to this register causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap.

TABLE 15-2 defines the format of the ASI_INST_MASK_REG register.

TABLE 15-2 SPARC Instruction Mask Register – ASI_INST_MASK_REG (ASI 42_{16}, VA 8_{16})

Bit	Field	Initial Value	R/W	Description
63:39	—	0	R	*Reserved*
38	enb31_30	0	R/W	Enable matching on inst 31:30
37	enb29_25	0	R/W	Enable matching on inst 29:25
36	enb24_19	0	R/W	Enable matching on inst 24:19
35	enb18_14	0	R/W	Enable matching on inst 18:14
34	enb13	0	R/W	Enable matching on inst 13
33	enb12_5	0	R/W	Enable matching on inst 12:5
32	enb4_0	0	R/W	Enable matching on inst 4:0
31:0	inst	0	R/W	Instruction pattern to be trapped

Programming Note | While there is a single instruction mask register per physical processor core, each strand is under the control of its own HPSTATE.ibe bit. All HPSTATE.ibe bits for the strands in a physical processor core must be set to have the physical processor core fully trap on the instruction pattern.

15.1.3 ASI_DMMU_VA_WATCHPOINT Register

Each strand has a hyperprivileged ASI_DMMU_VA_WATCHPOINT register at ASI 58_{16} VA{63:0} = 38_{16} that is used for controlling the address (or address range if bytes are masked) for a VA data watchpoint. The format of the register is shown in TABLE 15-3.

TABLE 15-3 DMMU Watchpoint – ASI_DMMU_VA_WATCHPOINT (ASI 58_{16}, VA 38_{16})

Bit	Field	Initial Value	R/W	Description
63:48	—	0	R	*Reserved*
47:3	va	X	R/W	VA watchpoint address
2:0	—	0	R	Bits 2:0 of the VA watchpoint address are all zeros; watchpointing is done for 8-byte memory locations.

Nonprivileged access to this register causes a privileged_action trap. Supervisor access causes a *data_access_exception* trap.

Notes	(1) *VA_watchpoint* traps are generated for ASIs 04_{16}, $0C_{16}$, 10_{16}, 11_{16}, 16_{16}–19_{16}, $1E_{16}$–$1F_{16}$, 22_{16}–24_{16}, 27_{16}, $2A_{16}$–$2C_{16}$, $2F_{16}$, 80_{16}–83_{16}, 88_{16}–$8B_{16}$, $C0_{16}$–$C5_{16}$, $C8_{16}$–CD_{16}, $D_{16}0$–$D3_{16}$, $D8_{16}$–DB_{16}, $E0_{16}$–$E3_{16}$, EA_{16}, EB_{16}, $F0_{16}$, $F1_{16}$, $F8_{16}$, and $F9_{16}$.

(2) The *VA_watchpoint* trap is never generated while executing in hyperprivileged mode. This implies that the AS_IF_USER ASIs will not generate a *VA_watchpoint* trap when accessed in hyperprivileged mode, even though a *VA_watchpoint* trap might be generated when accessed in user mode.

(3) For quadword accesses, VA watchpoint checking is only done on the lower 8 bytes of the access. This implies that a *VA_watchpoint* trap will only be generated for a quadword load if PA{3} is set to zero.

(4) No *VA_watchpoint* trap is generated by UltraSPARC T1 for real-to-physical translations (RA → PA in TABLE 13-12 and TABLE 13-14 in *Translation* on page 196).

(5) When PSTATE.am = 1, only bits 31:3 of the VA and the watchpoint address (in ASI_DMMU_VA_WATCHPOINT) are compared; bits 47:32 of both are ignored.

Implementation Note	The VA watchpoint comparison is performed only on VA{47:3}, and does not mask ASI_DMMU_VA_WATCHPOINT{47:32} when PSTATE.am = 1. This implies that out of range accesses to the VA hole will generate a VA_watchpoint trap if bits {47:3} and the byte mask match.

15.2 L1 I-Cache Diagnostic Access

15.2.1 ASI_ICACHE_INSTR Register

The strands share a hyperprivileged ASI_ICACHE_INSTR register at ASI: 66_{16}, VA{63:0} = 0_{16}–$3FF8_{16}$ that is used for diagnostic access to the L1 instruction cache data array. Diagnostic access is done through 64-bit read/writes that access a 32 bit data subblock along with the corresponding parity bit and switch bit (the switch bit is a pre-decode of the instruction). Parity is calculated over the 32-bit instruction plus the switch bit, and even parity is used (the parity bit is 1 if the number of ones

in the data word is odd, making the total number of ones in the codeword even). Note that I-cache coherence is lost if a diagnostic store is done to this ASI, so after doing one or more stores software must ensure that the entire cache is flushed.

The format for addressing the entire L1 instruction cache is shown in TABLE 15-4. Nonprivileged access to this ASI causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap. Access with loads or stores that do not request 64 bits causes a *data_access_exception* trap. Loads from this ASI do *not* receive an error if the parity in the data is bad.

Programming Note | The reserved fields in the address are ignored and may contain any value.

TABLE 15-4 Format for L1 Instruction Cache Data Diagnostic Addressing

Bit	Field	Description
63:14	—	*Reserved*
17:16	way	Selects way in cache set.
15:13	—	*Reserved*
12:6	set	Selects cache set. Corresponds to PA{11:5}.
5:3	word	Selects 32-bit word in 32-byte cache line. Corresponds to PA{4:2}.
2:0	—	All zero for 64-bit access.

Programming Note | Since 32-bit data quantities are accessed using 64-bit access, the physical address bits that would normally index the set and word cannot be used directly to compute the ASI_ICACHE_INSTR virtual address (that is, instead of using PA{11:5} to index the set and PA{4:2} to index the word, ASI_ICACHE_INSTR uses VA{12:6} to index the set and VA{5:3} to index the word.

The data format is shown in TABLE 15-5. On a write, the value written to the switch bit is ignored (the switch bit is computed by the hardware), and the parity bit is used as a parity error enable. If the parity bit is set to 1, the computed parity for the instruction is inverted.

TABLE 15-5 L1 Instruction Cache Diagnostic Data – ASI_ICACHE_INSTR (ASI 66_{16}, VA 0_{16}– 3FFF8$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:34	—	X	R	*Reserved*
33	switch	X	R	Switch bit for instruction.
32	parity	X	RW	Parity for instruction on read, parity error enable on write.
31:0	instr	X	RW	Instruction.

Note that anytime software does an ASI write to either the I-cache instruction array or I-cache tag/valid bit array, UltraSPARC T1 loses coherency. Even if after all the diagnostic accesses are done, software invalidate all the lines of the I-cache, UltraSPARC T1 still cannot guarantee coherency, because the directory will be out of sync with the actual cache. Even if the I-cache is turned off before a diagnostic access, we may lose coherency when it is turned back on after the accesses are complete.

The way to restore coherency after a diagnostic access is to (a) perform a power-on reset, (b) insert a parity error to all the modified lines of the I-cache, forcing a refetch and a resync with the directory, or (c) displacement-flush the entire L2 cache, which through inclusion forces a flush of the I-cache and directory.

15.2.2 ASI_ICACHE_TAG Register

The strands share a hyperprivileged ASI_ICACHE_TAG register at ASI 67_{16}, VA{63:0} = 0_{16}– $3FC0_{16}$ that is used for diagnostic access to the L1 instruction cache tag array. Diagnostic access to the L1 instruction cache tag array is done through 64-bit read/writes that access a 28-bit tag along with the corresponding parity bit and valid bit. Note that even parity is used (the parity bit is 1 if the number of ones in the data word is odd, making the total number of ones in the codeword even), and parity is across the tag only (does not include the valid bit). Note that I-cache coherence is lost if a diagnostic store is done to this ASI, so after doing one or more stores software must ensure that the entire cache is flushed. The format for addressing the entire L1 instruction cache is shown in TABLE 15-6.

Nonprivileged access to this ASI causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap. Access with loads or stores that do not request 64 bits causes a *data_access_exception* trap. Loads from this ASI do *not* receive an error if the parity in the tag is bad.

Programming Note	The reserved fields in the address are ignored and can contain any value.

TABLE 15-6 Format for L1 Instruction Cache Tag Diagnostic Addressing

Bit	Field	Description
63:18	—	*Reserved*
17:16	way	Selects way in cache set.
15:13	—	*Reserved*
12:6	set	Selects cache set. Corresponds to PA{11:5}.
5:3	—	*Reserved*

The data format for Instruction diagnostic tag access is shown in TABLE 15-7. On a write, the parity bit is used as a parity error enable. If the parity bit is set to 1, the computed parity for the instruction is inverted.

Notes | (1) The I-cache Tag includes physical address bit 39, even though in normal system operation bit 39 will always be 0 for any cacheable access.

(2) On a write, the value written to reserved bits 31:28 is included in the tag parity calculation, so these bits should always be zero for a ASI_ICACHE_TAG write.

TABLE 15-7 L1 Instruction Diagnostic Tag – ASI_ICACHE_TAG (ASI 67_{16}, VA 0_{16}– $3FFF8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:35	—	0	R	*Reserved*
34	valid	X	RW	Valid bit for tag
33	—	0	R	*Reserved*
32	parity	X	RW	On read, parity bit for tag. On writes, a parity error enable. If 0, parity is computed over bits 31:28 and tag. If 1, parity is computed over bits 31:28 and tag, and then inverted.
31:28	—	0	R	*Reserved*. Read as 0. These bits are included in the parity calculation and should be all 0 when writing.
27:0	tag	X	RW	Cache line tag (PA{39:12})

Programming Notes | (1) Anytime software does an ASI write to either the I-cache instruction array or I-cache tag/valid bit array, UltraSPARC T1 loses coherency. Even if after all the diagnostic accesses are done, software invalidate all the lines of the I-cache, UltraSPARC T1 still cannot guarantee coherency, because the directory will be out of sync with the actual cache. Even if the I-cache is turned off before a diagnostic access, we may lose coherency when it is turned back on after the accesses are complete.

The way to restore coherency after a diagnostic access is to (a) perform a power-on reset, (b) insert a parity error to all the modified lines of the icache, forcing a refetch and a resync with the directory, or (c) displacement-flush the entire L2 cache, which (through inclusion) forces a flush of the I-cache and directory.

15.3   L1 D-Cache Diagnostic Access

15.3.1   `ASI_LSU_DIAG_REG` Register

Each physical processor core has a hyperprivileged `ASI_LSU_DIAG_REG` register at ASI 42_{16}, VA{63:0} = 10_{16}. This register is used to disable associativity in the primary instruction and/or data caches for diagnostic or debug purposes.

Nonprivileged access to this register causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap.

TABLE 15-8 defines the format of the `ASI_LSU_DIAG_REG` register.

TABLE 15-8   LSU Diagnostic Register – `ASI_LSU_DIAG_REG` (ASI 42_{16}, VA 10_{16})

Bit	Field	Initial Value	R/W	Description
63:2	—	0	R	*Reserved*
1	dassocdis	0	R/W	If 1, changes the replacement algorithm in the data cache to use bits 12:11 of the physical address to select the way to place data into the cache instead of the random replacement bits.
0	iassocdis	0	R/W	If 1, disables all ways except one (way three) in the instruction cache.

15.3.2   `ASI_DCACHE_DATA` Register

The strands share a hyperprivileged `ASI_DCACHE_DATA` register at ASI 46_{16}, VA{63:0} = 0_{16}–7FFFF FFFF8$_{16}$ that is used for diagnostic access to the L1 data cache data array. Diagnostic store access to the L1 data cache data array is done through 64-bit writes that update 64 bits of data and can optionally invert one to all of the data parity bits. Note that even parity is used (the parity bit is 1 if the number of ones in the data word is odd, making the total number of ones in the codeword even).

Diagnostic load access to the L1 data cache is done through 64-bit reads that must hit in the cache (that is, the virtual address is treated as physical and must match a tag in the cache). If a diagnostic access is done using a load address that doesn't match a tag in the cache, the result of the operation is undefined. Note that D-cache coherence is lost if a diagnostic store is done to this ASI, so after doing one or more stores software must ensure that the entire cache is flushed. The format for addressing the entire L1 Data cache is shown in TABLE 15-9.

Nonprivileged access to this register causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap. Access with loads or stores that do not request 64 bits causes a *data_access_exception* trap. Loads from this ASI do *not* receive an error if the parity in the data is bad.

TABLE 15-9   Format 1 L1 Data Cache Data Diagnostic Addressing for Stores

Bit	Field	Description
63:21	—	*Reserved*
20:13	perrmask	Mask for parity error (if 1, corresponding computed parity bit is inverted).
12:11	way	Selects way in cache set.
10:4	set	Selects cache set.
3	word	Selects 64-bit word in 16-byte cache line.
2:0	—	All zero for 64-bit access.

TABLE 15-10 shows the format for addressing L1 data cache for loads.

TABLE 15-10   Format 2 L1 Data Cache Data Diagnostic Addressing for Loads

Bit	Field	Description
63:39	—	*Reserved*
38:11	tag	Must match a valid tag in the cache.
10:4	set	Selects cache set.
3	word	Selects 64-bit word in 16-byte cache line.
2:0	—	All zero for 64-bit access.

The data format is shown in TABLE 15-11.

TABLE 15-11   L1 Data Cache Diagnostic Data – ASI_DCACHE_DATA (ASI 46_{16}, VA 0_{16}–7FFFFFFF8_{16})

Bit	Field	Initial Value	R/W	Description
63:0	data	0	RW	Cache data

The following sequence can be used by software to test the D-cache Data Array.

```
// Init Dcache Tag Array with unique tags
  foreach set (0 to 127)
foreach way (0 to 3)
     store TAGfield = $way to ASI_DCACHE_TAG($way, $set)


// Test Dcache Data Array
Write Test Data
foreach set (0 to 127)
     foreach way (0 to 3)
          foreach word (0 to 1)
               store unique testdata to ASI_DCACHE_DATA($way, $set, $word)


// Read Test Data
foreach set (0 to 127)
     foreach way (0 to 3)
          foreach word (0 to 1)
               set tag = $way, this guarantees store address matches load address
               load treg with ASI_DCACHE_DATA($tag, $set, $word)
               compare treg with expected testdata
```

15.3.3   ASI_DCACHE_TAG Register

The strands share a hyperprivileged ASI_DCACHE_TAG register at ASI 47_{16}, VA{63:0} = 0_{16}–$3FF0_{16}$ that is used for diagnostic access to the L1 data cache tag array. Diagnostic access to the L1 data cache tag array is done through 64-bit accesses that load/store the tag and valid bit and can optionally invert the tag parity bits on stores. Note that even parity is used (the parity bit is 1 if the number of ones in the data word is odd, making the total number of ones in the codeword even), and parity is across the tag only (does not include the valid bit).

Note that D-cache coherence is lost if a diagnostic store is done to this ASI, so after doing one or more stores, software must ensure that the entire cache is flushed. The format for addressing the entire L1 Data cache is shown in TABLE 15-12.

Nonprivileged access to this register causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap. Access with loads or stores that do not request 64 bits causes a *data_access_exception* trap. Loads from this ASI do *not* receive an error if the parity in the tag is bad.

TABLE 15-12 Format 3 L1 Data Cache Tag Diagnostic Addressing

Bit	Field	Description
63:13	—	*Reserved*
13	perren	Parity error enable. If '1' on a store, the computed parity for the tag is inverted. This address bit is ignored on loads.
12:11	way	Selects way in cache set.
10:4	set	Selects cache set.
3:0	—	*Reserved*

The data format is shown in TABLE 15-13.

TABLE 15-13 L1 Data Cache Diagnostic Tag – `ASI_DCACHE_TAG` (ASI 47_{16}, VA 0_{16}–$3FF0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:31	—	0	R	*Reserved*
30	parity	X	R	Ignored on writes (see PERREN above), returns parity bit on read.
29:1	tag	X	RW	Cache tag (PA{39:11}).
0	valid	0	RW	Valid bit for tag.

15.4   L2 Cache Registers

This section discusses L2 registers.

RegisterBaseAddress L2CSR Registers – A0 0000 0000$_{16}$.

15.4.1   L2 Control Register

Each cache bank has a control register. To enable the L2 cache, the **dis** bit must be set to 0 for all banks. The L2 cache can be disabled by setting the **dis** bit for all banks to 1. Operation when the **dis** bit of the L2 banks are not all the same is undefined. Note that while the L2 cache can be disabled during normal operation, it must be completely flushed of dirty cache lines before being disabled because once disabled it will no longer participate in the coherence protocol (that is, when disabled, the L2 cache is treated as if all its contents are invalid). Likewise, reenabling the L2 cache requires that the L2 cache be completely invalidated before clearing the **dis** bit since

the data in the L2 cache can be stale. In addition, before disabling the L2 cache, the L1 instruction and data caches must be disabled in all strands' `ASI_LSU_CONTROL_REG` registers, as the L1 caches cannot operate properly when the L2 cache is disabled. The L2 Control Register is available at offsets A9 0000 0000$_{16}$ or B9 0000 0000$_{16}$. Address bits 7:6 select the cache bank, address bits 31:8 and 5:3 are ignored (that is, the register aliases across the address range).

The L2 Control Register format is shown in TABLE 15-14.

TABLE 15-14 L2 Control Register – **L2_CONTROL_REG** (9 0000 0000$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:22	—	X	R	*Reserved*
21	dirclear	0	RW	If 0 and set to 1, the L2 directory is initialized to be all cleared. Only a 0→1 transition has any effect, so if it already has a value of 1, software must first clear this bit before setting it to 1, in order to clear the L2 directory.
20	dbgen	Preserved	RW	Mux select for the debug bus that goes to the IOB.
19:15	errorsteer	0	RW	Specifies the physical core (bits 19:17) and strand (bits 16:15) that receives all the L2 errors whose cause can't be linked to a specific strand.
14:3	scrubinterval	0	RW	Interval between scrubbing of adjacent sets in L2 in CMP core clocks. In units of 1M cycles.
2	scrubenable	0	RW	If set to 1, enable hardware scrub.
1	dmmode	0	RW	If set to 1, address bits 21 to 18 indicate the replacement way (with values C$_{16}$–F$_{16}$ aliasing to 4$_{16}$–7$_{16}$). If set to 0, all L2 ways are enabled under pseudo-LRU control.
0	dis	1	RW	If set to 1, disable the L2 cache. If set to 0, enable the L2 cache.

Each time the scrubber is invoked, it scrubs all lines at the same index. At the minimum setting of scrubinterval, the scrubber will scrub the cache slightly faster than once per second.

15.4.2 Other L2 Registers

L2 Cache registers are defined in other chapters; in particular, the error control and status registers are in Chapter 12, *Error Handling*.

15.5 L2 Cache Diagnostic Access

This section describes the control registers and diagnostic access for the L2 cache.

Diagnostic accesses will functionally work in the midst of other operations, but the results of L2 diagnostic accesses are inherently somewhat indeterminate because the state of the L2 cache is a moving target. Of course, diagnostic writes can put UltraSPARC T1 into an inconsistent or illegal state.

RegisterBaseAddress L2CSR Registers – A0 0000 0000$_{16}$. TABLE 15-15 shows the breakdown of the L2 address range.

TABLE 15-15 RegisterBaseAddress L2 CSR Registers – A0 0000 0000$_{16}$

Address Range (8 MSBs of the 40-bit Address)	Assigned to:	Comment
A0$_{16}$–A3$_{16}$/B0$_{16}$–B3$_{16}$	L2 Data	Diagnostic access to the L2 data array
A4$_{16}$–A5$_{16}$/B4$_{16}$–B5	L2 Tag	Diagnostic access to the L2 tag array.
A6$_{16}$–A7$_{16}$/B6$_{16}$–B7$_{16}$	L2 Tag VUAD	Diagnostic access to the L2 VUAD array.
A8$_{16}$–AF$_{16}$/B8$_{16}$–B$_{16}$F	L2 Registers	Error, control, and status registers.

15.5.1 L2 Data Diagnostic Access

Diagnostic access to the L2 data array is done through 64 bit read/writes that access a 32-bit data subblock along with the corresponding 7-bit ECC. The format for addressing the entire L2 cache is shown in TABLE 15-16.

Diagnostic loads of the L2 data do not check the ECC, and thus cannot generate an ECC error.

TABLE 15-16 Format 6 L2 Data Diagnostic Addressing

Bit	Field	Description
63:40	—	*Reserved*
39:32	select	Must be one of A0$_{16}$, A1$_{16}$, A2$_{16}$, A3$_{16}$, B0$_{16}$, B1$_{16}$, B2$_{16}$, or B3$_{16}$ to select L2 data diagnostic access. Can be any of the listed values (that is, the data diagnostic access is aliased throughout the A0– 3/B0–B3 address range)
31:23	—	*Reserved*, can be any value (that is, the data diagnostic access is aliased throughout the A0– A3/B0–B3 address range)
22	oddeven	Selects 32 bit word from 64-bit word selected by the word field.
21:18	way	Selects way in cache set. (Must be 0$_{16}$–B$_{16}$; C$_{16}$–F$_{16}$ are undefined.)
17:8	set	Selects cache set in bank.
7:6	bank	Selects cache bank.
5:3	word	Selects 64-bit word in 64-byte cache line.
2:0	—	All zero for 64-bit access

> **Programming Note** | Address bits 17:6 in the diagnostic access match the same address bits used by the hardware to select a set and bank in the cache. In addition, bits 21:18 match the same address bits used by the hardware to select a way in the L2 cache when in direct-mapped mode (L2_CONTROL_REG.dmmode = 1).

The data format is shown in TABLE 15-17.

TABLE 15-17 Register 64 23 L2 Diagnostic Data – L2_DIAG_DATA (0_{16})

Bit	Field	Initial Value	R/W	Description
63:39	—	X	R	*Reserved*
38:7	data	X	RW	Data
6:0	ecc	X	RW	ECC checkbits for data. See Table G-4 on page 326.

> **Note** | The — field ignores any value written to it, but unlike most reserved fields, it is not guaranteed to return all zeros on a read.

15.5.2 L2 Tag Diagnostic Access

Diagnostic access to the L2 tag array is done through 64-bit read/writes that access the tag along with the corresponding 6-bit ECC. The format for addressing the entire L2 cache is shown in TABLE 15-18.

Diagnostic loads of the L2 tag do not check the ECC, and thus cannot generate an ECC error.

TABLE 15-18 Format 7 L2 Tag Diagnostic Addressing

Bit	Field	Description
63:40	—	*Reserved*
39:32	select	Must be one of $A4_{16}$, $A5_{16}$, $B4_{16}$, or $B5_{16}$ to select L2 tag diagnostic access. Can be any of the listed values (that is, the tag diagnostic access is aliased throughout the $A4_{16}$–$A5_{16}$/$B4_{16}$–$B5_{16}$ address range)
31:22	—	*Reserved*, can be any value (that is, the tag diagnostic access is aliased throughout the $A4_{16}$–$A5_{16}$/$B4_{16}$–$B5_{16}$ address range)
21:18	way	Selects way in cache set. (Must be 0_{16}–B_{16}; C_{16}–F_{16} are undefined).
17:8	set	Selects cache set in bank.
7:6	bank	Selects cache bank.
5:3	—	*Reserved*, can be any value.
2:0	—	All zero for 64 bit access

> **Programming Note** | Address bits 17:6 in the diagnostic access match the same address bits used by the hardware to select a set and bank in the cache. In addition, bits 21:18 match the same address bits used by the hardware to select a way in the L2 cache when in direct-mapped mode (L2_CONTROL_REG.dmmode = 1).

The data format is shown in TABLE 15-19.

TABLE 15-19 Register 64 24 L2 Diagnostic Tag – L2_DIAG_TAG (4 0000 0000$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:30	—	X	R	*Reserved*
27:6	tag	X	RW	Tag, corresponds to addr{39:18}
5:0	ecc	X	RW	ECC checkbits for tag. See Table G-6 on page 327.

> **Note** | Values written to bits 63:30 of this register are ignored, but unlike most reserved fields, this field is *not* guaranteed to return all zeros when read.

15.5.3   L2 VUAD Diagnostic Access

The valid, used, allocate, and dirty (VUAD) array contains the state bits for the L2 entries. TABLE 15-20 lists the conditions that cause a transition in VUAD bits.

TABLE 15-20   VUAD Bit Transitions

Bit	0 → 1 Transition	1 → 0 Transition
Valid	Line fill	Line eviction or DMA 64-byte write hit
Used (individual)	Line fill or hit where an unused and unallocated way is present in the set	Line eviction or DMA 64-byte write hit
Used (all)	NA	Hit where no unused and unallocated way present in the set
Allocated	Allocated way on an eviction	Fill with new line
	Hit way on a partial store, DMA 8B write, or atomic first pass through L2	Partial store, DMA 8-byte write, or atomic final pass through L2
Dirty	Store hit to line	Line eviction or DMA 64-byte write hit

Diagnostic access to the L2 VUAD array is done through a pair of address access ranges. The first accesses the valid and dirty bits for an entire set plus the parity for each of those bits across the set via 64-bit read/writes. The format for addressing the entire L2 cache is shown in TABLE 15-21.

Diagnostic loads of the VUAD do not check parity, and thus cannot generate a parity error.

TABLE 15-21   Format 8 L2 VD Diagnostic Addressing

Bit	Field	Description
63:40	—	*Reserved*
39:32	select	Must be one of $A6_{16}$, $A7_{16}$, $B6_{16}$, or $B7_{16}$ to select L2 VD diagnostic access. Can be any of the listed values (that is, the VD diagnostic access is aliased throughout the $A6_{16}$–$A7_{16}$/$B6_{16}$–$B7_{16}$ address range where bit 22 is 1).
31:23	—	*Reserved*, can be any value. (that is, the VD diagnostic access is aliased throughout the $A6_{16}$–$A7_{16}$/$B6_{16}$–$B7_{16}$ address range where bit 22 is 1).
22	vdsel	Must be set to 1.
21:18	—	*Reserved*, can be any value. (that is, the VD diagnostic access is aliased throughout the $A6_{16}$–$A7_{16}$/$B6_{16}$–$B7_{16}$ address range where bit 22 is 1).
17:8	set	Selects cache set in bank.
7:6	bank	Selects cache bank.
5:3	—	*Reserved*, can be any value.
2:0	rsvd4	All zero for 64 bit access.

Programming Note | Address bits 17:6 in the diagnostic access match the same address bits used by the hardware to select a set and bank in the cache.

The data format is shown in TABLE 15-22.

TABLE 15-22   L2 Diagnostic VD – **L2_DIAG_VD** ($6\ 0040\ 0000_{16}$)

Bit	Field	Initial Value	R/W	Description
63:26	—	X	R	*Reserved*
25	vparity	X	RW	Parity for all valid bits.
24	dparity	X	RW	Parity for all dirty bits.
23:12	valid	X	RW	Valid bits for way 11 down to way 0.
11:0	dirty	X	RW	Dirty bits for way 11 down to way 0.

Note | The — field ignores any value written to it, but unlike most reserved fields, it is not guaranteed to return all zeros on a read.

The second range accesses the UA bits for an entire set via 64 bit read/writes. The format for addressing the entire L2 cache is shown in TABLE 15-23.

TABLE 15-23 Format 9 L2 UA Diagnostic Addressing

Bit	Field	Description
63:40	—	*Reserved*
39:32	select	Must be one of $A6_{16}$ or $A7_{16}$ to select L2 UA diagnostic access. Can be any of the listed values (that is, the UA diagnostic access is aliased throughout the $A6_{16}$–$A7_{16}$/$B6_{16}$–$B7_{16}$ address range where bit 22 is 0).
31:23	—	*Reserved*, can be any value. (that is, the UA diagnostic access is aliased throughout the $A6_{16}$–$A7_{16}$ address range where bit 22 is 0).
22	vdsel	Must be set to 0.
21:18	—	*Reserved*, can be any value. (that is, the UA diagnostic access is aliased throughout the $A6_{16}$–$A7_{16}$ address range where bit 22 is 0).
17:8	set	Selects cache set in bank.
7:6	bank	Selects cache bank.
5:3	—	*Reserved*, can be any value.
2:0	rsvd4	All zero for 64-bit access.

> **Programming Note** | Address bits 17:6 in the diagnostic access match the same address bits used by the hardware to select a set and bank in the cache.

The data format is shown in TABLE 15-24.

TABLE 15-24 L2 Diagnostic UA – L2_DIAG_UA (6 0000 0000$_{16}$)

Bit	Field	Initial Value	R/W	Description
63:26	—	X	R	*Reserved*
25	—	X	R	*Reserved*; was U parity.
24	aparity	X	RW	Parity for all allocated bits.
23:12	used	X	RW	Used bits for way 11 down to way 0.
11:0	alloc	X	RW	Allocated bits for way 11 down to way 0.

> **Note** | The *reserved* fields ignore any value written to them, but unlike most reserved fields, they are not guaranteed to return all zeros when read.

The used bits are not parity protected, since their value is noncritical. Any error in the used bits will cause potentially different replacement order, but still functionally correct operation.

15.5.4   Software Error Scrubbing Support

Some errors will leave the L2 cache with a correctable error, which then needs to be scrubbed to prevent repetitive traps for effectively the same soft error. Flushing the data out of the cache back to memory will correct the error.

With a 12-way set associative cache, with pseudo-LRU replacement and no explicit flush instruction, flushing the error line is not entirely trivial. However, to make the flush provably workable, the L2 has a "Direct Mapped Replacement Mode" which forces the replacement algorithm to simulate a direct-mapped cache. This direct-mapped mode is safe to spuriously enable in a running system, since lines inserted in either mode can still be found (tag matched) normally.

To flush a line, software (hypervisor) would enable Direct-Mapped mode, fault in 12 cache lines with the same index as the error line (but is not the error line), then restore (disable?) the original state of the Direct-Mapped mode.

Scrubbing correctable main memory errors uses the same support. To scrub, fault the line into the L2, dirty it without modifying it (CAS), then use Direct-Mapped mode and 12 other fault-ins to force the error line out.

15.6   EFUSE Registers

The EFUSE block contains permanent programmed information that is burned in as part of the manufacturing process. The following registers give software access to the portion that is pertinent to software.

RegisterBaseAddress 1 IOBMAN – 98 0000 0000$_{16}$.   The following Processor Serial Number register contains a unique serial number. The format is shown in TABLE 15-27.

TABLE 15-25 Processor Serial Number Register – PROC_SER_NUM (0820_{16})

Bit	Field	Initial Value	R/W	Description
63:44	salt	X	R	Extra bits.
43:42	ti	X	R	TI Tracking bits.
41	r	X	R	Unit has EFUSE array repair
40:35	row	X	R	Wafer row ID.
34:29	col	X	R	Wafer column ID.
28:24	waf	X	R	Wafer ID.
23:22	fab	X	R	Fab number.
21:0	lot	X	R	Fab lot number.

The CORE_AVAIL register, shown in TABLE 15-28, indicates which virtual processor/ strand is available. Defective virtual processors are marked unavailable during manufacturing by means of fuses. The IOB does not implement any protection against access to unavailable virtual processors. For example, software may try to send a RESUME to an unavailable strand by writing to int_vec_dis, or the TAP may read/write to ASIs of an unavailable strand. The resultant behavior is undefined. The IO Bridge wakes up the first available strand after a reset.

TABLE 15-26 Strand Available – CORE_AVAIL (0000_{16}–0830_{16})

Bit	Field	Initial Value	R/W	Description
63:32	—	0	R	*Reserved*
31:0	avail	FFFF FFFF$_{16}$	R	1 means the strand is available. Initial value is that all strands are available, until loaded from EFUSE array.

> **Note** | The granularity of availability in UltraSPARC T1 is a physical core. Thus, each nibble of the CORE_AVAIL register will either be all ones or all zeros.

The register defined in TABLE 15-29 contains the parity results of all of the EFUSE entries. This should be checked at boot, and if any entry is 1, the configuration of the chip is suspect.

TABLE 15-27 Fuse Status Register – IOB_FUSE (0000_{16}–0840_{16})

Bit	Field	Initial Value	R/W	Description
63:0	status	0	R	1 means an EFUSE entry had a parity error and is suspect.

Modular Arithmetic

Processor streaming extensions are provided to accelerate network stack processing. The extensions support modular multiplication and exponentiation for SSL.

Modular arithmetic operations operate on data stored in the Modular Arithmetic memory. Once initiated, these operations operate in parallel with normal SPARC instruction execution. The SPARC system can synchronize with the completion of the operation using a load from the Modular Arithmetic Sync register or with interrupts. The modular arithmetic state is maintained per physical processor core and is accessed through load and store alternate instructions with the hyperprivileged ASI_MA (40_{16}). The state accesses must be 8-byte aligned, or a *mem_address_not_aligned* trap is taken.

16.1 Modular Arithmetic State

The modular arithmetic state consists of the elements shown in TABLE 16-1.

TABLE 16-1   Modular Arithmetic State

Name	Address	Size	Function
MPA		8 bytes	Modular physical address
MA_ADDR		8 bytes	Modular arithmetic memory addresses
NP		8 bytes	Modular arithmetic N Prime value
MA_CTL		8 bytes	Control parameters

16.1.1   `ASI_MA_CONTROL_REG` Register

Each physical processor core has a hyperprivileged read-write ASI_MA_CONTROL_REG register at ASI 40_{16}, VA{63:0} = 80_{16} that is used to control modular arithmetic operations. The format of the register is shown in TABLE 16-2.

TABLE 16-2 Modular Arithmetic Control Register – `ASI_MA_CONTROL_REG`
(ASI 40_{16}, VA 80_{16}) *(1 of 2)*

Bit	Field	Initial Value	R/W	Description
63:14	—	0	R	*Reserved*
13	perrinj	0	RW	If 1, each operation that writes to the MA memory will invert the parity bit generated.
12:11	strand	X	RW	Specifies the strand that will receive the disrupting trap on the completion of the modular arithmetic operation (if the int bit is set). Also sets the strand that will receive the disrupting *corrected_ECC_error* trap if a correctable ECC error in memory is encountered on an MA load. The *corrected_ECC_error* trap is sent to the strand regardless of the value of the int bit. Software can load balance interrupt service across strands by changing this field.
10	busy	0	R	If set, a modular arithmetic operation is in progress. If clear, no modular arithmetic operation is in progress.
				Note: When the operation completes and the int bit is set, the busy bit is cleared regardless of whether the completion interrupt has been delivered. Therefore, seeing the busy bit cleared does not guarantee that the completion interrupt has been delivered. However, the completion interrupt will never be delivered before the busy bit is cleared, so seeing the busy bit set implies that the completion interrupt has not been delivered.
				Note: If the control register is written while an operation is in progress (the busy bit is set), the operation in progress will continue to a stable point (that is, all posted stores are acknowledged) before starting the new operation. The busy bit may go low after the aborted operation reaches this stable point, but will be reset to high once the second operation starts.
9	int	X	RW	If set, the streaming operation will generate a disrupting trap to the current strand on the completion of the modular arithmetic operation. The trap will use the implementation_dependent_exception_20 (modular_arithmetic_interrupt) vector at priority level 16. If not set, then software can synchronize with the completion the streaming operation using the MASync instruction.
				Note: If the control register is written while an operation is in progress (the busy bit is set) and the interrupt bit is set, no completion interrupt will be generated.

Bit	Field	Initial Value	R/W	Description
8:6	op	X	RW	Specifies the type of modular arithmetic operation as shown below. Details of the operations are described in subsequent sections.

Value	Operations
0	Load MA memory
1	Store MA memory
2	Modular multiply
3	Modular reduction
4	Modular exponentiation loop
5-7	*Reserved*

Bit	Field	Initial Value	R/W	Description
5:0	length	X	RW	Specifies one less than the length of the modular arithmetic operation in words.

Note: Due to size limits on the MA memory (1280 bytes), the maximum value that should be specified for length is 31. With length at 31, the full MA memory will be used for a modular multiply or exponentiation loop (5 operands * (31+1) words * 8 bytes / word = 1280 bytes).

Implementation Note: On a Store MA memory operation, the value of the length field is decremented as the words are stored to memory. For all other operations, the value of the length field will remain the value written on the previous MA_CTL store.

Storing into the MA_CTL register will trigger the start of a modular arithmetic operation. Other state registers should be updated before loading the MA_CTL register.

All fields must be set to a valid, nonreserved value on the store of the `ASI_MA_CONTROL_REG`, or an *illegal_instruction* trap will be taken.

Implementation Note | Not all revisions of UltraSPARC T1 implement the *illegal_instruction* trap for invalid values being stored to `ASI_MA_CONTROL_REG`. Operation of an UltraSPARC T1 chip is undefined when an invalid value is stored to `ASI_MA_CONTROL_REG`.

Note that only hyperprivileged code is affected by this. Any attempted access by user or privileged code to `ASI_MA_CONTROL_REG` will cause a *data_access_exception* trap.

> **Note** | There is an UltraSPARC T1 "feature" where it is possible for
> nonprivileged or privileged code can silently abort SPU ops, by issuing
> illegal SPU opcodes (which trap, but still have the aborting side-effect
> on hyperprivileged use). To avoid this, all STXAs to the
> ASI_MA_CONTROL_REG should be preceeded by a "MEMBAR
> #Sync", to ensure that no stores in the store buffer delay execution of
> this STXA. In addition, do not cause an abort by issuing a new
> operation (where you care about the results). Instead, aborts can be
> caused by issuing an effective SPU NOP.

16.1.2   `ASI_MA_MPA_REG` Register

Each physical processor core has a hyperprivileged read-write `ASI_MA_MPA_REG`
register at ASI 40_{16}, VA$\{63:0\} = 88_{16}$ that is used to specify the physical address of the
bytes to be loaded or stored into the MA memory. Since this address must be aligned
to 8 bytes, the lower 3 bits of the address are forced to all zeros. MA load and store
operations are not permitted to I/O addresses, and bit 39 of the address is forced to
zero as well. Nonprivileged access to this register causes a *privileged_action* trap.
Supervisor access causes a *data_access_exception* trap.

TABLE 16-3 defines the format of the `ASI_MA_MPA_REG` register.

TABLE 16-3   Modular Arithmetic Address Register – ASI_MA_MPA_REG (ASI 40_{16}, VA 88_{16})

Bit	Field	Initial Value	R/W	Description
63:40	—	0	R	*Reserved*
39	address39	0	R	Most significant bit of physical address of bytes to be loaded or stored into MA memory, always forced to 0 to prevent loads and stores to IO space.
38:3	address	X	RW	Physical address of bytes to be loaded or stored into MA memory.
2:0	—	0	R	*Reserved*

The number of words to be loaded or stored is specified by the length field in the
ASI_MA_CONTROL_REG register plus one and should be less than or equal to 64.
MA memory loads and stores will produce undefined results if the length plus one
plus offset is greater than 160. Multiword operands are normally stored in little
endian order across words (least significant word at the lowest address), but big
endian order within a 64-bit word (most significant byte at the lowest address). The
exponent operand for modular exponentiation is stored in big endian order across
words as well as within a 64-bit word.

16.1.3 `ASI_MA_ADDR_REG` Register

Each physical processor core has a hyperprivileged read-write `ASI_MA_ADDR_REG` register at ASI 40_{16}, VA{63:0} = 90_{16} that is used to specify the initial word offsets into the MA memory for the modular operations. The format of the register is shown in TABLE 16-4.

TABLE 16-4 Modular Arithmetic MA Offset Register – `ASI_MA_ADDR_REG` (ASI 40_{16}, VA 90_{16})

Bit	Field	Initial Value	R/W	Description
63:48	—	0	R	*Reserved*
47:40	addr5	X	RW	Specifies the size − 1 of the exponent in bytes for exponentiation. Not used for modular reduction or multiplication.
39:32	addr4	X	RW	Specifies the offset of the least significant word of the result (X) for multiplication, the exponent (E) for exponentiation. Not used for modular reduction.
31:24	addr3	X	RW	Specifies the offset of the least significant word of the temporary (M) for multiplication, the result (X) for exponentiation. Not used for modular reduction.
23:16	addr2	X	RW	Specifies the offset of the least significant word of the result (R) for modular reduction, the modulus (N) for multiplication, the modulus (N) for exponentiation.
15:8	addr1	X	RW	Specifies the offset of the least significant word of the modulus (N) for modular reduction, the multiplicand (B) for multiplication, the temporary (M) for exponentiation.
7:0	addr0	X	RW	Specifies the offset of the least significant word of the source (A) for modular reduction, the multiplier (A) for multiplication, the base (A) for exponentiation. Also specifies the offset of the initial word for a MA memory load or store.

16.1.4 `ASI_MA_NP_REG` Register

Each physical processor core has a hyperprivileged read-write `ASI_MA_NP_REG` register at ASI 40_{16}, VA{63:0} = 98_{16} that is used to specify the modular arithmetic N prime value. Nonprivileged access to this register causes a *privileged_action* trap. Supervisor access causes a *data_access_exception* trap.

TABLE 16-5 defines the format of the `ASI_MA_NP_REG` register.

TABLE 16-5 Modular Arithmetic N Prime Register – `ASI_MA_NP_REG` (ASI 40_{16}, VA 98_{16})

Bit	Field	Initial Value	R/W	Description
63:0	n_prime	X	R/W	Modular arithmetic N prime value.

16.1.5 `ASI_MA_SYNC_REG` Register

Each physical processor core has a hyperprivileged `ASI_MA_SYNC` register at ASI 40_{16}, VA{63:0} = $A0_{16}$.

TABLE 16-6 defines the format of the `ASI_MA_SYNC` register.

TABLE 16-6   Modular Arithmetic Sync Register – `ASI_MA_SYNC` (ASI 40_{16}, VA $A0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:0	—	0	R	Always loads as 0, has side effect of generating a modular arithmetic sync.

A load from this register is used to synchronize a strand with the completion of asynchronous modular arithmetic operations. If the strand doing the load is the strand specified in the strand field of the Modular Arithmetic Control Register, completion of this load operation will ensure that subsequent stores are ordered after any modular arithmetic operation stores. The result of the Modular Arithmetic Sync load for this case will be zero if the MA operation was not aborted. If the MA operation was aborted, the load will complete once the operation aborts, and the destination register of the load will not be updated.

If the strand doing the load is not the strand specified in the strand field of the Modular Arithmetic Control register, the load will complete immediately, regardless of the current state of the MA operation, and the destination register of the load will not be updated. A store to this register will result in a *data_access_exception* trap.

Note	If software desires to know whether the Modular Arithmetic Sync load actually did a synchronization following a successful completion of the modular arithmetic operation, it can set the destination register to a nonzero value before doing the load. If the load did a successful synchronization, the destination register will be zero; otherwise, it will remain at the initial non-zero value.

Programming Notes	(1) Because the Modular Arithmetic Sync load does not finish execution for the strand until the modular arithmetic operation is completed, loads from this register should only be used for short MA operations. A long MA operation, such as a 2048-bit Modular Exponentiation, which could take nearly 3 million cycles in the worst case, should only use the modular_arithmetic_interrupt for completion notification or alternatively perform the MA_SYNC load only when software knows the long operation is nearly complete.

(2) It is possible for a strand to hang on a Modular Arithmetic Sync load. This situation arises because when MA_CTL.int = 1, the modular arithmetic hardware does not consider the operation to have reached a stable point during an abort (see Section 16.2) until the completion interrupt is taken by the strand. So if the MA_CTL.int bit is 1, and PSTATE.ie is 0 at the time that the completion interrupt would be generated, and software does an abort operation from another strand followed by a Modular Arithmetic Sync load, that load will hang until either (a) the first strand sets PSTATE.ie to 1 and takes the completion interrupt or (b) that second operation is aborted by another strand (and then that new strand could also hang itself by doing a Modular Arithmetic Sync). This implies that a strand that starts a modular arithmetic operation with MA_CTL.int = 1 and then never sets PSTATE.ie to 1 will make the Modular Arithmetic unit unusable by the other strands on the physical processor core.

(3) It is possible for a strand doing a legal Modular Arithmetic Sync load to get the wrong result if that load is followed by an access from another strand to the Modular Arithmetic ASI (40_{16}), but to a virtual address that does not address a MA register (VA 0_{16}–78_{16} or $A8_{16}$–$F8)_{16}$. The strand doing the subsequent access to the VA that does not address a MA register will hang. This is the result of a hardware erratum in UltraSPARC T1 where the result for the second strand's access is delivered to the first strand, and no result is ever delivered to the second strand. To avoid generating this hang, software should never generate accesses to the MA ASI using virtual addresses other than 80_{16}, 88_{16}, 90_{16}, 98_{16}, or $A0_{16}$.

16.2   Aborting an MA Operation

A modular arithmetic operation can be aborted at any time by overwriting the Modular Arithmetic Control register with a new valid operation. When the Modular Arithmetic Control register is written while an operation is in progress (the busy bit is set), the operation in progress will continue to a stable point (that is, all posted stores are acknowledged) before starting the new operation. If the int bit of the Modular Arithmetic Control register was set for the aborted operation, no completion interrupt will be generated to the strand specified in the strand field of the aborted operation. If the strand specified in the strand field of the aborted

operation had already issued a Modular Arithmetic Sync load before the operation was aborted, the Modular Arithmetic Sync load will immediately complete without updating its destination register.

See the Note on page 262 regarding abortion of SPU ops.

16.3 MA Memory

1280 bytes of local MA memory is used to supply operands to modular arithmetic operations. Each physical processor core has one copy of the streaming MA memory state. Refer to TABLE 16-7.

TABLE 16-7   MA Memory State

Name	Size	Function
MAMEM	1280 bytes	Modular arithmetic memory array

Load MA memory is used to load bytes into the MA memory of the current physical processor core. Store MA memory is used to store bytes from the local MA memory of the current physical processor core. The main memory physical address of the bytes is specified by the MPA register. This address must be aligned to 8 bytes. The lower 8 bits of the MA_ADDR register specify the initial word offset into the MA memory where the first byte is loaded or stored. The number of words to be loaded or stored is specified by the length field in the CTL register plus one and should be less than or equal to 64. MA memory loads and stores will produce undefined results if the length plus one plus offset is greater than 160. Multiword operands are normally stored in little endian order across words (least significant word at the lowest address), but big endian order within a 64-bit word (most significant byte at the lowest address). The exponent operand for modular exponentiation is stored in big endian order across words as well as within a 64-bit word.

The MA memory is protected by parity. *Modular Arithmetic Memory (MAU)* on page 124 describes the behavior when a parity error is detected on any MA memory access (either an MA operation or an MA memory store operation).

For a modular arithmetic memory store operation, the completion interrupt (or `ASI_MA_SYNC` completion) is generated only when all the stores have been acknowledged. For stores to data space (memory that has never been executed and thus will not be present in any instruction cache), all stores are guaranteed to be globally visible after receiving the MA completion interrupt (or completing the `ASI_MA_SYNC` load). For stores to instruction space (memory that has been executed and thus may be present in an instruction cache), all stores are guaranteed to be globally visible only after a FLUSH instruction is executed following the MA completion interrupt (or the `ASI_MA_SYNC` load completion).

16.4   Modular Reduction

Modular reduction performs a modular reduction of one operand and a modulus stored in the MA memory array. The result is stored in the MA memory array. The length field plus one specifies the number of 64-bit words in each of the operands. Bits 7:0 of the MA_ADDR register specify the offset of the least significant word of the source (A operand). Bits 15:8 of the MA_ADDR register specify the offset of the least significant word of the modulus (N operand). Bits 23:16 of the MA_ADDR register specify the offset of the least significant word of the result operand (R operand).

If the source operand is less than the modulus, then the source operand is stored in the destination operand; otherwise, the source operand minus the modulus is stored in the destination operand. The source operand should not overlap with the destination operand.

Modular reduction should be performed at 4 clocks per output word.

Processing should start from the most significant source, modulus and output word. When the source word and the corresponding modulus word are equal, store a zero to the corresponding output word and proceed to the next word. Otherwise, a comparison of these two words will determine whether the input operand is greater than the modulus. If the input operand is less than the modulus, then the remaining input operands can be copied to the corresponding output words. Otherwise, the modulus needs to be subtracted from remaining input words. This multi-precision subtract should be done by starting with the least significant input word and proceeding up to and including the first nonequal input word. The algorithm below lists the modular reduction behavior.

```
if ACCUM != 0 {
    for I=0 to Length {    // Length = number of words - 1
        X[I] = A[I] - N[I]   // Subtraction with borrow
    }
} else {
    I = Length
    while (I>=0) && (A[I]==N[I]) {
        I = I-1
    }
    if (I>=0) && (A[I]<N[I]) {
        for I=0 to Length {
            X[I] = A[I]
        }
    } else {
        if (I>=0) && (A[I]>N[I]) {
            for I=0 to Length {
                X[I] = A[I] - N[I] // Subtraction with borrow
            }
        } else {
            for I=0 to Length {
                X[I]=0
            }
        }
    }
}
```

16.5   Modular Multiplication

Modular multiplication performs a Montgomery reduction modular multiplication of two operands and a modulus stored in the MA memory array. The result is stored in the MA memory array.

The length field plus one specifies the number of 64-bit words in each of the operands. Bits 7:0 of the MA_ADDR register specify the offset of the least significant word of the multiplier (A operand). Bits 15:8 of the MA_ADDR register specify the offset of the least significant word of the multiplicand (B operand). Bits 23:16 of the MA_ADDR register specify the offset of the least significant word of the modulus (N operand). Bits 31:24 of the MA_ADDR register specify the offset of the least significant word of the temporary operand (M operand). Bits 39:32 of the MA_ADDR register specify the offset of the least significant word of the result operand (X operand).

The NP register contains the inverse modulus operand (N′). The source operands should not overlap with the destination operand.

The modular multiplication algorithm operates on operands A, B, N, M and stores the result in X. The operands are stored at various offsets in the streaming memory specified by the MA_ADDR register as described above. The algorithm uses an unsigned 64*64 multiply operation, which produces a 128-bit result. The multiplier result is accumulated into a 136-bit accumulator (ACCUM). The algorithm proceeds as follows.

```
ACCUM = 0
For I=0 to Length { // Length is one less than the number of words
    For j=0 to I-1 { // skipped on first I=0 iteration
          ACCUM += A[j]*B[I-j]
          ACCUM += M[j]*N[I-j]
        ACCUM += A[I]*B[0]
        M[I] = ACCUM * N' //64 LSB of accum, store 64 LSB of product
        ACCUM += M[I]*N[0]
        ACCUM >>= 64
    }
}
For I=Length+1 to (2*Length)+1  {
    For j=I-Length to Length  {//skipped last I=(2*Length)+1 iteration
        ACCUM += A[j]*B[I-j]
        ACCUM += M[j]*N[I-j]
    }
    M[I-length-1] = ACCUM //64 LSB of accum
    ACCUM >>= 64
}
X := ModReduction(ACCUM|M, N) //LSB of ACCUM is prepended to M
```

When A = B (same offset in streaming memory), the algorithm is modified to reduce the number of multiplies as shown below:

```
ACCUM = 0
For I=0 to Length { // Length is one less than the number of words
       For j=0 to (I-1)>>1  // skipped on first I=0 iteration
                   ACCUM += 2*A[j]* A[I-j]
       If I is even
             ACCUM += A[I/2]^2
       For j=0 to I-1  // skipped on first I=0 iteration
             ACCUM += M[j]*N[I-j]
       M[I] = ACCUM * N' //64 LSB of accum, store 64 LSB of product
             ACCUM += M[I]*N[0]
             ACCUM >>= 64
       For I=Length+1 to (2*Length)+1
       For j=I-Length to (I-1)>>1 //skipped last two I iterations
             ACCUM += 2*A[j]* A[I-j]
       If I is even
             ACCUM += A[I/2]^2
       For j=I-Length to Length //skipped last I=2*Length+1 iteration
             ACCUM += M[j]*N[I-j]
       M[I-length-1] = ACCUM //64 LSB of accum
             ACCUM >>= 64
       X := ModReduction(ACCUM|M, N) //LSB of ACCUM is prepended to M
}
```

16.6   Modular Exponentiation Loop

Modular exponentiation performs the inner loop of modular exponentiation of a base, exponent and modulus operands stored in the MA memory array. The result is stored in the MA memory array.

The length field plus one specifies the number of 64-bit words in the base, temporary and modulus operands. Bits 7:0 of the MA_ADDR register specify the offset of the least significant word of the base (A operand). Bits 15:8 of the MA_ADDR register specify the offset of the least significant word of the temporary (M operand). Bits 23:16 of the MA_ADDR register specify the offset of the least significant word of the modulus (N operand). Bits 31:24 of the MA_ADDR register specify the offset of the least significant word of the result operand (X operand). Bits 39:32 of the MA_ADDR register specify the offset of the least significant word of the exponent operand (E operand). Bits 47:40 specify the size of the exponent in bytes minus one (ES).

The NP register contains the inverse modulus operand (N'). A, M, and N operands are stored in little endian order across words (least significant word at the lowest address), but big endian order within a 64-bit word (most significant byte at the lowest address).

E is split into bytes, and these bytes are packed into 64-bit registers using a big-endian ordering across words and across bytes within a word. There is no bit reversal, that is, in a MA_MEM register holding bits from E, bit significance increases with bit number. Since E is an arbitrary number of bytes, the last word could contain less then 8 bytes and these are similarly packed from the most significant byte in the register downwards.

TABLE 16-8 contains an example for E containing 20 bytes (E{19} is MSB down to E{0} for LSB) and stored at MA_MEM{10}.

TABLE 16-8 Example for E Containing 20 Bytes Stored at MA_MEM{10}

Word	Bits 63:56	Bits 55:48	Bits 47:40	Bits 39:32	Bits 31:24	Bits 23:16	Bits 15:8	Bits 7:0
MA_MEM{10}	E{19}	E{18}	E{17}	E{16}	E{15}	E{14}	E{13}	E{12}
MA_MEM{11}	E{11}	E{10}	E{9}	E{8}	E{7}	E{6}	E{5}	E{4}
MA_MEM{12}	E{3}	E{2}	E{1}	E{0}	—	—	—	—

Notes (1) E is stored in the opposite word-order than all other MA_MEM operands.

(2) If (ES + 1) is not a multiple of 8, then the packing of E into registers leaves a hole in the last register after the least significant byte and software needs to take care of this.

The modular exponentiation loop operates on operands A, E, X, N, M and stores the result in X. The operands are stored at various offsets in the streaming memory specified by the SRC register as described above. The modular exponentiation loop is based on the modular multiply (ModMultiply) and modular reduction (ModReduction) operations described above. X needs to be initialized to X = r mod N, where $r = 2^n$ for n-bit Montgomery modular exponentiation, for example, $r = 2^{1024}$ for 1024-bit exponentiation. The algorithm proceeds as follows:

```
for i = 0 to 8*(ES+1) - 1
X := ModMultiply(X, X, N, M)
      if E[i] = 1 then  // Check if bit i of the exponent is set (bit 0 is the MSB)
      X := ModMultiply (X, A, N, M)
```

16.7 Error Behavior

Error behavior for streaming and modular arithmetic operations is described in Chapter 12, *Error Handling*.

Assembly Language Syntax

The assembly language syntax used in this document follows that described in the "Assembly Language Syntax" appendix of the *UltraSPARC Architecture 2005* specification.

Programming Guidelines

B.1   Multithreading

In UltraSPARC T1, execution is switched in round-robin fashion every cycle among the strands that are ready to issue another instruction. Context switching is built into the UltraSPARC T1 pipeline and takes place during the SWITCH stage, thus contexts are switched each cycle with no pipeline stall penalty.

The following instructions change a strand from a ready-to-issue state to a not-ready-to-issue state, until hardware determines that their input/execution requirements can be satisfied:

- All branches (including CALL, JMPL, etc.)
- All VIS instructions
- All floating point (FPops)
- All WRPR, WR, WRHPR
- All RDPR, RD, RDHPR
- SAVE(D), RESTORE(D), RETURN, FLUSHW (all register management)
- All MUL and DIV
- MULSCC
- MEMBAR #Sync, MEMBAR #StoreLoad, MEMBAR #MemIssue
- FLUSH
- All loads
- All floating-point memory operations
- All memory operations to alternate space
- All atomics load-store operations
- Prefetch

B.2 Pipeline Strand Flush

The front end of the UltraSPARC T1 pipeline prevents instructions from being issued to the rest of the pipeline unless there is a high probability (for most instructions, a probability of 1.0) of the instruction having all its input dependencies satisfied. For certain instructions, the input dependencies cannot be determined by the front end, and the instruction (and any subsequent instructions issued from that strand) need to be flushed from the pipeline and replayed. TABLE B-1 lists instructions that may end up causing a strand flush.

TABLE B-1 Pipeline Strand Flush Events

Event	Strand Flush Description
Loads	The strand will be flushed if the load encounters a cache miss while executing with STRAND_STS_REG.spec_en = 1.
multiply/divide/ floating-point operate	Resource contention can cause a strand flush.
store buffer full	The strand will be flushed until space is available in the store buffer.
trap	Instruction and any subsequent instructions in the pipeline from that strand are flushed, and fetching restarts at the trap vector.
Idle/Resume interrupts	Instruction and any subsequent in the pipeline from that strand are flushed.
I-cache parity errors	The strand will be flushed and the instruction refetched from the L2 cache.
IRF or FRF ECC errors	Instruction and any subsequent in the pipeline from that strand are flushed.

B.3 Instruction Latencies

TABLE B-2 lists the single-strand instruction latencies for UltraSPARC T1. When multiple strands are executing, much of the additional latency for multicycle instructions will be overlapped with execution of the additional strands.

In this table, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 5-1 on page 22.

Instruction	Latency	Comments
ADD	1	
ADDc	1	
ADDCC	1	
ADDNcc	1	
AND	1	
ANDcc	1	
ANDN	1	
ANDNcc	1	
BA	3?	
BA_A	4	
BA_A_PN	4?	
BA_PN	3	
BA_XCC	3	
BA_XCC_A	4?	
BA_XCC_A_PN	4	
BA_XCC_PN	3?	
BCC	3	
BCC_A	3	
BCC_A_PN	4	
BCC_PN	3	
BCC_XCC	3	
BCC_XCC_A	3	
BCC_XCC_A_PN	4	
BCC_XCC_PN	3?	
BCS	3	
BCS_A	4	
BCS_A_PN	4?	
BCS_PN	3	
BCS_XCC	3	
BCS_XCC_A	4	

TABLE B-2 Instruction Latencies *(2 of 14)*

Instruction	Latency	Comments
BCS_XCC_A_PN	4	
BCS_XCC_PN	4?	
BE	3	
BE_A	4?	
BE_A_PN	4	
BE_PN	3	
BE_XCC	3	
BE_XCC_A	4	
BE_XCC_A_PN	4	
BE_XCC_PN	3	
BG	3	
BG_A	3	
BG_A_PN	4	
BG_PN	3	
BG_XCC	3	
BG_XCC_A	3	
BG_XCC_A_PN	3?	
BG_XCC_PN	3	
BGE	3	
BGE_A	3	
BGE_A_PN	4?	
BGE_PN	3	
BGE_XCC	3	
BGE_XCC_A	4	
BGE_XCC_A_PN	4?	
BGE_XCC_PN	3	
BGU	3	
BGU_A	3	
BGU_A_PN	3?	
BGU_PN	3	

Instruction	Latency	Comments
BGU_XCC	3	
BGU_XCC_A	4	
BGU_XCC_A_PN	4	
BGU_XCC_PN	3	
BL	3	
BL_A	3	
BL_A_PN	3?	
BL_PN	3	
BL_XCC	3	
BL_XCC_A	4?	
BL_XCC_A_PN	4	
BL_XCC_PN	3	
BLE	3	
BLE_A	3	
BLE_A_PN	3?	
BLE_PN	3	
BLE_XCC	3	
BLE_XCC_A	4	
BLE_XCC_A_PN	4?	
B LE_XCC_PN	3	
BLEU	3	
BLEU_A	4?	
BLEU_A_PN	4	
BLEU_PN	3?	
BLEU_XCC	3	
BLEU_XCC_A	4	
BLEU_XCC_A_PN	4	
BLEU_XCC_PN	3	
BN	3	
BN_A	4	

TABLE B-2 Instruction Latencies *(4 of 14)*

Instruction	Latency	Comments
BN_A_PN	4	
BN_PN	3	
BN_XCC	3	
BN_XCC_A	4?	
BN_XCC_A_PN	4	
BN_XCC_PN	3	
BNE	3	
BNE_A	3	
BNE_A_PN	3?	
BNE_PN	3	
BNE_XCC	3?	
BNE_XCC_A	4	
BNE_XCC_A_PN	4	
BNE_XCC_PN	3?	
BNEG	3	
BNEG_A	4	
BNEG_A_PN	4	
BNEG_PN	3	
BNEG_XCC	3	
BNEG_XCC_A	4	
BNEG_XCC_A_PN	4	
BNEG_XCC_PN	3	
BPOS	3	
BPOS_A	4?	
BPOS_A_PN	4	
BPOS_PN	3	
BPOS_XCC	3	
BPOS_XCC_A	4?	
BPOS_XCC_A_PN	4	
BPOS_XCC_PN	3	

Instruction	Latency	Comments
BRGEZ	3	
BRGEZ_A	4?	
BRGEZ_A_PN	4	
BRGEZ_PN	3	
BRGZ	3	
BRGZ_A	4?	
BRGZ_A_PN	4	
BRGZ_PN	3	
BRLEZ	3	
BRLEZ_A	4	
BRLEZ_A_PN	4	
BRLEZ_PN	3	
BRLZ	3	
BRLZ_A	3	
BRLZ_A_PN	3?	
BRLZ_PN	3	
BRNZ	3	
BRNZ_A	4	
BRNZ_A_PN	4	
BRNZ_PN	3	
BRZ	3	
BRZ_A	4	
BRZ_A_PN	4	
BRZ_PN	3	
BVC	3	
BVC_A	4?	
BVC_A_PN	4	
BVC_PN	3	
BVC_XCC	3	
BVC_XCC_A	4	

TABLE B-2 Instruction Latencies *(6 of 14)*

Instruction	Latency	Comments
BVC_XCC_A_PN	4?	
BVC_XCC_PN	3	
BVS	3	
BVS_A	4	
BVS_A_PN	4	
BVS_PN	3	
BVS_XCC	3	
BVS_XCC_A	4	
BVS_XCC_A_PN	4	
BVS_XCC_PN	3	
CASA[PASI]	39	performed in L2
CASXA[PASI]	39	performed in L2
FABSd	8	
FABSs	21	
FADDd	26	
FADDs	26	
FBA	3	
FBA_A	4?	
FBA_A_PN	4	
FBA_PN	3	
FBE	3	
FBE_A	4	
FBE_A_PN	4	
FBE_PN	3	
FBG	3	
FBG_A	4	
FBG_A_PN	4?	
FBG_PN	3	
FBGE	3	
FBGE_A	4	

TABLE B-2 Instruction Latencies *(7 of 14)*

Instruction	Latency	Comments
FBGE_A_PN	4	
FBGE_PN	3	
FBL	3	
FBL_A	4?	
FBL_A_PN	4	
FBL_PN	3	
FBLE	3	
FBLE_A	4	
FBLE_A_PN	4	
FBLE_PN	3	
FBLG	3	
FBLG_A	4	
FBLG_A_PN	4	
FBLG_PN	3	
FBN	3?	
FBN_A	4	
FBN_A_PN	4	
FBN_PN	3	
FBNE	3	
FBNE_A	4	
FBNE_A_PN	4?	
FBNE_PN	3	
FBUE	3	
FBUE_A	4	
FBUE_A_PN	4	
FBUE_PN	3	
FBUG	3	
FBUG_A	4	
FBUG_A_PN	4	
FBUG_PN	3	

TABLE B-2   Instruction Latencies *(8 of 14)*

Instruction	Latency	Comments
FBUGE	3	
FBUGE_A	4	
FBUGE_A_PN	4	
FBUGE_PN	3	
FBUL	3	
FBUL_A	4	
FBUL_A_PN	4	
FBUL_PN	3	
FBULE	3	
FBULE_A	4	
FBULE_A_PN	4	
FBULE_PN	3	
FDIVd	83	
FDIVs	54	
FdTOi	25	
FdTOs	25	
FdTOx	25	
FiTOd	25	
FiTOs	26	
FMOVd	8	
FMOVDA		
FMOVDE	8	
FMOVDG	8	
FMOVDGE	8	
FMOVDL	8	
FMOVDLE	8	
FMOVDLG	8	
FMOVDN	8	
FMOVDNE	8	
FMOVDO	8	

TABLE B-2   Instruction Latencies *(9 of 14)*

Instruction	Latency	Comments
FMOVDU	8	
FMOVDUE	8	
FMOVDUG	8	
FMOVDUGE	8	
FMOVDUL	8	
FMOVDULE	8	
FMOVs	8	
FMOVSA	8	
FMOVSE	8?	
FMOVSG	8?	
FMOVSGE	8?	
FMOVSL	8?	
FMOVSLE	8?	
FMOVSLG	8?	
FMOVSN	8	
FMOVSNE	8	
FMOVSO	8?	
FMOVSU	8	
FMOVSUE	8	
FMOVSUG	8	
FMOVSUGE	8	
FMOVSUL	8	
FMOVSULE	8	
FMULd	29	
FMULs	29	
FNEGd	8	
FNEGs	8	
FsMULd	29	
FsTOd	25	
FsTOi	25	

TABLE B-2   Instruction Latencies *(10 of 14)*

Instruction	Latency	Comments
FsTOx	25	
FSUBd	26	
FSUBs	26	
FxTOd	26	
FxTOs	26	
LD_FP	9	
LDFSR	9	
LDD_FP	9?	
LDD_FPD	9	
LDDFA^{PASI}	9	
LDSB	22	performed in L2
LDSBA	21	performed in L2
LDSH	3	
LDSHA^{PASI}	3	
LDSTUB	37	performed in L2
LDSTUBA^{PASI}	37	performed in L2
LDSW	3	
LDSWA	3	
LDUB	3	
LDUBA	3	
LDUH	3	
LDUHA^{PASI}	3	
LDUW	3	
LDUWA^{PASI}	3	
LDX	3	
LDX_FSR	27	
LDXA^{PASI}	3	
MOVA	1	
MOVA_FCC	1	
MOVCC	1	

TABLE B-2 Instruction Latencies *(11 of 14)*

Instruction	Latency	Comments
MOVCS	1	
MOVE	1	
MOVE_FCC	1	
MOVG	1?	
MOVG_FCC	1	
MOVGE	1	
MOVGE_FCC	1	
MOVGU	1?	
MOVL	1	
MOVL_FCC	1	
MOVLE	1	
MOVLE_FCC	1	
MOVLEU	1	
MOVLG_FCC	1	
MOVN	1	
MOVN_FCC	1	
MOVNE	1	
MOVNE_FCC	1	
MOVNEG	1?	
MOVO_FCC	1	
MOVPOS	1	
MOVRE	1	
MOVRGEZ	1	
MOVRGZ	1	
MOVRLEZ	1	
MOVRLZ	1	
MOVRNE	1	
MOVU_FCC	1	
MOVUE_FCC	1	
MOVUG_FCC	1	

TABLE B-2   Instruction Latencies *(12 of 14)*

Instruction	Latency	Comments
MOVUGE_FCC	1	
MOVUL_FCC	1	
MOVULE_FCC	1	
MOVVC	1	
MOVVS	1	
MULSCC	7	
MULX	11	
OR	1	
ORcc	1	
ORN	1	
ORNcc	1	
RD_CCR	4	
RDASI	4	
RD_FPRS	4	
RD_Y	4	
SDIV^D	72	
SDIVcc^D	72	
SDIVX	72	
SETHI	1	
SLL	1	
SLLX	1	
SMUL^D	11	
SMULcc^D	11	
SRA	1?	
SRAX	1	
SRL	1	
SRLX	1	
STFA^{PASI}	8	
ST_FSR	8	
STFA^{PASI}	8	

TABLE B-2 Instruction Latencies *(13 of 14)*

Instruction	Latency	Comments
STB	1	
STBA	4	
STDF	8	
STDA_FP	8?	
STDA_FP_ASI	8	
STH	1	
STHA	4	
STW	1?	
STWA	1?	
STX	1	
STX_FSR	8	
STXA	1-? (4-?)?	varies, depending on ASI
SUB	1	
SUBC	1	
SUBcc	1	
SUBCcc	1	
SWAP^D	49	performed in L2
SWAPA$^{D, P_{ASI}}$	37	performed in L2
TADDcc	1?	
TADDccTV^D	1	
TSUBcc	1	
TSUBccTV^D	1	
UDIV^D	72	
UDIVcc^D	72?	
UDIVX	72	
UMUL^D	11	
UMULcc^D	11	
WR_CCR	9	
WRASI	9	
WR_FPRS	9	

Instruction	Latency	Comments
XNOR	1	
XNORcc	1	
XOR	1	
XORcc	1	

B.4   Grouping Rules

Each physical cores in UltraSPARC T1 are single-issue, so there are no grouping rules for UltraSPARC T1.

B.5   Floating-Point Operations

UltraSPARC T1 supports hardware floating-point operations, but since one floating-point unit (FPU) is shared among 8 physical cores (32 strands), there are limitations on dispatch of floating-point instructions to the FPU. Each physical processor core (four strands) can have a single floating-point instruction outstanding at any given time. For the purpose of this restriction, floating-point instructions include floating-point operations, VIS floating-point operations, floating-point loads and stores, and block loads and stores.

B.6   Hyperprivileged Execution

In UltraSPARC T1, the target of a branch will be fetched, even if it is annulled by a second branch or a trap in the delay slot. When operating in hypervisor mode with translation disabled, this means the target of every branch should be a valid physical address unless translation is being reenabled by the delay slot.

There should be five 32-bit words of padding past the last valid instruction executed with instruction address translation disabled. This ensures that instruction prefetching will use valid physical addresses.

B.7   Synchronization

UltraSPARC T1 has two varieties of instructions for synchronization: memory barriers and flush. The following memory barrier instructions ensure that any load, store, or atomic memory operation issued after it take effect after all memory operations issued before it:

■ MEMBAR with mmask{1} = 1 (MEMBAR #StoreLoad)

■ MEMBAR with cmask{1} = 1 (MEMBAR #MemIssue)

■ MEMBAR with cmask{2} = 1 MEMBAR #Sync)

All other types of membar instructions are treated as NOPs, since they are implied by the TSO memory ordering protocol followed by UltraSPARC T1.

However, the memory barriers do not guarantee that the instruction caches on UltraSPARC T1 have become consistent with the preceding memory operations. A FLUSH instruction guarantees that in addition to the preceding memory operations taking effect in the global memory system, all the instruction caches on UltraSPARC T1 are consistent with these operations. It also ensures that the instruction fetch buffer for the strand issuing the flush has become consistent with the preceding memory operations.

Thus, when one strand is modifying the instructions of another, the "producer" strand should

1. Complete all necessary modifications

2. Issue a FLUSH

3. Signal completion to the "consumer" strand

    Completion may be signalled by a store/atomic instruction which modifies a predetermined location, or by issuing an interrupt to the consumer strand.

The consumer strand at this point should make sure that its instruction fetch buffer (of size 2 entries) becomes consistent with the global memory system. This can be done by either

1. Issuing a branch to the modified location; or

2. Issuing a flush instruction; or

3. Waiting for a two-instruction gap, to allow the two instructions in the fetch buffer to drain.

In the case of a branch, the delay slot is not guaranteed to be consistent with global memory. The branch is a better option than flush for high performance.

Note that the HALT instruction is not meant to be a synchronization instruction and should *not* be used as such. For example, the following code, which uses halt to make sure func_A executes before func_B, may cause T0 to hang:

T0	T1
st *X*	ld *X*
halt	do_func_A
do_func_B	intr T0

Opcode Maps

This appendix contains the UltraSPARC T1 instruction opcode maps.

Opcodes marked with a dash (—) are reserved; an attempt to execute a reserved opcode causes a trap unless the opcode is an implementation-specific extension to the instruction set.

In this appendix, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 5-1 on page 22.

In the tables in this appendix, *reserved* (—) and shaded entries indicate opcodes that are not implemented in the UltraSPARC T1 processor.

Shading	Meaning
	An attempt to execute opcode will cause an *illegal_instruction* exception.
	An attempt to execute opcode will cause an *fp_exception_other* exception with FSR.ftt = 3 (unimplemented_FPop).

TABLE C-1   op{1:0}

op {1:0}			
0	**1**	**2**	**3**
Branches and SETHI *See* TABLE C-2.	CALL	Arithmetic and Miscellaneous *See* TABLE C-3	Loads/Stores *See* TABLE C-4

TABLE C-2   op2{2:0} (op = 0)

op2 {2:0}							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc – *See* TABLE C-7	Bicc[D]– *See* TABLE C-7	BPr – *See* TABLE C-8	SETHI NOP[†]	FBPfcc – *See* TABLE C-7	FBfcc[D]– *See* TABLE C-7	—

[†]rd = 0, imm22 = 0

The ILLTRAP and *reserved* (—) encodings generate an *illegal_instruction* trap.

TABLE C-3 op3{5:0} (op = 2) *(1 of 3)*

		op3 {5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**0**	ADD	ADDcc	TADDcc	WRY[D] (rd = 0) —— (rd=1) WRCCR (rd = 2) WRASI (rd = 3) —— (rd = 4, 5) SIR[H] (rd = 5, rd = 0, rd = 1) WRFPRS (rd = 6) —— (rd = 7–14) —— (rd = 15 and (rs1 > 0 or i = 0)) WRPCR[P] (rd = 16, rd = 0) —— (rd =16, rd = 1) WRPIC (rd = 17, rd = 0) —— (rd = 17, rd = 1) —— (rd = 18) WRGSR (rd = 19) WRSOFTINT_SET[P] (rd = 0) WRSOFTINT_CLR[P] (rd = 21) WRSOFTINT[P] (rd = 22) WRTICK_CMPR[P] (rd = 23) WRSTICK[H] (rd = 24) WRSTICK_CMPR (rd = 25) WR %asr26 (rd = 26, rd = 1) —— (rd = 26, rd = 0) —— (rd=27–31))
	1	AND	ANDcc	TSUBcc	SAVED[P] (fcn = 0), RESTORED[P] (fcn = 1) — (fcn > 1)
	2	OR	ORcc	TADDccTV[D]	WRPR[P] — (rd = 15, 17–31)
	3	XOR	XORcc	TSUBccTV[D]	WRHPR[H] — (rs1 = 2, 4, 7–30)
	4	SUB	SUBcc	MULScc[D]	FPop1 – *See* TABLE C-5
	5	ANDN	ANDNcc	SLL (x = 0), SLLX (x = 1)	FPop2 – *See* TABLE C-6
	6	ORN	ORNcc	SRL (x = 0), SRLX (x = 1)	IMPDEP1 (VIS) – *See* TABLE C-12
	7	XNOR	XNORcc	SRA (x = 0), SRAX (x = 1)	IMPDEP2

TABLE C-3   op3{5:0} (op = 2) *(2 of 3)*

		op3 {5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**8**	ADDC	ADDCcc	RDY^D (rs1 = 0, i = 0)) ── (rs1 = 0, i = 1) ── (rs1 = 1) RDCCR (rs1 = 2, i = 0)) ── (rs1 = 2, i = 1) RDASI (rs1 = 3, i = 0) ── (rs1 = 3, i = 1) RDTICK$^{P_{npt}}$ (rs1 = 4, i = 0) ── (rs1 = 4, i = 1) RDPC (rs1 = 5, i = 0) ── (rs1 = 5, i = 1) RDFPRS (rs1 = 6, i = 0) ── (rs1 = 6, i = 1) ── (rs1 = 7–14) MEMBAR (rs1 = 15, rd=0, i = 1) STBAR^D (rs1 = 15, rd=0, i = 0) ── (rs1 = 15, rd > 0) RDPCR^P (rs1 = 16) RDPIC (rs1 = 17) ── (rs1 = 18) RDGSR (rs1 = 19, i = 0) ── (rs1 = 19, i = 1) ── (rs1 = 20, 21) RDSOFTINT^P (rs1 = 22, i = 0) ── (rs1 = 22, i = 1) RDTICK_CMPR^P (rs1 = 23, i = 0) ── (rs1 = 23, i = 1) RDSTICK^P (rs1 = 24, i = 0) ── (rs1 = 24, i = 1) RDSTICK_CMPR^P (rs1 = 25, i = 0) ── (rs1 = 25, i = 1) rd %asr26 (rs1 = 26) ── (rs1 = 27 - 31)	JMPL

TABLE C-3 op3{5:0} (op = 2) *(3 of 3)*

		op3 {5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	9	MULX	—	RDHPR[H] — (rs1 = 2, 4, 7 - 30)	RETURN
	A	UMUL[D]	UMULcc[D]	RDPR[P] — (rs1 = 15, 17 - 30)	Tcc {(i = 0 and inst{10:5} = 0) or i = 1 and inst{10:8} = 0)}–*See* TABLE C-7 *and* TABLE C-11. — {((i = 0 and inst{10:5} > 0) or i = 1 and inst{10:8} > 0)}
	B	SMUL[D]	SMULcc[D]	FLUSHW	FLUSH
	C	SUBC	SUBCcc	MOVcc *See* TABLE C-9	SAVE
	D	UDIVX	—	SDIVX	RESTORE
	E	UDIV[D]	UDIVcc[D]	POPC (rs1 = 0) — (rs1 > 0)	DONE[P] (fcn = 0) RETRY[P] (fcn = 1) — (fcn > 1)
	F	SDIV[D]	SDIVcc[D]	MOVr *See* TABLE C-8	—

Shaded and the reserved (—) opcodes cause an illegal_instruction trap.

TABLE C-4 op3{5:0} (op = 3)

		op3{5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**0**	LDUW	LDUWA[PASI]	LDF	LDFA[PASI]
	1	LDUB	LDUBA[PASI]	LDFSR[D], LDXFSR — (rd > 1)	—
	2	LDUH	LDUHA[PASI]	LDQF	LDQFA[PASI]
	3	LDD[D] — (rd odd)	LDDA[D, PASI] — (rd odd)	LDDF	LDDFA[PASI] See 8.6.4 XREF
	4	STW	STWA[PASI]	STF	STFA[PASI]
	5	STB	STBA[PASI]	STFSR[D], STXFSR — (rd > 1)	—
	6	STH	STHA[PASI]	STQF	STQFA[PASI]
	7	STD[D] — (rd odd)	STDA[PASI] — (rd odd)	STDF	STDFA[PASI] See 8.6.4 XREF
	8	LDSW	LDSWA[PASI]	—	—
	9	LDSB	LDSBA[PASI]	—	—
	A	LDSH	LDSHA[PASI]	—	—
	B	LDX	LDXA[PASI]	—	—
	C	—	—	—	CASA[PASI]
	D	LDSTUB	LDSTUBA[PASI]	PREFETCH — (fcn = 5–15)	PREFETCHA[PASI] — (fcn = 5–15)
	E	STX	STXA[PASI]	—	CASXA[PASI]
	F	SWAP[D]	SWAPA[D, PASI]	—	—

LDQF, LDQFA, STQF, STQFA, and the *reserved* (—) opcodes cause an *illegal_instruction* trap.

TABLE C-5 opf$\{8:0\}$ (op $= 2$, op3 $= 34_{16} =$ FPop1)

opf$\{8:3\}$	opf$\{2:0\}$
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
00_{16}	—	FMOVs	FMOVd	FMOVq	—	FNEGs	FNEGd	FNEGq
01_{16}	—	FABSs	FABSd	FABSq	—	—	—	—
02_{16}	—	—	—	—	—	—	—	—
03_{16}	—	—	—	—	—	—	—	—
04_{16}	—	—	—	—	—	—	—	—
05_{16}	—	FSQRTs	FSQRTd	FSQRTq	—	—	—	—
06_{16}	—	—	—	—	—	—	—	—
07_{16}	—	—	—	—	—	—	—	—
08_{16}	—	FADDs	FADDd	FADDq	—	FSUBs	FSUBd	FSUBq
09_{16}	—	FMULs	FMULd	FMULq	—	FDIVs	FDIVd	FDIVq
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	—	—	—	—	—	—	—	—
$0D_{16}$	—	FsMULd	—	—	—	—	FdMULq	—
$0E_{16}$	—	—	—	—	—	—	—	—
$0F_{16}$	—	—	—	—	—	—	—	—
10_{16}	—	FsTOx	FdTOx	FqTOx	FxTOs	—	—	—
11_{16}	FxTOd	—	—	—	FxTOq	—	—	—
12_{16}	—	—	—	—	—	—	—	—
13_{16}	—	—	—	—	—	—	—	—
14_{16}	—	—	—	—	—	—	—	—
15_{16}	—	—	—	—	—	—	—	—
16_{16}	—	—	—	—	—	—	—	—
17_{16}	—	—	—	—	—	—	—	—
18_{16}	—	—	—	—	FiTOs	—	FdTOs	FqTOs
19_{16}	FiTOd	FsTOd	—	FqTOd	FiTOq	FsTOq	FdTOq	—
$1A_{16}$	—	FsTOi	FdTOi	FqTOi	—	—	—	—
$1B_{16}$–$3F_{16}$	—	—	—	—	—	—	—	—

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with FSR.ftt = 3 (unimplemented_FPop).

TABLE C-6   opf{8:0} (op = 2, op3 = 35_{16} = FPop2)

opf{8:4}	0	1	2	3	4	5	6	7	8–F
				opf{3:0}					
00	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	†	†	†	—
01	—	—	—	—	—	—	—	—	—
02	—	—	—	—	—	FMOVsZ	FMOVdZ	FMOVqZ	—
03	—	—	—	—	—	—	—	—	—
04	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVsLEZ	FMOVdLEZ	FMOVqLEZ	—
05	—	FCMPs	FCMPd	FCMPq	—	FCMPEs	FCMPEd	FCMPEq	—
06	—	—	—	—	—	FMOVsLZ	FMOVdLZ	FMOVqLZ	—
07	—	—	—	—	—	—	—	—	—
08	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
09	—	—	—	—	—	—	—	—	—
0A	—	—	—	—	—	FMOVsNZ	FMOVdNZ	FMOVqNZ	—
0B	—	—	—	—	—	—	—	—	—
0C	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVsGZ	FMOVdGZ	FMOVqGZ	—
0D	—	—	—	—	—	—	—	—	—
0E	—	—	—	—	—	FMOVsGEZ	FMOVdGEZ	FMOVqGEZ	—
0F	—	—	—	—	—	—	—	—	—
10	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
11–17	—	—	—	—	—	—	—	—	—
18	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
19–1F	—	—	—	—	—	—	—	—	—

† Reserved variation of FMOVr

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with FSR.ftt = 3 (unimplemented_FPop).

TABLE C-7 cond{3:0}

		BPcc	Bicc[D]	FBPfcc	FBfcc[D]	Tcc
		op = 0 op2 = 1	op = 0 op2 = 2	op = 0 op2 = 5	op = 0 op2 = 6	op = 2 op3 = $3a_{16}$
cond {3:0}	0	BPN	BN[D]	FBPN	FBN[D]	TN
	1	BPE	BE[D]	FBPNE	FBNE[D]	TE
	2	BPLE	BLE[D]	FBPLG	FBLG[D]	TLE
	3	BPL	BL[D]	FBPUL	FBUL[D]	TL
	4	BPLEU	BLEU[D]	FBPL	FBL[D]	TLEU
	5	BPCS	BCS[D]	FBPUG	FBUG[D]	TCS
	6	BPNEG	BNEG[D]	FBPG	FBG[D]	TNEG
	7	BPVS	BVS[D]	FBPU	FBU[D]	TVS
	8	BPA	BA[D]	FBPA	FBA[D]	TA
	9	BPNE	BNE[D]	FBPE	FBE[D]	TNE
	A	BPG	BG[D]	FBPUE	FBUE[D]	TG
	B	BPGE	BGE[D]	FBPGE	FBGE[D]	TGE
	C	BPGU	BGU[D]	FBPUGE	FBUGE[D]	TGU
	D	BPCC	BCC[D]	FBPLE	FBLE[D]	TCC
	E	BPPOS	BPOS[D]	FBPULE	FBULE[D]	TPOS
	F	BPVC	BVC[D]	FBPO	FBO[D]	TVC

TABLE C-8 Encoding of rcond{2:0} Instruction Field

		BPr	MOVr	FMOVr
		op = 0 op2 = 3	op = 2 op3 = $2f_{16}$	op = 2 op3 = 35_{16}
rcond {2:0}	0	—	—	—
	1	BRZ	MOVRZ	FMOVRZ
	2	BRLEZ	MOVRLEZ	FMOVRLEZ
	3	BRLZ	MOVRLZ	FMOVRLZ
	4	—	—	—
	5	BRNZ	MOVRNZ	FMOVRNZ
	6	BRGZ	MOVRGZ	FMOVRGZ
	7	BRGEZ	MOVRGEZ	FMOVRGEZ

TABLE C-9   cc / opf_cc Fields (MOVcc and FMOVcc)

opf_cc			
cc2	**cc1**	**cc0**	**Condition Code Selected**
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc
1	0	1	—
1	1	0	xcc
1	1	1	—

TABLE C-10   cc Fields (FBPfcc, FCMP, and FCMPE)

cc1	**cc0**	**Condition Code Selected**
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

TABLE C-11   cc Fields (BPcc and Tcc)

cc1	**cc0**	**Condition Code Selected**
0	0	icc
0	1	—
1	0	xcc
1	1	—

TABLE C-12 VIS Opcodes op = 2, op3 = 36_{16} = IMPDEP1

		opf {3:0}							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
opf {8:4}	00	EDGE8	EDGE8N	EDGE8L	EDGE8LN	EDGE16	EDGE16N	EDGE16L	EDGE16LN
	01	ARRAY8		ARRAY16		ARRAY32			
	02	FCMPLE16		FCMPNE16		FCMPLE32		FCMPNE32	
	03		FMUL8X16		FMUL8X16AU		FMUL8X16AL	FMUL8SUX16	FMUL8ULX16
	04								
	05	FPADD16	FPADD16S	FPADD32	FPADD32S	FPSUB16	FPSUB16S	FPSUB32	FPSUB32S
	06	FZERO	FZEROS	FNOR	FNORS	FANDNOT2	FANDNOT2S	FNOT2	FNOT2S
	07	FAND	FANDS	FXNOR	FXNORS	FSRC1	FSRC1S	FORNOT2	FORNOT2S
	08	SHUTDOWN	SIAM						
	09..1F								

		opf {3:0}							
		8	**9**	**A**	**B**	**C**	**D**	**E**	**F**
opf {8:4}	00	EDGE32	EDGE32N	EDGE32L	EDGE32LN				
	01	ALIGNADDRESS	BMASK	ALIGNADDRESS_LITTLE					
	02	FCMPGT16		FCMPEQ16		FCMPGT32		FCMPEQ32	
	03	FMULD8SUX16	FMULD8ULX16	FPACK32	FPACK16		FPACKFIX	PDIST	
	04	FALIGNDATA			FPMERGE	BSHUFFLE	FEXPAND		
	05								
	06	FANDNOT1	FANDNOT1S	FNOT1	FNOT1S	FXOR	FXORS	FNAND	FNANDS
	07	FSRC2	FSRC2S	FORNOT1	FORNOT1S	FOR	FORS	FONE	FONES
	08								
	09..1F								

Note | An *illegal_instruction* exception is generated if the undefined or shaded opcodes in the IMPDEP1 space are used.

Instructions and Exceptions

The instructions supported by UltraSPARC T1 and the exceptions they generate are listed in the UltraSPARC Architecture 2005 specification.

IEEE 754 Floating Point Support

UltraSPARC T1 conforms to the SPARC V9 Appendix B (IEEE Std 754-1985 Requirements for SPARC-V9) recommendations.

Note | UltraSPARC T1 detects tininess before rounding.

E.1   Special Operand Handling

The UltraSPARC T1 FPU provides full hardware support for subnormal operands and results. Unlike UltraSPARC I/II and UltraSPARC III, UltraSPARC T1 will never generate an unfinished_FPop trap type. Also, unlike UltraSPARC I/II and UltraSPARC III, UltraSPARC T1 does not implement a nonstandard floating-point mode. The ns bit of the FSR is always read as 0, and writes to it are ignored.

The FPU generates +inf, –inf, +largest number, –largest number (depending on round mode) for overflow cases for multiply, divide, and add operations.

For higher-to-lower precision conversion instructions {FDTOS}:

- overflow, underflow, and inexact exceptions can be raised.
- overflow is treated the same way as an unrounded add result; depending on the round mode, we will either generate the properly signed infinity or largest number.
- underflow will produce a signed zero, smallest number, or subnormal result.

For conversion to integer instructions {F(s,d)TOi, F(s,d)TOx}: UltraSPARC T1 follows SPARC V9 appendix B.5, pg 246.

For NaN's: UltraSPARC T1 Follows SPARC V9 appendix B.2 (particularly Table 27) and B.5, pg 244-246.

- Please note that Appendix B applies to those instructions listed in IEEE 754 section 5: "All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the sqrt, find the remainder, round to integer in fp format, convert between different fp formats, convert

between fp and integer formats, convert binary<->decimal, and compare. Whether copying without change of format is considered an operation is an implementation option."

- The instructions involving copying/moving of fp data (FMOV, FABS, and FNEG) will follow earlier UltraSPARC implementations by doing the appropriate sign bit transformation but will not cause an invalid exception nor do a rs2 = SNaN to rd = QNaN transformation.

- Following UltraSPARC I/II implementations, all Fpops as defined in V9 will update cexc. All other instructions will leave cexc unchanged.

- Following SPARC V9 Manual 5.1.7.6, 5.1.7.8, 5.1.7.9, and figures in 5.1.7.10 Overflow Result is defined as:

> If the appropriate trap enable masks are not set (FSR.ofm = 0 *and* FSR.nxm = 0), then set aexc and cexc overflow and inexact flags: FSR.ofa = 1, FSR.nxa = 1, FSR.ofc = 1, FSR.nxc = 1. No trap is generated.

> If any or both of the appropriate trap enable masks are set (FSR.ofm = 1 *or* FSR.nxm = 1), then only an IEEE overflow trap is generated: FSR.ftt = 1. The particular cexc bit that is set diverges from UltraSPARC I/II to follow the SPARC V9 section 5.1.7.9 errata:

> If FSR.ofm = 0 *and* FSR.nxm = 1, then FSR.nxc = 1.

> If FSR.ofm = 1, independent of FSR.nxm, then FSR.ofc = 1 and FSR.nxc = 0.

- Following SPARC V9 Manual 5.1.7.6, 5.1.7.8, 5.1.7.9, and figures in 5.1.7.10 Underflow Result is defined as:

> If the appropriate trap enable masks are not set (FSR.ufm = 0 *and* FSR.nxm = 0), then set aexc and cexc underflow and inexact flags: FSR.ufa = 1, FSR.nxa = 1, FSR.ufc = 1, FSR.nxc = 1. No trap is generated.

> If any or both of the appropriate trap enable masks are set (FSR.ufm = 1 *or* FSR.nxm = 1), then only an IEEE underflow trap is generated: FSR.ftt = 1. The particular cexc bit that is set diverges from UltraSPARC I/II to follow the SPARC V9 section 5.1.7.9 errata:

> If FSR.ufm = 0 *and* FSR.nxm = 1, then FSR.nxc = 1.

> If FSR.ufm = 1, independent of FSR.nxm, then FSR.ufc = 1 and FSR.nxc = 0.

The remainder of this section gives examples of special cases to be aware of that could generate various exceptions.

E.1.1 Infinity Arithmetic

Let "num" be defined as unsigned in the following tables.

E.1.1.1   One Infinity Operand Arithmetic

■ Do not generate exceptions

TABLE E-1   One Infinity Operations That Do Not Generate Exceptions

Cases
+inf plus +num = +inf
+inf plus -num = +inf
-inf plus +num = -inf
-inf plus -num = -inf
+inf minus +num = +inf
+inf minus -num = +inf
-inf minus +num = -inf
-inf minus -num = -inf
+inf multiplied by +num = +inf
+inf multiplied by -num = -inf
-inf multiplied by +num = -inf
-inf multiplied by -num = +inf
+inf divided by +num = +inf
+inf divided by -num = -inf
-inf divided by +num = -inf
-inf divided by -num = +inf
+num divided by +inf = +0
+num divided by -inf = -0
-num divided by +inf = -0
-num divided by -inf = +0
fstod, fdtos (+inf) = +inf
fstod, fdtos (-inf) = -inF
+inf divided by +0 = +inf
+inf divided by -0 = -inf
-inf divided by +0 = -inf
-inf divided by -0 = +inf

TABLE E-1   One Infinity Operations That Do Not Generate Exceptions *(Continued)*

Cases
Any arithmetic operation involving infinity as 1 operand and a QNaN as the other operand: SPARC V9 B.2.2 Table 27 (+/- inf) OPERATOR (QNaN2) = QNaN2 (QNaN1) OPERATOR (+/- inf) = QNaN1
Compares when other operand is not a NaN treat infinity just like a regular number: +inf = +inf, +inf > anything else; -inf = -inf, -inf < anything else. Affects following instructions: V9 fp compares (rs1 and/or rs2 could be +/- inf): * FCMPE * FCMP
Compares when other operand is a QNaN, SPARC V9 A.13, B.2.1; fcc value = unordered = 2′b11 fcmp(s/d) (+/- inf) with (QNaN2) - no invalid exception fcmp(s/d) (QNaN1) with (+/- inf) - no invalid exception

- ■ Could generate exceptions

TABLE E-2   One Infinity Operations That Could Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) iF tem is cleared
SPARC V9 Appendix B.5[1]	IEEE_754 7.1	
F{s,d}TOi (+inf) = invalid	IEEE_754 invalid	2^{31}-1
F{s,d}TOx (+inf) = invalid	IEEE_754 invalid	2^{63}-1
F{s,d}TOi (-inf) = invalid	IEEE_754 invalid	-2^{31}
F{s,d}TOx (-inf) = invalid	IEEE_754 invalid	-2^{63}
SPARC V9 B.2.2	IEEE_754 7.1	(No NaN operand result)
+inf multiplied by +0 = invalid	IEEE_754 invalid	QNaN
+inf multiplied by -0 = invalid	IEEE_754 invalid	QNaN
-inf multiplied by +0 = invalid	IEEE_754 invalid	QNaN
-inf multiplied by -0 = invalid	IEEE_754 invalid	QNaN

TABLE E-2   One Infinity Operations That Could Generate Exceptions *(Continued)*

Cases	Possible Exception	Result (in addition to accrued exception) iF tem is cleared
SPARC V9 B.2.2 Table 27[2] Any arithmetic operation involving infinity as 1 operand and a SNaN as the other operand except copying/moving data	IEEE_754 7.1	(One operand, a SNaN)
(+/- inf) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (+/- inf)	IEEE_754 invalid	QSNaN1
SPARC V9 A.13, B.2.1[2] Any compare operation involving infinity as 1 operand and a SNaN as the other operand:	IEEE_754 7.1	
FCMP(s/d) (+/- inf) with (SNaN2)	IEEE_754 invalid	fcc value = unordered = 2'b11
FCMP(s/d) (SNaN1) with (+/- inf)	IEEE_754 invalid	fcc value = unordered = 2'b11
FCMPE(s/d) (+/- inf) with (SNaN2)	IEEE_754 invalid	fcc value = unordered = 2'b11
FCMPE(s/d) (SNaN1) with (+/- inf)	IEEE_754 invalid	fcc value = unordered = 2'b11
SPARC V9 A.13[2] Any compare & generate exception operation involving infinity as 1 operand and a QNaN as the other operand:	IEEE_754 7.1	
FCMPE(s/d) (+/- inf) with (QNaN2)	IEEE_754 invalid	fcc value = unordered = 2'b11
FCMPE(s/d) (QNaN1) with (+/- inf)	IEEE_754 invalid	fcc value = unordered = 2'b11

1. Similar invalid exceptions also included in SPARC V9 B.5 are generated when the source operand is a NaN(QNaN or SNaN) or a re-sulting number that cannot fit in 32b[64b] integer format: (large positive argument >= 2^{31}[2^{63}] or large negative argument <= -(2^{31} + 1)[-(2^{63}+1)]

2. Note that in the IEEE 754 standard, infinity is an exact number; so this exception could also applies to non-infinity operands as well. Also note that the invalid exception and SNaN to QNaN transformation does not apply to copying/moving fpops (fmov,fabs,fneg).

E.1.1.2 Two Infinity Operand Arithmetic

■ Do not generate exceptions

TABLE E-3 Two Infinity Operations That Do Not Generate Exceptions

Cases
+inf plus +inf = +inf -inf plus -inf = -inf
+inf minus -inf = +inf -inf minus +inf = -inf
+inf multiplied by +inf = +inf +inf multiplied by -inf = -inf -inf multiplied by +inf = -inf -inf multiplied by -inf = +inf
Compares treat infinity just like a regular number: +inf = +inf, +inf > anything else; -inf = -inf, -inf < anything else. Affects following instructions: V9 fp compares (rs1 and/or rs2 could be +/- inf): * FCMPE * FCMP

■ Could generate exceptions

TABLE E-4 Two Infinity Operations That Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 B.2.2 +inf plus -inf = invalid -inf plus +inf = invalid	IEEE_754 7.1 IEEE_754 invalid IEEE_754 invalid	(No NaN operand result) QNaN QNaN
+inf minus +inf = invalid -inf minus -inf = invalid	IEEE_754 invalid IEEE_754 invalid	QNaN QNaN
SPARC V9 B.2.2 +inf divided by +inf = invalid +inf divided by -inf = invalid -inf divided by +inf = invalid -inf divided by -inf = invalid	IEEE_754 7.1 IEEE_754 invalid IEEE_754 invalid IEEE_754 invalid IEEE_754 invalid	(No NaN operand result) QNaN QNaN QNaN QNaN

E.1.2 Zero Arithmetic

TABLE E-5 Zero Arithmetic Operations That Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 B.2.2 & 5.1.7.10.4	IEEE_754 7.1	(No NaN operand
+0 divided by +0 = invalid	IEEE_754 invalid	result)
+0 divided by -0 = invalid	IEEE_754 invalid	QNaN
-0 divided by +0 = invalid	IEEE_754 invalid	QNaN
-0 divided by -0 = invalid	IEEE_754 invalid	QNaN
		QNaN
SPARC V9 5.1.7.10.4	IEEE_754 7.2	
+num divided by +0 = divide by zero	IEEE_754 div_by_zero	+inf
+num divided by -0 = divide by zero	IEEE_754 div_by_zero	-inf
-num divided by +0 = divide by zero	IEEE_754 div_by_zero	-inf
-num divided by -0 = divide by zero	IEEE_754 div_by_zero	+inf
SPARC V9 B.2.2 Table 27[1]	IEEE_754 7.1	(One operand, a SNaN)
Any arithmetic operation involving zero as 1 operand and a SNaN as the other operand except copying/moving		
data	IEEE_754 invalid	QSNaN2
(+/- 0) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN1
(SNaN1) OPERATOR (+/- 0)		

1.In this context, 0 is again another exact number; so this exception could also applies to non-zero operands as well. Also note that the invalid exception and SNaN to QNaN transformation does not apply to copying/moving data instructions (FMOV, FABS, FNEG)

TABLE E-6 Interesting Zero Arithmetic Sign Result Case

Cases
+0 plus -0 = +0 for all round modes except round to -infinity where the result is -0.

E.1.3 NaN Arithmetic

- Do not generate exceptions

TABLE E-7 NaN Arithmetic Operations that do not generate exceptions

Cases

SPARC V9 B.2.1: Fp convert to wider NaN transformation
FsTOd (QNaN2) = QNaN2 widened
    FsTOd(0x7fd10000) = 0x7ffa2000 8′h0
    FsTOd(0xffd10000) = 0xfffa2000 8′h0

SPARC V9 B.2.1: Fp convert to narrower NaN transformation
FdTOs (QNaN2) = QNaN2 narrowed
    FdTOs(0x7ffa2000 8′h0) = 0x7fd1000
    FdTOs(0xfffa2000 8′h0) = 0xffd1000

SPARC V9 B.2.2 Table 27 Any non-compare arithmetic operations --result takes sign of QNaN pass through operand. (+/- num) OPERATOR (QNaN2) = QNaN2 (QNaN1) OPERATOR (+/- num) = QNaN1 (QNaN1) OPERATOR (QNaN2) = QNaN2

■ Could generate exceptions

TABLE E-8   NaN Arithmetic Operations That Could Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 B.2.1: Fp convert to wider NaN transformation FsTOd (SNaN2) = QSNaN2 widened	IEEE_754 7.1	
FsTOd(0x7f910000) = 0x7ffa2000 8'h0 FsTOd(0xff910000) = 0xfffa2000 8'h0	IEEE_754 invalid	QSNaN2 widened
SPARC V9 B.2.1: Fp convert to narrower NaN transformation FdTos (SNaN2) = QSNaN2 narrowed	IEEE_754 7.1	
FdTos(0x7ff22000 8'h0) = 0x7fd1000 FdTos(0xfff22000 8'h0) = 0xffd1000	IEEE_754 invalid	QSNaN2 narrowed
SPARC V9 B.2.2 Table 27 Any non-compare arithmetic operations except copying/ moving (fmov, fabs, fneg)	IEEE_754 7.1	
(+/- num) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (+/- num)	IEEE_754 invalid	QSNaN1
(SNaN1) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(QNaN1) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (QNaN2)	IEEE_754 invalid	QSNaN1
SPARC V9 Appendix B.5	IEEE_754 7.1	
F{s,d}TOi (+QNaN) = invalid	IEEE_754 invalid	2^{31}-1
F{s,d}TOi (+SNaN) = invalid	IEEE_754 invalid	2^{31}-1
F{s,d}TOx (+QNaN) = invalid	IEEE_754 invalid	2^{63}-1
F{s,d}TOx (+SNaN) = invalid	IEEE_754 invalid	2^{63}-1
F{s,d}TOi (-QNaN) = invalid	IEEE_754 invalid	-2^{31}
F{s,d}TOi (-SNaN) = invalid	IEEE_754 invalid	-2^{31}
F{s,d}TOx (-QNaN) = invalid	IEEE_754 invalid	-2^{63}
F{s,d}TOx (-SNaN) = invalid	IEEE_754 invalid	-2^{63}

E.1.4   Special Inexact Exceptions

UltraSPARC T1 Follows SPARC V9 5.1.7.10.5 (IEEE_754 Section 7.5) and sets FSR_inexact whenever the rounded result of an operation differs from the infinitely precise unrounded result.

Additionally, there are a few special cases to be aware of:

TABLE E-9   Fp <-> Int Conversions With Inexact Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 A.14: Fp convert to 32b integer when source operand lies between $-(2^{31}-1)$ and 2^{31}, but is not exactly an integer FsTOi, FdTOi	IEEE_754 7.5	
	IEEE_754 inexact	An integer number
SPARC V9 A.14: Fp convert to 64b integer when source operand lies between $-(2^{63}-1)$ and 2^{63}, but is not exactly an integer FsTOx, FdTOx	IEEE_754 7.5	
	IEEE_754 inexact	An integer number
SPARC V9 A.15: Convert integer to fp format when 32b integer source operand magnitude is not exactly representable in single precision (23b mantissa). Note, even if the operand is $> 2^{24}-1$, if enough of its trailing bits are zeros, it may still be exactly representable. FiTOs	IEEE_754 7.5	
	IEEE_754 inexact	A SP number
SPARC V9 A.15: Convert integer to fp format when 64b integer source operand magnitude is not exactly representable in single precision (23b mantissa). Note, even if the operand is $> 2^{24}-1$, if enough of its trailing bits are zeros, it may still be exactly representable. FxTOs	IEEE_754 7.5	
	IEEE_754 inexact	A SP number
SPARC V9 A.15: Convert integer to fp format when 64b integer source operand magnitude is not exactly representable in double precision (52b mantissa). Note, even if the operand is $> 2^{53}-1$, if enough of its trailing bits are zeros, it may still be exactly representable. FxTOd	IEEE_754 7.5	
	IEEE_754 inexact	A DP number

E.2   Subnormal Handling

The UltraSPARC T1 FPU provides full hardware support for subnormal operands and results. Unlike UltraSPARC I/II and UltraSPARC III, UltraSPARC T1 will never generate an unfinished_FPop trap type.

Caches and Cache Coherency

Chapter Revision History

Date	By	Comment
28 Aug 03	Bill Bryg	Started outline of appendix.
22 Sep 03	Bill Croxton	Copied and reformatted Chapter 8, Cache and Memory Interactions, from USIIi User Manual.
25 Nov 03	J. Laudon	Initial changes for UltraSPARC T1
4 Jun 04	J. Laudon	More information on unit of coherence.
2 Aug 04	J. Laudon	Add cache index information.
20 Dec 04	J. Laudon	Better documentation of how to flush L2.
11 Feb 05	M. L. Nohr	Converted to UltraSPARC Architecture format

This appendix 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 the following subjects:

- Invalidation of one or more cache entries – when and how to do it
- Differences between cacheable and noncacheable accesses
- Ordering and synchronization of memory accesses
- Accesses to addresses that cause side effects (I/O accesses)
- Nonfaulting loads
- Cache sizes, associativity, replacement policy, etc.

F.1 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 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 by using ASI accesses. When ASI accesses are used, software must ensure that the flush is done on the same virtual core 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. This is done with a displacement flush (*Displacement Flushing* on page 316).

- **L2 cache**: Flush is needed for stable storage. Examples of stable storage include battery-backed memory and transaction logs. This is done with a displacement flush (see *Displacement Flushing* on page 316). Flushing the L2 cache flushes the corresponding blocks from the I- and D-caches because UltraSPARC T1 maintains inclusion between the L2 and L1 caches.

F.1.1 Displacement Flushing

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

> **Note** | Diagnostic ASI accesses to the L2 cache can be used to invalidate a line, but they are generally not an alternative to displacement flushing. Modified data in the L2 cache will not be written back to memory using these ASI accesses.

F.1.2 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.

F.1.3 Coherence Domains

Two types of memory operations are supported in UltraSPARC T1: 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 T1 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.

On UltraSPARC T1, a MEMBAR #Lookaside executes more efficiently than a MEMBAR #StoreLoad.

F.1.3.1 Cacheable Accesses

Accesses that fall within the coherence domain are called cacheable accesses. They are implemented in UltraSPARC T1 with the following properties:

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

F.1.3.2 Noncacheable and Side-Effect Accesses

Accesses that are outside the coherence domain are called noncacheable accesses. Accesses of some of these memory (or memory mapped) locations may result in side effects. Noncacheable accesses are implemented in UltraSPARC T1 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. 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.

F.1.3.3   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 F-1 illustrates the issues involved in mixing cacheable and noncacheable accesses.

CODE EXAMPLE F-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 T1, CODE EXAMPLE F-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.

F.1.4 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 T1 are described below.

- MEMBAR **#LoadLoad** — Forces all loads after the MEMBAR to wait until all loads before the MEMBAR have reached global visibility.
- MEMBAR **#StoreLoad** — Forces all loads after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility.
- MEMBAR **#LoadStore** — Forces all stores after the MEMBAR to wait until all loads before the MEMBAR have reached global visibility.
- MEMBAR **#StoreStore** and STBAR — Forces all stores after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility.

> **Notes** (1) STBAR has the same semantics as MEMBAR #StoreStore; it is included for SPARC V8 compatibility.
>
> (2) The above four MEMBARs do not guarantee ordering between cacheable accesses after noncacheable accesses.

- 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.

- MEMBAR **#MemIssue** — Forces all outstanding memory accesses to be *completed* before any memory access instruction after the MEMBAR is issued. It must be used to guarantee ordering of cacheable accesses following noncacheable accesses. 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.

  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.

- 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.

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.

F.1.4.1   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 T1, 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.

F.1.5   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 nonprivileged 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 noncacheable). An atomic access with an unsupported ASI causes a *data_access_exception* trap (with SFSR.ft = 8, illegal ASI value or virtual address). TABLE F-1 lists the ASIs that support atomic accesses.

TABLE F-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

TABLE F-1   ASIs that Support SWAP, LDSTUB, and CAS *(Continued)*

ASI Name	Access
ASI_PRIMARY{_LITTLE}	Unrestricted
ASI_SECONDARY{_LITTLE}	Unrestricted
ASI_REAL{_LITTLE}	Unrestricted

> **Note** | Atomic accesses with nonfaulting ASIs are not allowed, because these ASIs have the load-only attribute.

F.1.5.1   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 noncacheable but physically cacheable, allocation is done to the L2 cache only.

F.1.5.2   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.

F.1.5.3   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.

F.1.6   Nonfaulting Load

A nonfaulting load behaves like a normal load, except as follows:

- 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}).

ECC Codes

Chapter Revision History

Date	Name	Comments
August 8, 2003	Bill Bryg	Moved text from RAS Review doc.
August 26, 2003	Bill Bryg	Added ECC tables.
Sept 10, 2003	Bill Bryg	Added Poison descriptions.
Sept 19, 2003	Bill Bryg	Fixed ECC syndrome tables for L2 data and FRF. Deleted syndrome table for L2 tag (not sw visible).
Oct 16, 2003	Bill Bryg	Fixed typo in specifying L2 poison syndrome.
Oct 23, 2003	Bill Bryg	Removed A.7.1.3, since RMW to a line with error will not cause poison.
Nov 19, 2003	Bill Bryg	Modified source of poison description, describing case where sub-word DMA write corrupts both halves of 8B double-word.
Dec 1, 2003	Bill Bryg	Added description of memory external nibble order, because the check nibbles are scrambled (externally) relative to their software visible ordering.
Aug 20, 2004	Bill Bryg	Fixed L2 tag checkbit table, to correspond with only 22 bits of tag.
Aug 23, 2004	Bill Bryg	Fixed L2 tag checkbit table, changing order from C[5:0] to C[4:0,5], then relabeling back to C[5:0]..
11 Feb 2005	M. L. Nohr	Converted to UltraSPARC Architecture format
4 Jan 2006	D. Weaver	Added Tables back in. Cleanup of code formatting in G.6.2 on page 329
13 Feb 2006	D.Weaver	Corrections, per Jim Laudon's review

G.1 ECC Summary

The following arrays are protected by ECC:

TABLE G-1   Table 6 Error Handling

Array	ECC	Size	Instances	Total
L2 Cache Data	32+7 SEC/DED	936 KB	4	3744 KB
L2 Writeback Buffer	32+7 SEC/DED	624 B	4	2496 B
L2 Fill Buffer	32+7 SEC/DED	624 B	4	2496 B

TABLE G-1   Table 6 Error Handling

Array	ECC	Size	Instances	Total
L2 DMA Input Buffer	32+7 SEC/DED	312 B	4	1248 B
L2 Cache Tag	22+6 SEC	42 KB	4	168 KB
FP Register File	32+7 SEC/DED	1280 B	8	10 KB
Integer Register File	64+8 SEC/DED	9 KB	8	72 KB

The L2 Writeback Buffer, L2 Fill Buffer, and L2 DMA Buffer contain data that is in the processed of being moved to or from the cache. The ECC generation and check blocks were placed to include these buffers, but errors in these buffers are indistinguishable from L2 cache errors.

G.2   IRF ECC Code

TABLE G-2   IRF Check Bit Generation

Check [6:0]	Data[31:0] (bits 31 … 0)
	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
C0	0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 1
C1	1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1
C2	1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 0
C3	0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
C4	0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
C5	1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
C6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
C7	0 0 1 0 1 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1

Check [6:0]	Data[63:32] (bits 63 … 32)
	63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
C0	1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
C1	1 1 0 0 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1
C2	1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1

TABLE G-2　IRF Check Bit Generation

Check [6:0]	Data[31:0]																															
	3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0
C3	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
C4	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
C5	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C6	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C7	1	0	0	1	0	1	1	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	1

1 — Data bit is **xor**ed into calculation of that check bit.

0 — Data bit is not part of the check bit calculation.

The syndrome calculation is the inverse of the check bit calculation, for synd{6:0}. synd{7}, however, is simply the **xor** of data{63:0} and check{7:0}.

TABLE G-3　Syndrome Table for IRF ECC Code

SYND [7:4] Value	SYND [3:0] Value															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ne	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
1	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
2	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
3	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
4	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
5	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
6	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
7	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
8	C7	C0	C1	0	C2	1	2	3	C3	4	5	6	7	8	9	10
9	C4	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	C5	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
B	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56
C	C6	57	58	59	60	61	62	63	M	M	M	M	M	M	M	M
D	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
F	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

ne — No error

C0–C7 — Single-bit error on syndrome/check bit of that number.

0–63 — Single-bit error on data bit of that number.

U — Uncorrectable double (or 2N) bit error.

M — Triple or worse (2N + 1) bit error.

G.3 FRF ECC Code

The floating-point register files use the same 32+7 SEC/DED codes as the L2 Data, except that the FRF is never intentionally marked with a poison indication.

G.4 L2 Data ECC Code

TABLE G-4   L2 Data Check Bit Generation

Check [6:0]	Data[31:0]																															
	3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0
C0	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	1	0	1	1
C1	1	0	0	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1
C2	1	1	1	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	1	1	1	1	0	0	0	1	1	1	0
C3	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0
C4	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
C5	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C6	0	0	1	0	1	1	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0	1	1	0	1	1	1

1 — Data bit is **xor**ed into calculation of that check bit.

0 — Data bit is not part of the check bit calculation.

The syndrome calculation is the inverse of the check bit calculation, for synd{5:0}. synd{6}, however, is simply the **xor** of data{31:0} and check{6:0}.

TABLE G-5   Syndrome Table for L2 Data ECC Code

SYND [6:4] Value	SYND [3:0] Value															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ne	U	U	P/U	U	U	U	U	U	U	U	U	U	U	U	U
1	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
2	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
3	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
4	C6	C0	C1	0	C2	1	2	3	C3	4	5	6	7	8	9	10
5	C4	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
6	C5	26	27	28	29	30	31	M	M	M	M	M	M	M	M	M
7	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

ne — No error

C0–C6 — Single-bit error on syndrome/check bit of that number.

0–31 — Single-bit error on data bit of that number.

P/U — Poison, or uncorrectable double (or 2N) bit error

U — Uncorrectable double (or 2N) bit error.

M — Triple or worse (2N + 1) bit error

G.5   L2 Tag ECC Code

The syndrome is not captured on L2 Tag ECC errors (LTC), so we only document check bit calculation for L2 tag.

TABLE G-6   L2 Tag Check Bit Generation

Check [6:0]	Tag[21:0]																					
	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C0	1	0	0	1	1	0	0	1	0	1	1	1	1	0	0	1	0	1	1	0	1	1
C1	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	1	0	1	1
C2	1	1	0	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1

TABLE G-6   L2 Tag Check Bit Generation

Check [6:0]	Tag[21:0]

Check [6:0]	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0
C3	0	0	0	0	1	1	1	1	0	0	0	1	1	1	1	0	0	0	1	1	1	0
C4	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
C5	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

1 — Data bit is **xor**ed into calculation of that check bit.

0 — Data bit is not part of the check bit calculation.

G.6   Memory Extended ECC Support

UltraSPARC T1 supports Extended ECC error correction (QEC/OED) for main memory, where we can correct any error contained within a single memory nibble (4b) and detect as uncorrectable any error that is contained within any two nibbles. The ECC coding scheme uses 4-bit word (16 symbol), 128-bit data (16 words), and 16-bit syndrome (4 words) and uses Galois Field of (2^4) to implement Add/ Multiply Operation that is completely inclusive within its field (Definition of Galois Field). We use 3×4 bit correction code + 1×4 bit code (16 bits total) to correct 4-bit errors and detect 8-bit errors. While the addition is a trivial bitwise **xor**, the multiplication is not as straightforward and involves a Modulo multiplication, using its field Primitive Polynomial of value 10011. Also, the syndrome or Parity generated is **xor**ed bitwise with the parity of the address (^(PA{39:8}) to that location.

G.6.1   Nomenclature and Nibble Order

The data nibbles and check/syndrome nibbles are numbered in a little-endian fashion, so the order as seen by software is—

   C3 C2 C1 C0 N31 N30 ... N5 N4 N3 N2 N1 N0

—so N0 is made up of data{3:0}, N1 is data{7:4}, etc.

The data for the ECC code is referred to either as check nibbles or as syndrome (nibbles), depending on where it is relative to the ECC generation calculation and to the ECC check calculation. After the ECC generation calculation, it is called "check nibbles", and this is what is stored in physical DRAM. When memory is accessed, the data nibbles and check nibbles are read out of DRAM and run through an ECC check calculation, which produces the "syndrome nibbles", synd{15:0}.

The check nibbles are not directly accessible by software.

G.6.1.1   External Hardware Bit Order

External to the chip, the data nibbles are numbered in a little-endian fashion, but the check nibbles are numbered big-endian, so the order as seen by an external logic analyzer is:

C0   C1   C2   C3 N31 N30 ... N5 N4 N3 N2 N1 N0

← DRAMn_CB{15:0} → ←   DRAMn_DQ{127:0}   →

so N0 is made up of DQ{3:0}, N1 is DQ{7:4}, etc. However, the check nibbles are wired in big-endian order by nibble, but little-endian within each check nibble, so C0{3:0} is made up of CB{15:12}, C1{3:0} is CB{11:8}, C2{3:0} is CB{7:4}, and C3{3:0} is CB{3:0}.

G.6.2   Memory ECC Code Description

The calculation for the check nibbles is as follows:

```
Check Nibble0 (4 bits) = (N0 + 2*N1 + 3*N2 +  4*N3 + 5*N4 + 6*N5 +
      7*N6 + 8*N7 +
      9*N8 + A*N9 + B*N10 + C*N11 + D*N12 + E*N13 + F*N14 +
      N15 + 2*N16 + 3*N17 + 4*N18 + 5*N19 + 6*N20 + 7*N21 + 8*N22 +
      9*N23 + A*N24 + B*N25 + C*N26 + D*N27 + E*N28 + F*N29 + N31)
         ^ {4{addr_parity}}

Check Nibble1 (4 bits) = (N0 + N1 + N2 + N3 + N4 + N5 + N6 + N7 +
      N8 + N9 + N10 +
      N11 + N12 + N13 + N14 + N30 + N31) ^ {4{addr_parity}}

Check Nibble2 (4 bits) = (N15 + N16 + N17 + N18 + N19 + N20 + N21 +
      N22 + N23 + N24 +
      N25 + N26 + N27 + N28 + N29 + N30 + N31) ^ {4{addr_parity}}

Check Nibble3 (4 bits) = (N0 + 9*N1+ E*N2 + D*N3 + B*N4 + 7*N5 +
      6*N6 + F*N7 + 2*N8 +
      C*N9 + 5*N10+ A*N11 + 4*N12 + 3*N13 + 8*N14 + N15 + 9*N16 +
      E*N17 + D*N18 + B*N19 + 7*N20 + 6*N21 + F*N22 + 2*N23 + C*N24 +
      5*N25 + A*N26+ 4*N27 + 3*N28 + 8*N29 + N30) ^ {4{addr_parity}}
```

The calculation for the syndrome nibbles is similar, but includes the corresponding check nibble:

```
Syndrome Nibble0 (4 bits) = (C0 + N0 + 2*N1 + 3*N2 +  4*N3 + 5*N4 +
      6*N5 + 7*N6 + 8*N7 +
      9*N8 + A*N9 + B*N10 + C*N11 + D*N12 + E*N13 + F*N14 +
      N15 + 2*N16 + 3*N17 + 4*N18 + 5*N19 + 6*N20 + 7*N21 + 8*N22 +
      9*N23 + A*N24 + B*N25 + C*N26 + D*N27 + E*N28 + F*N29 + N31)
         ^ {4{addr_parity}}

Syndrome  Nibble1 (4 bits) = (C1 + N0 + N1 + N2 + N3 + N4 + N5 +
      N6 + N7 + N8 + N9 + N10 +
      N11 + N12 + N13 + N14 + N30 + N31) ^ {4{addr_parity}}

Syndrome  Nibble2 (4 bits) = (C2 + N15 + N16 + N17 + N18 + N19 +
      N20 + N21 + N22 + N23 + N24 +
      N25 + N26 + N27 + N28 + N29 + N30 + N31) ^ {4{addr_parity}}

Syndrome  Nibble3 (4 bits) = (C3 + N0 + 9*N1+ E*N2 + D*N3 + B*N4 +
      7*N5 + 6*N6 + F*N7 + 2*N8 +
      C*N9 + 5*N10+ A*N11 + 4*N12 + 3*N13 + 8*N14 + N15 + 9*N16 +
      E*N17 + D*N18 + B*N19 + 7*N20 + 6*N21 + F*N22 + 2*N23 + C*N24 +
      5*N25 + A*N26+ 4*N27 + 3*N28 + 8*N29 + N30) ^ {4{addr_parity}}
```

Error correction is accomplished by following equations. If S0, S1, S2 and S3 are the
4 Syndrome nibbles,

Position 0->14 (nibble position)
```
      if (S2 == 0 && S1 != 0 && S0 != 0), then {
            nibble_to_correct = (S0/S1 – 1);
            corrected_data = S1 + erred_nibble;
      }
```

Position 15->29 (nibble position)
```
      if (S1 == 0 && S0 != 0 && S2 != 0), then  {
            nibble_to_correct = (S0/S2 + 14);
            corrected_data = S2 + erred_nibble;
      }
```

Position 30 (nibble position)
```
      if (S0 == 0 && S1 != 0 && S2 != 0 && S1 == S2), then  {
            nibble_to_correct = N30
            corrected_data = S1+ erred_nibble;
      }
```

Position 31 (nibble position)

```
If ((S0 != 0) && (S1 != 0) && (S2 != 0) &&
    (S0 == S1 == S2)), then  {
        nibble_to_correct = N31;
    corrected_data = S1+ erred_nibble;
}
```

Notes | (1) Nibble S3 is not used in correction but only used for multiple error detection. Double errors are detected if (a) exactly two of the check-nibbles are non-zero, or (b) all four of the check-nibbles are non-zero, or (c) the nibble position as indicated by S0/S1or S0/S2 does not match the nibble position as indicated by S3/S1 or S3/S2, or (d) S1 and S2 are non-zero, and the non-zero check-nibbles are not all equal.

(2) This memory ECC scheme assumes that memory is implemented with x4 DRAMs. If x8 parts are used, this is effectively a SEC/DED scheme, with some multibit correction, but no Extended ECC survival capability.

G.6.3 Memory Address Parity Protection

Note that address parity is added (**xor**ed) into all of the check bits. Any normal address parity error will be detected as a multi-nibble uncorrectable error, since all four syndrome nibbles will be all-ones ($FFFF_{16}$) when the data is read back.

Address parity is defined as the **xor** of all the address bits that specify the bank-specific line address, which is ($^\wedge$(PA{39:8}) for a four-bank memory system, or ($^\wedge$(PA{39:7}) for a two-bank memory system.

G.6.4 Galois Multiplication Table

TABLE G-7   Galois Field Multiplication Table, Polynomial 10011

Multiplier	Multiplicand															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
2	0	2	4	6	8	a	c	e	3	1	7	5	b	9	f	d
3	0	3	6	5	c	f	a	9	b	8	d	e	7	4	1	2
4	0	4	8	c	3	7	b	f	6	2	e	a	5	1	d	9
5	0	5	a	f	7	2	d	8	e	b	4	1	9	c	3	6

TABLE G-7   Galois Field Multiplication Table, Polynomial 10011

Multiplier	Multiplicand															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
6	0	6	c	a	b	d	7	1	5	3	9	f	e	8	2	4
7	0	7	e	9	f	8	1	6	d	a	3	4	2	5	c	b
8	0	8	3	b	6	e	5	d	c	4	f	7	a	2	9	1
9	0	9	1	8	2	b	3	a	4	d	5	c	6	f	7	e
A	0	a	7	d	e	4	9	3	f	5	8	2	1	b	6	c
B	0	b	5	e	a	1	f	4	7	c	2	9	d	6	8	3
C	0	c	b	7	5	9	e	2	a	6	1	d	f	3	4	8
D	0	d	9	4	1	c	8	5	2	f	b	6	3	e	a	7
E	0	e	f	1	d	3	2	c	9	7	6	8	4	a	b	5
F	0	f	d	2	9	6	4	b	1	e	c	3	8	7	5	a

G.6.5   DRAM Syndrome Interpretation

When examining an UltraSPARC T1 DRAM syndrome, first look at TABLE G-8, to find that pattern of zeros and nonzero nibbles in the 16-bit syndrome. If the pattern of the syndrome nibbles is "a0bc" or "ab0c" (only the second or third nibble is zero), two more tables need to be checked to see which nibble is in error (TABLE G-9 or G), and (2) whether there is a multi-nibble error (TABLE G-10 or TABLE G-12).

The other syndrome nibble patterns are fully described in TABLE G-8.

TABLE G-8   Memory Syndrome Summary

S3	S2	S1	S0	Description
0	0	0	0	No error
a	0	b	c	Possible correctable error in N0-N14. See Table G-9 on page 334 and Table G-10 on page 335.
a	b	0	c	Possible correctable error in N15-N19. See Table G on page 323 and Table G-12 on page 337.
d	d	d	0	Correctable error in N30. Bits to correct are the value "d" in data nibble N30.
0	d	d	d	Correctable error in N31. Bits to correct are the value "d" in data nibble N31.

S3	S2	S1	S0	Description
0	0	0	e	Error in check nibble C0. Bits to correct are the value "e" in check nibble C0.
0	0	e	0	Error in check nibble C1. Bits to correct are the value "e" in check nibble C1.
0	e	0	0	Error in check nibble C2. Bits to correct are the value "e" in check nibble C2.
e	0	0	0	Error in check nibble C3. Bits to correct are the value "e" in check nibble C3.
0x1	0x1	0x1	0x8	Poison Indication, or possibly uncorrectable multiple nibble error
0xF	0xF	0xF	0xF	Address Parity Error, or possibly uncorrectable multiple nibble error
other				Uncorrectable multiple nibble error

a,b,c — Nonzero values for syndrome nibbles, potentially different values, or may be same

d — Nonzero identical values in these three syndrome nibbles.

e — Nonzero value in a single syndrome nibble.

0 — Syndrome value is zero as part of this pattern.

TABLE G-9 Memory Syndrome, Case a0bc, Contents = Nibble in error, Bits in nibble to correct

SYND [7:4]	SYND [3:0]															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1	*	N0 0x1	N1 0x1	N2 0x1	N3 0x1	N4 0x1	N5 0x1	N6 0x1	N7 0x1	N8 0x1	N9 0x1	N10 0x1	N11 0x1	N12 0x1	N13 0x1	N14 0x1
2	*	N8 0x2	N0 0x2	N7 0x2	N1 0x2	N10 0x2	N2 0x2	N9 0x2	N3 0x2	N12 0x2	N4 0x2	N11 0x2	N5 0x2	N14 0x2	N6 0x2	N13 0x2
3	*	N13 0x3	N14 0x3	N0 0x3	N12 0x3	N2 0x3	N1 0x3	N11 0x3	N8 0x3	N6 0x3	N5 0x3	N7 0x3	N3 0x3	N9 0x3	N10 0x3	N4 0x3
4	*	N12 0x4	N8 0x4	N3 0x4	N0 0x4	N11 0x4	N7 0x4	N4 0x4	N1 0x4	N14 0x4	N10 0x4	N5 0x4	N2 0x4	N13 0x4	N9 0x4	N6 0x4
5	*	N10 0x5	N4 0x5	N13 0x5	N9 0x5	N0 0x5	N14 0x5	N3 0x5	N6 0x5	N11 0x5	N1 0x5	N8 0x5	N12 0x5	N5 0x5	N7 0x5	N2 0x5
6	*	N6 0x6	N13 0x6	N8 0x6	N14 0x6	N7 0x6	N0 0x6	N5 0x6	N12 0x6	N9 0x6	N2 0x6	N3 0x6	N1 0x6	N4 0x6	N11 0x6	N10 0x6
7	*	N5 0x7	N11 0x7	N9 0x7	N10 0x7	N12 0x7	N6 0x7	N0 0x7	N4 0x7	N2 0x7	N8 0x7	N14 0x7	N13 0x7	N7 0x7	N1 0x7	N3 0x7
8	*	N14 0x8	N12 0x8	N1 0x8	N8 0x8	N5 0x8	N3 0x8	N10 0x8	N0 0x8	N13 0x8	N11 0x8	N2 0x8	N7 0x8	N6 0x8	N4 0x8	N9 0x8
9	*	N1 0x9	N3 0x9	N5 0x9	N7 0x9	N9 0x9	N11 0x9	N13 0x9	N2 0x9	N0 0x9	N6 0x9	N4 0x9	N10 0x9	N8 0x9	N14 0x9	N12 0x9
A	*	N11 0xa	N10 0xa	N6 0xa	N4 0xa	N8 0xa	N13 0xa	N1 0xa	N9 0xa	N5 0xa	N0 0xa	N12 0xa	N14 0xa	N2 0xa	N3 0xa	N7 0xa
B	*	N4 0xb	N9 0xb	N14 0xb	N6 0xb	N1 0xb	N12 0xb	N7 0xb	N13 0xb	N10 0xb	N3 0xb	N0 0xb	N8 0xb	N11 0xb	N2 0xb	N5 0xb
C	*	N9 0xc	N6 0xc	N12 0xc	N13 0xc	N3 0xc	N8 0xc	N2 0xc	N14 0xc	N4 0xc	N7 0xc	N1 0xc	N0 0xc	N10 0xc	N5 0xc	N11 0xc
D	*	N3 0xd	N7 0xd	N11 0xd	N2 0xd	N6 0xd	N10 0xd	N14 0xd	N5 0xd	N1 0xd	N13 0xd	N9 0xd	N4 0xd	N0 0xd	N12 0xd	N8 0xd
E	*	N2 0xe	N5 0xe	N4 0xe	N11 0xe	N14 0xe	N9 0xe	N8 0xe	N10 0xe	N7 0xe	N12 0xe	N13 0xe	N6 0xe	N3 0xe	N0 0xe	N1 0xe
F	*	N7 0xf	N2 0xf	N10 0xf	N5 0xf	N13 0xf	N4 0xf	N12 0xf	N11 0xf	N3 0xf	N14 0xf	N6 0xf	N9 0xf	N1 0xf	N8 0xf	N0 0xf

* — This table doesn't apply. Re-lookup on Table G-8 on page 332

N?? / 0x?? - Top identifies which nibble is in error. Bottom identifies error bits within nibble

TABLE G-10 Memory Syndrome, Case a0bc, Contents = SYND[15:12] value

SYND [7:4]	SYND [3:0]															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1	*	0x1	0x9	0xe	0xd	0xb	0x7	0x6	0xf	0x2	0xc	0x5	0xa	0x4	0x3	0x8
2	*	0x4	0x2	0xd	0x1	0xa	0xf	0xb	0x9	0x8	0x5	0x7	0xe	0x3	0xc	0x6
3	*	0x5	0xb	0x3	0xc	0x1	0x8	0xd	0x6	0xa	0x9	0x2	0x4	0x7	0xf	0xe
4	*	0x3	0x8	0x1	0x4	0xe	0x9	0xa	0x2	0x6	0x7	0xf	0xd	0xc	0x5	0xb
5	*	0x2	0x1	0xf	0x9	0x5	0xe	0xc	0xd	0x4	0xb	0xa	0x7	0x8	0x6	0x3
6	*	0x7	0xa	0xc	0x5	0x4	0x6	0x1	0xb	0xe	0x2	0x8	0x3	0xf	0x9	0xd
7	*	0x6	0x3	0x2	0x8	0xf	0x1	0x7	0x4	0xc	0xe	0xd	0x9	0xb	0xa	0x5
8	*	0xc	0x6	0x4	0x3	0xd	0x2	0xe	0x8	0xb	0xf	0x9	0x1	0x5	0x7	0xa
9	*	0xd	0xf	0xa	0xe	0x6	0x5	0x8	0x7	0x9	0x3	0xc	0xb	0x1	0x4	0x2
A	*	0x8	0x4	0x9	0x2	0x7	0xd	0x5	0x1	0x3	0xa	0xe	0xf	0x6	0xb	0xc
B	*	0x9	0xd	0x7	0xf	0xc	0xa	0x3	0xe	0x1	0x6	0xb	0x5	0x2	0x8	0x4
C	*	0xf	0xe	0x5	0x7	0x3	0xb	0x4	0xa	0xd	0x8	0x6	0xc	0x9	0x2	0x1
D	*	0xe	0x7	0xb	0xa	0x8	0xc	0x2	0x5	0xf	0x4	0x3	0x6	0xd	0x1	0x9
E	*	0xb	0xc	0x8	0x6	0x9	0x4	0xf	0x3	0x5	0xd	0x1	0x2	0xa	0xe	0x7
F	*	0xa	0x5	0x6	0xb	0x2	0x3	0x9	0xc	0x7	0x1	0x4	0x8	0xe	0xd	0xf

* — This table doesn't apply. Re-lookup on Table G-8 on page 332

0x?? - Specifies the value synd{15:12} must have; otherwise, this is a multi-nibble error.

TABLE G-11 Memory Syndrome, Case a0bc, Contents = Nibble in error, Bits in nibble to correct

SYND [7:4]	SYND [3:0]															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1	*	N0 0x1	N1 0x1	N2 0x1	N3 0x1	N4 0x1	N5 0x1	N6 0x1	N7 0x1	N8 0x1	N9 0x1	N10 0x1	N11 0x1	N12 0x1	N13 0x1	N14 0x1
2	*	N8 0x2	N0 0x2	N7 0x2	N1 0x2	N10 0x2	N2 0x2	N9 0x2	N3 0x2	N12 0x2	N4 0x2	N11 0x2	N5 0x2	N14 0x2	N6 0x2	N13 0x2
3	*	N13 0x3	N14 0x3	N0 0x3	N12 0x3	N2 0x3	N1 0x3	N11 0x3	N8 0x3	N6 0x3	N5 0x3	N7 0x3	N3 0x3	N9 0x3	N10 0x3	N4 0x3
4	*	N12 0x4	N8 0x4	N3 0x4	N0 0x4	N11 0x4	N7 0x4	N4 0x4	N1 0x4	N14 0x4	N10 0x4	N5 0x4	N2 0x4	N13 0x4	N9 0x4	N6 0x4
5	*	N10 0x5	N4 0x5	N13 0x5	N9 0x5	N0 0x5	N14 0x5	N3 0x5	N6 0x5	N11 0x5	N1 0x5	N8 0x5	N12 0x5	N5 0x5	N7 0x5	N2 0x5
6	*	N6 0x6	N13 0x6	N8 0x6	N14 0x6	N7 0x6	N0 0x6	N5 0x6	N12 0x6	N9 0x6	N2 0x6	N3 0x6	N1 0x6	N4 0x6	N11 0x6	N10 0x6
7	*	N5 0x7	N11 0x7	N9 0x7	N10 0x7	N12 0x7	N6 0x7	N0 0x7	N4 0x7	N2 0x7	N8 0x7	N14 0x7	N13 0x7	N7 0x7	N1 0x7	N3 0x7
8	*	N14 0x8	N12 0x8	N1 0x8	N8 0x8	N5 0x8	N3 0x8	N10 0x8	N0 0x8	N13 0x8	N11 0x8	N2 0x8	N7 0x8	N6 0x8	N4 0x8	N9 0x8
9	*	N1 0x9	N3 0x9	N5 0x9	N7 0x9	N9 0x9	N11 0x9	N13 0x9	N2 0x9	N0 0x9	N6 0x9	N4 0x9	N10 0x9	N8 0x9	N14 0x9	N12 0x9
A	*	N11 0xa	N10 0xa	N6 0xa	N4 0xa	N8 0xa	N13 0xa	N1 0xa	N9 0xa	N5 0xa	N0 0xa	N12 0xa	N14 0xa	N2 0xa	N3 0xa	N7 0xa
B	*	N4 0xb	N9 0xb	N14 0xb	N6 0xb	N1 0xb	N12 0xb	N7 0xb	N13 0xb	N10 0xb	N3 0xb	N0 0xb	N8 0xb	N11 0xb	N2 0xb	N5 0xb
C	*	N9 0xc	N6 0xc	N12 0xc	N13 0xc	N3 0xc	N8 0xc	N2 0xc	N14 0xc	N4 0xc	N7 0xc	N1 0xc	N0 0xc	N10 0xc	N5 0xc	N11 0xc
D	*	N3 0xd	N7 0xd	N11 0xd	N2 0xd	N6 0xd	N10 0xd	N14 0xd	N5 0xd	N1 0xd	N13 0xd	N9 0xd	N4 0xd	N0 0xd	N12 0xd	N8 0xd
E	*	N2 0xe	N5 0xe	N4 0xe	N11 0xe	N14 0xe	N9 0xe	N8 0xe	N10 0xe	N7 0xe	N12 0xe	N13 0xe	N6 0xe	N3 0xe	N0 0xe	N1 0xe
F	*	N7 0xf	N2 0xf	N10 0xf	N5 0xf	N13 0xf	N4 0xf	N12 0xf	N11 0xf	N3 0xf	N14 0xf	N6 0xf	N9 0xf	N1 0xf	N8 0xf	N0 0xf

* — This table doesn't apply. Re-lookup on Table G-8 on page 332

N?? / 0x?? - Top identifies which nibble is in error. Bottom identifies error bits within nibble

TABLE G-12 Memory Syndrome, Case ab0c, Contents = SYND[15:12] value

SYND [11:8]	SYND [3:0]															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
1	*	0x1	0x9	0xe	0xd	0xb	0x7	0x6	0xf	0x2	0xc	0x5	0xa	0x4	0x3	0x8
2	*	0x4	0x2	0xd	0x1	0xa	0xf	0xb	0x9	0x8	0x5	0x7	0xe	0x3	0xc	0x6
3	*	0x5	0xb	0x3	0xc	0x1	0x8	0xd	0x6	0xa	0x9	0x2	0x4	0x7	0xf	0xe
4	*	0x3	0x8	0x1	0x4	0xe	0x9	0xa	0x2	0x6	0x7	0xf	0xd	0xc	0x5	0xb
5	*	0x2	0x1	0xf	0x9	0x5	0xe	0xc	0xd	0x4	0xb	0xa	0x7	0x8	0x6	0x3
6	*	0x7	0xa	0xc	0x5	0x4	0x6	0x1	0xb	0xe	0x2	0x8	0x3	0xf	0x9	0xd
7	*	0x6	0x3	0x2	0x8	0xf	0x1	0x7	0x4	0xc	0xe	0xd	0x9	0xb	0xa	0x5
8	*	0xc	0x6	0x4	0x3	0xd	0x2	0xe	0x8	0xb	0xf	0x9	0x1	0x5	0x7	0xa
9	*	0xd	0xf	0xa	0xe	0x6	0x5	0x8	0x7	0x9	0x3	0xc	0xb	0x1	0x4	0x2
A	*	0x8	0x4	0x9	0x2	0x7	0xd	0x5	0x1	0x3	0xa	0xe	0xf	0x6	0xb	0xc
B	*	0x9	0xd	0x7	0xf	0xc	0xa	0x3	0xe	0x1	0x6	0xb	0x5	0x2	0x8	0x4
C	*	0xf	0xe	0x5	0x7	0x3	0xb	0x4	0xa	0xd	0x8	0x6	0xc	0x9	0x2	0x1
D	*	0xe	0x7	0xb	0xa	0x8	0xc	0x2	0x5	0xf	0x4	0x3	0x6	0xd	0x1	0x9
E	*	0xb	0xc	0x8	0x6	0x9	0x4	0xf	0x3	0x5	0xd	0x1	0x2	0xa	0xe	0x7
F	*	0xa	0x5	0x6	0xb	0x2	0x3	0x9	0xc	0x7	0x1	0x4	0x8	0xe	0xd	0xf

* — This table doesn't apply. Re-lookup on Table G-8 on page 332

0x?? - Specifies the value synd{15:12} must have; otherwise, this is a multi-nibble error.

G.7 Data Poisoning

Data poisoning is the practice of marking known corrupt data with bad ECC, so that any later access will get an ECC error. This is normally done when data is transferred from one cache or memory to another, to keep track that the data is corrupted.

G.7.1 Sources of Poison

G.7.1.1 JBUS Errors

If UltraSPARC T1 receives a DMA write with an error, the write will be completed but marked with a poison indication. An error on a DMA write can either be a JBUS Data Parity Error, or a Reported UE Error (which means the master of the transaction already had detected the error).

The destination of the DMA write will either be the L2 cache (if it is a subline write) or main memory (if it is a full line write), and hence that is what will be poisoned. JBUS errors occur with a 16-byte granularity, so poison will tend to be in 16-byte chunks on 16-byte boundaries.

The only exception to this (for JBUS-originating poison) is for subline writes that have finer than 16-byte granularity. If the subline write is made up of some number of 4-byte writes on 4-byte boundaries, only those 4-byte words into which corrupted data is written will be marked as poisoned. For any 8-byte doubleword that gets a corrupted subword (less and 4 bytes) write, both halves of the 8-byte doubleword will be marked as poisoned.

G.7.1.2 ECC Conversion of UEs

Another source of poison is uncorrectable ECC errors, when data is being transferred to a structure that has a different ECC (or parity) encoding. Thus, whenever data is transferred between memory, L2, and the L1 caches, and the source data has an uncorrectable error, the destination will be marked as poison, since the ECC code cannot be transferred intact.

G.7.2 Poisoning L1

The L1 caches (I and D) have only parity protection, so poisoning is implemented by marking the corrupt data (not tag) with a parity error (which is indistinguishable from other parity errors, except that there is also an L2 error with the same address). Since data is always transferred into the L1 in 16-byte chunks, poison in the L1 will always be in aligned 16-byte chunks.

Even though the L1 caches never have dirty data, it is still necessary to receive the corrupt data and install it into the cache, in order to maintain consistency between the L1 tags and the L2 directory.

If the L1 just dropped the corrupt data without installing into the cache, there are scenarios in which the L1 tags and L2 directory get out of sync and lose coherency, thus causing silent data corruption (because an invalidate didn't work).

G.7.3   Poisoning L2

L2 poisoning is implemented by flipping the two LSB check bits, $c\{1:0\}$, for each data word (4-byte) that is corrupt, which means that a syndrome of 03_{16} is probably a poison syndrome.

L2 ECC (and thus poison) has a 4-byte granularity, so every 4-byte word of uncorrectable data that is written into the cache will have a poison indication. However, all transfers out of the cache are done in 16-byte chunks, so DMA reads, L1 misses, and writebacks to memory will all increase poison/error indications to 16-byte granularity to the recipient, but the L2 is unmodified if it keeps the data.

G.7.3.1   Partial Write details

Subword writes are Read-Modify-Write, with any ECC correction after the read. If the word read has an uncorrectable error, however, the write is aborted, which leaves the original uncorrectable error intact. Thus, subline writes will not convert uncorrectable errors into poison.

G.7.4   Poisoning Memory

Memory poisoning is implemented by flipping four specific check bits, specifically $c\{15,9,5,0\}$ which means that a syndrome of 8221_{16} is most likely a poison indication. This syndrome was chosen because no single nibble error can convert a poison syndrome into a correctable error. A side effect of this is that a minimum of a triple nibble error is needed to "accidentally" generate a failing syndrome of 8221_{16}, so the possibility that a syndrome of 8221_{16} was generated by anything but poison is infinitesimal.

Memory ECC (and thus poison) has a 16-byte granularity, so every 16-byte chunk of uncorrectable data that is written to memory will have a poison indication.

G.7.5   Erasing Poison

Poison can be erased by overwriting it with good data, in such a way that no subword L2 writes occur, and no writebacks occur to main memory while the poison is partially erased.

Alternately, ASI_BLK_INIT_ST stores can be used to force poison to be overwritten, by causing the entire line to be zeroed out if the line was faulted back to memory. In this scheme, issues 4-byte or 8-byte stores to the poisoned area, followed by an ASI_BLK_INIT_ST to the first word of the line containing the poison.

Glossary

This chapter defines concepts and terminology common to all implementations of SPARC V9. It also includes terms that are unique to the UltraSPARC T1 implementation.

AFAR   Asynchronous Fault Address Register.

AFSR   Asynchronous Fault Status Register.

aliased   Said of each of two virtual addresses that referó to the same physical address.

ALU   Arithmetic Logical Unit

address space identifier (ASI)   An 8-bit value that identifies an address space. For each instruction or data access, the integer unit appends an ASI to the address. *See also* **implicit ASI**.

application program   A program executed with the processor in nonprivileged *mode*. **Note:** Statements made in this specification regarding application programs may not be applicable to programs (for example, debuggers) that have access to *privileged* processor state (for example, as stored in a memory-image dump).

architectural state   Software-visible registers and memory (including caches).

ARF   Architectural Register File.

ASI   Address Space Identifier.

ASI Ring   A daisy-chained bus connected in a loop fashion that goes through all of the blocks that have structures with diagnostic path or control registers for ASI access.

ASR   Ancillary State Register.

big-endian   An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte's significance decreases as its address increases.

BLD   (Obsolete) abbreviation for Block Load instruction; replaced by LDBLOCKF.

blocking ASI	An ASI that will access its ASI register/array location once all older instructions in that strand have retired, there are no instructions in the other strand which can issue, and the store queue, TSW, and LMB are all empty. Additionally, the snoop pipeline is stalled before accessing the ASI register/ array location. *See* **nonblocking ASI**.
branch outcome	Refers to whether or not a branch instruction will alter the flow of execution from the sequential path. A taken branch outcome results in execution proceeding with the instruction at the branch target; a not-taken branch outcome results in execution proceeding with the instruction along the sequential path after the branch.
branch resolution	A branch is said to be resolved when the result (that is, the branch outcome and branch target address) has been computed and is known for certain. Branch resolution can take place late in the pipeline.
branch target address	The address of the instruction to be executed if the branch is taken.
BST	(Obsolete) abbreviation for Block Store instruction; replaced by STBLOCKF.
bypass ASI	An ASI that refers to memory and for which the MMU does not perform virtual-to-real address translation (that is, memory is accessed using a direct real address).
byte	Eight consecutive bits of data.
CAM	Content Addressable Memory
CCR	Condition Codes register
clean window	A register window in which all of the registers contain 0, a valid address from the current address space, or valid data from the current address space.
CMP	Chip multiprocessor. A single chip processor that contains more than one virtual processor. *See also* **processor** and **virtual processor**.
coherence	A set of protocols guaranteeing that all memory accesses are globally visible to all caches in a shared-memory system.
commit	An instruction commits when it modifies architectural state.
completed	A memory transaction is said to be completed when an idealized memory has executed the transaction with respect to all processors. A load is considered completed when no subsequent memory transaction can affect the value returned by the load. A store is considered completed when no subsequent load can return the value that was overwritten by the store.
complex instruction	An instruction that requires the creation of secondary "helper" instructions for normal operation, excluding trap conditions such as spill/fill traps (which use helpers). Refer to *Instruction Latencies* on page 277 for a complete list of all complex instructions and their helper sequences.
consistency	*See* **coherence**.

context	A set of translations that supports a particular address space. *See also* **Memory Management Unit (MMU)**.
CWP	Current Window Pointer
CPI	Cycles per instruction. The number of clock cycles it takes to execute an instruction.
CPU	Central Processing Unit. A synonym for **virtual processor**.
cross-call	An interprocessor call in a multiprocessor system.
CSR	Control Status Register.
CTI	Control transfer instruction
current window	The block of 24 R registers that is currently in use. The Current Window Pointer (CWP) register points to the current window.
DCTI	Delayed control transfer instruction.
DDR	Double Data Rate.
demap	To invalidate a mapping in the MMU.
deprecated	The term applied to an architectural feature (such as an instruction or register) for which a SPARC V9 implementation provides support only for compatibility with previous versions of the architecture. Use of a deprecated feature must generate correct results but may compromise software performance. Deprecated features should not be used in new SPARC V9 software and may not be supported in future versions of the architecture.
DFT	Designed for test.
doublet	Two bytes (16 bits) of data.
doubleword	An aligned octlet. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.
DTLB	Data Cache Translation lookaside buffer.
ECC	Error Correction Code.
even parity	The mode of parity checking in which each combination of data bits plus a parity bit contains an even number of set bits.
exception	A condition that makes it impossible for the processor to continue executing the current instruction stream without software intervention. *See also* **trap**.
extended word	An aligned octlet, nominally containing integer data. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.
EXU	Execution Unit

F register	A floating-point register. SPARC V9 includes single-, double-, and quad-precision F registers.
fcc*n*	One of the floating-point condition code fields fcc0, fcc1, fcc2, or fcc3.
FGU	Floating-point and Graphics Unit.
floating-point exception	An exception that occurs during the execution of an FPop instruction as defined by the Fpop1, Fpop2, IMPDEP1, and IMPDEP2 opcodes. The exceptions are unfinished_FPop, unimplemented_FPop, sequence_error, hardware_error, invalid_fp_register, or IEEE_754_exception.
floating-point IEEE-754 exception	A floating-point exception, as specified by IEEE Std 754-1985. Listed within this specification as IEEE_754_exception.
floating-point operate (FPop) instructions	Instructions that perform floating-point and graphics calculations, as defined by the FPop1, FPop2,and IMPDEP1 opcodes. FPop instructions do not include FBfcc instructions, loads and stores between memory and the floating-point unit, or instructions defined by the IMPDEP2 opcodes.
floating-point trap type	The specific type of a floating-point exception, encoded in the FSR.ftt field.
floating-point unit	A processing unit that contains the floating-point registers and performs floating-point operations, as defined by this specification.
FP	Floating Point
FPRS	Floating Point Register State (register).
FRF	Floating-point register file.
FSR	Floating-Point Status register.
GL	Global-Level register.
GSR	General Status register.
halfword	An aligned doublet. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.
helper	An instruction generated by the IRU in response to a complex instruction. Helper instructions are not visible to software. Refer to *Instruction Latencies* on page 277 for a complete list of all complex instructions and their helper sequences.
hexlet	Sixteen bytes (128 bits) of data.

hyperprivileged	An adjective that describes (1) the state of the processor when it is executing in *hyperprivileged mode* (HPSTATE.hpriv = 1); (2) processor state that is only accessible to software while the processor is in *hyperprivileged mode*; for example, hyperprivileged registers, hyperprivileged ASRs, hyperprivileged ASIs, or, in general, hyperprivileged state; (3) an instruction that can be executed only when the processor is in *hyperprivileged mode*.
IFU	Instruction Fetch Unit.
implementation	Hardware or software that conforms to all of the specifications of an instruction set architecture (ISA).
implementation dependent	An aspect of the architecture that can 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 must not deviate from that range.
implicit ASI	The address space identifier that is supplied by the hardware on all instruction accesses and on data accesses that do not contain an explicit ASI or a reference to the contents of the ASI register.
informative appendix	An appendix containing information that is useful but not required to create an implementation that conforms to the SPARC V9 specification. *See also* **normative appendix**.
initiated	*Synonym*: **issued**.
instruction field	A bit field within an instruction word.
instruction set architecture	A set that defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. Does not define clock cycle times, cycles per instruction, data paths, etc. The bulk of the ISA implemented by UltraSPARC T1 is defined in the *UltraSPARC Architecture 2005*; a few extensions are described in this document.
integer unit (IU)	A processing unit that performs integer and control-flow operations and contains general-purpose integer registers and processor state registers, as defined by this specification.
interrupt request	A request for service presented to the processor by an external device.
IRF	Integer register file.
IRU	
ISA	Instruction set architecture.

issue	Used to describe the act of conveying an instruction from the instruction fetch unit for execution on the pipeline.
ITLB	Instruction Cache Translation lookaside buffer.
L2C	Level 2 Cache.
leaf procedure	A procedure that is a leaf in the program's call graph; that is, one that does not call (by using CALL or JMPL) any other procedures.
little-endian	An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the least significant; a byte's significance increases as its address increases.
load	An instruction that reads (but does not write) memory or reads (but does not write) location(s) in an alternate address space. *Load* includes loads into integer or floating-point registers, block loads, Load Quadword Atomic, and alternate address space variants of those instructions. *See also* **load-store** and **store**, the definitions of which are mutually exclusive with *load*.
load-store	An instruction that explicitly both reads and writes memory or explicitly reads and writes location(s) in an alternate address space. *Load-store* includes instructions such as CASA, CASXA, LDSTUB, LDSTUBA and the deprecated SWAP and SWAPA instructions. *See also* **load** and **store**, the definitions of which are mutually exclusive with *load-store*.
may	A keyword indicating flexibility of choice with no implied preference. **Note:** "May" indicates that an action or operation is allowed; "can" indicates that it is possible.
Memory Management Unit (MMU)	The address translation hardware that translates 64-bit virtual address into physical addresses. The MMU is composed of the TLBs, ASRs, and registers accessed through ASIs and is used to manage address translation. *See also* **context**, **physical address**, and **virtual address**.
must	*Synonym*: **shall**.
next program counter (NPC)	A register that contains the address of the instruction to be executed next if a trap does not occur.
NFO	Nonfault access only.
nonblocking ASI	An ASI that will access its ASI register/array location once all older instructions in that strand have retired and there are no instructions in the other strand which can issue. *See* **blocking ASI**.
nonfaulting load	A load operation that, in the absence of faults or in the presence of a recoverable fault, completes correctly, and in the presence of a nonrecoverable fault returns (with the assistance of system software) a known data value (nominally zero). *See* **speculative load**.

nonprivileged	An adjective that describes: (1) the state of the processor when PSTATE.priv = 0, that is, nonprivileged mode; (2) processor state information that is accessible to software while the processor is in either privileged mode or nonprivileged mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state; (3) an instruction that can be executed when the processor is in either privileged mode or nonprivileged mode.
nonprivileged mode	The mode in which a processor is operating when PSTATE.priv = 0. *See also* **privileged**.
normative appendix	An appendix containing specifications that must be met by an implementation conforming to the SPARC V9 specification. *See also* **informative appendix**.
nontranslating ASI	An ASI that does not refer to memory (for example, refers to control/status register(s)) and for which the MMU does not perform address translation.
NPC	Next program counter.
npt	Nonprivileged trap.
N_REG_WINDOWS	The number of register windows present in a particular implementation.
OBP	OpenBoot PROM.
octlet	Eight bytes (64 bits) of data. Not to be confused with "octet," which has been commonly used to describe eight bits of data. In this document, the term *byte*, rather than octet, is used to describe eight bits of data.
odd parity	The mode of parity checking in which each combination of data bits plus a parity bit contains an odd number of set bits.
older instruction	Refers to the relative fetch order of instructions. Instruction *i* is older than instruction *j* if instruction *i* was fetched before instruction *j*. Data dependencies flow from older instructions to younger instructions, and an instruction can only be dependent upon older instructions. *See* **younger instruction**.
one-hot	An *n*-bit binary signal is one-hot if, and only if, $n - 1$ of the bits are each 0 and a single bit is 1.
opcode	A bit pattern that identifies a particular instruction.
optional	A feature not required for compliance to an architecture specification (such as UltraSPARC Architecture 2005 or SPARC V9).
PA	*Physical address*.

Page Table Entry (PTE)	Describes the virtual-to-physical translation and page attributes for a specific page. A PTE generally means an entry in the page table or in the TLB, but it is sometimes used as an entry in the TSB (translation storage buffer). In general, a PTE contains fewer fields than does a TTE. *See also* **TLB** *and* **TSB**.
PC	Program counter.
PCR	Performance Control Register.
physical address	An address that maps real physical memory or I/O device space. *See also* **virtual address** *and* **real address**.
physical core	The term **physical processor core**, or just **physical core**, includes an execution pipeline and associated structures, such as caches, that are required for performing the execution of instructions from one or more software threads. A physical core contains one or more strands. The physical core provides the necessary resources for the threads on each strand to make forward progress at a reasonable rate. A multi-stranded physical core can execute multiple software threads either by time multiplexing or partitioning resources (or any combination thereof). In UltraSPARC T1, a physical core consists of four strands sharing the register files, execution pipeline, modular arithmetic unit, MMUs, and caches. *See also* **strand, thread,** *and* **virtual processor**.
PIC	Performance Instrumentation Counter.
PIL	Processor Interrupt Level.
PIO	Programmed I/O.
PIPT	Physically indexed, physically tagged.
POR	Power-on reset.
PPN	Physical Page Number
prefetchable	(1) An attribute of a memory location that indicates to an MMU that PREFETCH operations to that location may be applied. (2) A memory location condition for which the system designer has determined that no undesirable effects will occur if a PREFETCH operation to that location is allowed to succeed. Typically, normal memory is prefetchable. Nonprefetchable locations include those that, when read, change state or cause external events to occur. For example, some I/O devices are designed with registers that clear on read; others have registers that initiate operations when read. *See* **side effect**.
privileged	An adjective that describes (1) the state of the processor when it is executing in *privileged mode* (PSTATE.priv = 1 and HPSTATE.hpriv = 0); (2) processor state that is only accessible to software while the processor is in *privileged mode*; for example, privileged registers, privileged ASRs, privileged

ASIs, or in general, privileged state;
(3) an instruction that can be executed only when the processor is in *privileged mode*.

privileged mode The mode in which a processor is operating when PSTATE.priv = 1. *See also* **nonprivileged**.

processor The unit on which a shared interface is provided to control the configuration and execution of a collection of strands. A processor contains one or more physical cores, each of which contains one or more strands. On a more physical side, a processor is a physical module that plugs into a system. A processor is expected to appear logically as a single agent on the system interconnect fabric.

Therefore, a simple processor, like an UltraSPARC I processor, that can only execute one thread at a time would be a processor with a single physical core that is single-stranded. A processor that follows the academic model of simultaneous multithreading (SMT) would be a processor with a single physical core, where that physical core supports multiple strands in order to execute multiple threads at the same time (multi-stranded physical core). A processor that follows the academic model of a CMP would be a processor with multiple physical cores, each only supporting a single strand.

One can also have multiple physical cores where each physical core is multi-stranded. UltraSPARC T1 is an example of the latter, where each UltraSPARC T1 processor contains eight physical cores, each of which contains four strands.

program counter (PC) A register that contains the address of the instruction currently being executed by the IU.

PR Processor reset.

PSO Partial store order.

PTE Page Table Entry.

quadlet Four bytes (32 bits) of data.

quadword Aligned hexlet. **Note:** The definition of this term is architecture dependent and may be different from that used in other processor architectures.

RA Real address.

RAS Return Address Stack;
also Reliability, Availability and Serviceability.

RAW Read After Write

R register An integer register. Also called a general-purpose register or working register.

rd Rounding direction.

RDPR Read Privileged Register instruction

real address An address used by privileged mode code to describe the underlying physical memory. Real address are usually translated by a combination of hyperprivileged hardware and software to physical addresses which can be used to access real physical memory or I/O device space.

RED_state **R**eset, **E**rror, and **D**ebug state. The processor state when HPSTATE.red = 1. A restricted execution environment used to process resets and traps that occur when TL = *MAXTL* – 1.

reserved Describing an instruction field, certain bit combinations within an instruction field, or a register field that is reserved for definition by future versions of the architecture.

Reserved instruction fields shall read as 0, unless the implementation supports extended instructions within the field. The behavior of SPARC V9 processors when they encounter nonzero values in reserved instruction fields is undefined.

Reserved bit combinations within instruction fields are defined in Chapter 5, *Instruction Definitions*. In all cases, SPARC V9 processors shall decode and trap on these reserved combinations.

Reserved register fields should always be written by software with values of those fields previously read from that register or with zeroes; they should read as zero in hardware. Software intended to run on future versions of SPARC V9 should not assume that these fields will read as 0 or any other particular value. Throughout this specification, figures and tables illustrating registers and instruction encodings indicate reserved fields and combinations 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.

restricted Describing an address space identifier (ASI) that may be accessed only while the processor is operating in privileged mode.

rs1, rs2, rd The integer or floating-point register operands of an instruction. rs1 and rs2 are the source registers; rd is the destination register.

RMO Relaxed memory order.

RTO Read to own (cache line state).

RTS Read to share (cache line state).

SFAR Synchronous Fault Address register.

SFSR Synchronous Fault Status register.

shall A keyword indicating a mandatory requirement. Designers shall implement all such mandatory requirements to ensure interoperability with other SPARC V9-compliant products. *Synonym:* **must**.

should	A keyword indicating flexibility of choice with a strongly preferred implementation. *Synonym:* **it is recommended**.
SIAM	Set interval arithmetic mode instruction.
side effect	The result of a memory location having additional actions beyond the reading or writing of data. A side effect can occur when a memory operation on that location is allowed to succeed. Locations with side effects include those that, when accessed, change state or cause external events to occur. For example, some I/O devices contain registers that clear on read; others have registers that initiate operations when read. *See also* **prefetchable**.
SIMD	Single instruction stream, multiple data stream.
SIR	Software-initiated reset.
store	An instruction that writes (but does not explicitly read) memory or writes (but does not explicitly read) location(s) in an alternate address space. *Store* includes stores from either integer or floating-point registers, block stores, partial store, and alternate address space variants of those instructions. *See also* **load** and **load-store**, the definitions of which are mutually exclusive with *store*.
strand	A term for thread-specific hardware support that identifies the hardware state used to hold a software thread in order to execute it. Strand is specifically the software visible architected state (PC, NPC, general-purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) of a thread and any microarchitecture state required by hardware for its execution. "Strand" replaces the ambiguous term "hardware thread". The number of strands in a processor defines the number of threads that the operating system can schedule on that processor at any given time. *See also* **physical core**, **thread**, *and* **virtual processor**.
strand identifier (SID)	An n-bit value, in a processor implementing 2^n strands, that uniquely identifies each strand. The strand identifier in UltraSPARC T1 is five bits wide.
superscalar	An implementation that allows several instructions to be issued, executed, and committed in one clock cycle.
supervisor software	Software that executes when the processor is in privileged mode.
TBA	Trap base address.
thread	An executing process or lightweight process (LWP). Historically, the term thread is overused and ambiguous. Software and hardware have historically used it differently. From software's (operating system) perspective, the term thread refers to an entity that can be run on hardware, it is something that is scheduled and may or may not be actively running on hardware at any given time, and may migrate around the hardware of a system. From hardware's perspective, the term multithreaded processor refers to a processor that run multiple software threads simultaneously. To avoid confusion the term thread is used exclusively in the manner in which it is used by software and,

specifically, the operating system. A thread can be viewed in a practical sense as a Solaris™ process or lightweight process (LWP). *See also* **physical core**, **strand**, *and* **virtual processor**.

TICK   Hardware clock—TICK counter register.

TL   Trap Level

TLB   Translation lookaside buffer.

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.

TPC   Trap-saved PC.

Translation Lookaside Buffer (TLB)   A cache within an MMU that contains recent partial translations. TLBs speed up closely following translations by often eliminating the need to reread Page Table Entries from memory.

trap   The action taken by the processor when it changes the instruction flow in response to the presence of an exception, a Tcc instruction, or an interrupt. The action is a vectored transfer of control to privileged or hyperprivileged software through a table, the address of which is specified by the privileged Trap Base Address (TBA) register or the Hyperprivileged Trap Base Address (HTBA) register. *See also* **exception**.

TSB   Translation storage buffer. A table of the address translations that is maintained by software in system memory and that serves as a cache of the address translations.

TSO   Total store order.

TTE   Translation table entry. Describes the virtual-to-physical translation and page attributes for a specific page in the Page Table. In some cases, the term is explicitly used for the entries in the TSB.

unassigned   A value (for example, an ASI number), the semantics of which are not architecturally mandated and which may be determined independently by each implementation within any guidelines given.

undefined   An aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, or make any assumptions about, an undefined feature or behavior. Use of such a feature can deliver unexpected results, may or may not cause a trap, can vary among implementations, and can vary with time on a given implementation.

Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes (such as allowing user software to access privileged state), put the processor into supervisor mode, or put the processor into an unrecoverable state.

unimplemented	An architectural feature that is not directly executed in hardware because it is optional or is emulated in software.
unpredictable	*Synonym:* **undefined**.
unrestricted	Describing an address space identifier (ASI) that can be used regardless of the processor mode; that is, regardless of the values of PSTATE.priv and HPSTATE.hpriv.
user application program	*Synonym:* **application program**.
VA	*Virtual address*.
virtual address	An address produced by a processor that maps all systemwide, program-visible memory. Virtual addresses usually are translated by a combination of hardware and software to real addresses. Real address are usually translated by a combination of hardware and software to physical addresses which can be used to access physical memory and I/O device spaces.
virtual processor	The term **virtual processor**, is used to identify each strand in a processor. Each virtual processor corresponds to a specific strand on a specific physical core where there may be multiple physical cores each with multiple strands. In most respects, a virtual processor appears to the system, and to the operating system software, as a processing unit equivalent to a traditional single-stranded microprocessor (as in UltraSPARC I). Each virtual processor has its own interrupt ID and the operating system can schedule independent threads on each virtual processor. How multiple virtual processors are achieved within a processor is an implementation issue, and as much as possible the software interface is independent of how multiple virtual processors are implemented. The term virtual processor is used in place of strand because of the common association of the term strand with multi-stranded physical cores. *See also* **physical core**, **strand,** *and* **thread**.
VIS™	Visual instruction set.
VPA	Virtual Page Array
VPN	Virtual Page Number
WDR	Watchdog reset.
word	An aligned quadlet. Note: The definition of this term is architecture dependent and may differ from that used in other processor architectures.
younger instruction	*See* **older instruction**.
writeback	The process of writing a dirty cache line back to memory before it is refilled.
WRPR	Write Privileged Register.
XIR	Externally initiated reset.

Index