THE NETWORK IS THE COMPUTER™

UltraSPARC T2™ Supplement

to the *UltraSPARC Architecture 2007*

Draft D1.4.2, 01 Aug 2007

Privilege Levels: *Privileged and Nonprivileged*

Distribution: Public

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

Part No: 950-5556-01
Revision: Draft 1.4.2, 01 Aug 2007

Contents

CHAPTER 1

UltraSPARC T2 Basics

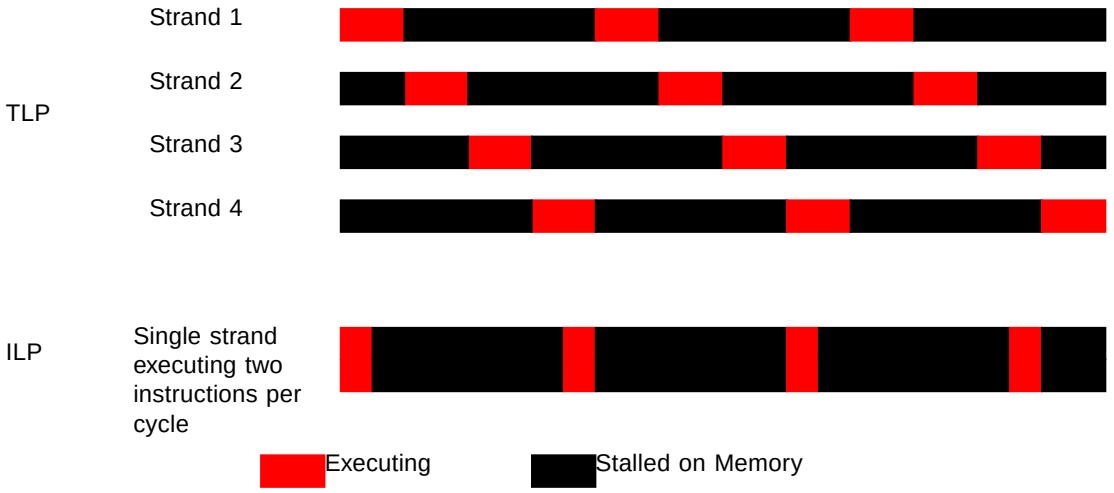
1.1 Background

UltraSPARC T2 is the follow-on chip multi-threaded (CMT) processor to the highly successful UltraSPARC T1 processor. The UltraSPARC T1 product line fully implements Sun's Throughput Computing initiative for the horizontal system space. 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 processors 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 processor 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 processors that employ multithreading are built in the same chip area. Combining multiple processors on a single chip with

multiple strands per processor, allows very high performance for highly threaded commercial applications. This approach is called thread-level parallelism (TLP), and the difference between TLP and ILP is shown in the FIGURE 1-1.

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 processor, and multiple processors 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 processors of today are the evolutionary outgrowth from a time when the processor was the bottleneck in delivering good performance. With processors 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 to employing TLP over ILP is that execution of a single thread will be slower on the TLP processor than an ILP processor. With processors running well over a GHz, a strand capable of executing only a single instruction per cycle is fully capable of completing tasks in the time required by the application, making this disadvantage a nonissue for nearly all commercial applications.

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 T2 Overview

UltraSPARC T2 is a single chip multi-threaded (CMT) processor. UltraSPARC T2 contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for eight strands, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The floating-point and memory pipelines are shared by all eight strands. The eight strands are hard-partitioned into two groups of four, and the four strands within a group share a single integer pipeline. While all eight strands run simultaneously, at any given time at most two strands will be active in the physical core, and those two strands will be issuing either a pair of integer pipeline operations, an integer operation and a floating-point operation, an integer operation and a memory operation, or a floating-point operation and a memory operation. Strands are switched on a cycle-by-cycle basis between the available strands within the hard-partitioned group of four using a least recently issued priority scheme. When a strand encounters a long-latency event, such as a cache miss, it is marked unavailable and instructions will not be issued from that strand until the long-latency event is resolved. Execution of the remaining available strands will continue while the long-latency event of the first strand is resolved.

Each SPARC physical core has a 16 KB, 8-way associative instruction cache (32-byte lines), 8 Kbytes, 4-way associative data cache (16-byte lines) that are shared by the eight strands. The eight SPARC physical cores are connected through a crossbar to an on-chip unified 4 Mbyte, 16-way associative L2 cache (64-byte lines). The L2 cache is banked eight ways to provide sufficient bandwidth for the eight SPARC physical cores.

1.3 UltraSPARC T2 Components

This section describes each component in UltraSPARC T2.

1.3.1 SPARC Physical Core

Each SPARC physical core has hardware support for eight strands. This support consists of a full register file (with eight register windows) per strand, with most of the ASI, ASR, and privileged registers replicated per strand. The eight strands share the instruction and data caches

A single floating-point unit is shared by all eight strands within a SPARC physical core. The shared floating-point unit is sufficient for most commercial applications which typically have less than 1% of their instructions being a floating-point operation.

1.3.2 L2 Cache

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

1.3.3 Memory Controller Unit (MCU)

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

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

1.3.4 Noncacheable Unit (NCU)

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

1.3.5 System Interface Unit (SIU)

The System Interface Unit connects the DMU and L2 Cache. SIU is the L2 Cache access point for the Network subsystem. The SIU-L2 Cache interface is also the ordering point for PCI-Express ordering rule.

1.3.6 SSI ROM Interface (SSI)

UltraSPARC T2 has a 50 Mb/s serial interface (SSI), which connects to an external field-programmable gate array (FPGA) that interfaces to the boot ROM. In addition, the SSI supports PIO accesses across the SSI, thus supporting optional Control and Status registers (CSRs) or other interfaces within the FPGA.

CHAPTER 2

Data Formats

Data formats supported by UltraSPARC T2 are described in the *UltraSPARC Architecture 2007* specification.

CHAPTER 3

Registers

3.1 Ancillary State Registers (ASRs)

This chapter discusses the UltraSPARC T2 ancillary state registers. TABLE 3-1 summarizes and defines these registers.

TABLE 3-1 Summary of UltraSPARC T2 Ancillary State Registers

Address	ASR Name	Access	priv	Description
00_{16}	Y	RW	N	Y Register
01_{16}	*Reserved*	—		Any access causes a *illegal_instruction* trap
02_{16}	CCR	RW	N	Condition Code register
03_{16}	ASI	RW	N	ASI register
04_{16}	TICK	RW	Y[1]	TICK register
05_{16}	PC	RO[2]	N	Program counter
06_{16}	FPRS	RW	N	Floating-Point Registers Status register
07_{16}–$0E_{16}$	*Reserved*	-		Any access causes an *illegal_instruction* trap
$0F_{16}$	(MEMBAR, STBAR)	—	N	Instruction opcodes only, not an actual ASR.
10_{16}	PCR	RW	Y[3]	Performance counter control register
11_{16}	PIC	RW	Y[4]	Performance instrumentation counter
12_{16}	*Reserved*	—		Any access causes an *illegal_instruction* trap
13_{16}	GSR	RW	N	General Status register
14_{16}	SOFTINT_SET	W	Y[5]	Set bit in Soft Interrupt register

TABLE 3-1 Summary of UltraSPARC T2 Ancillary State Registers *(Continued)*

Address	ASR Name	Access	priv	Description
15_{16}	SOFTINT_CLR	W	Y[5]	Clear bit in Soft Interrupt register
16_{16}	SOFTINT	RW	Y[3]	Soft Interrupt register
17_{16}	TICK_CMPR	RW	Y[3]	TICK Compare register
18_{16}	STICK	RW	Y[6]	System Tick register
19_{16}	STICK_CMPR	RW	Y[3]	System TICK Compare register
$1A_{16}$–$1F_{16}$	*Reserved*	—		Any access causes an *illegal_instruction* trap

Notes:

1. Nonprivileged software may read this register if the npt bit is 0. An attempt to read this register by nonprivileged software with npt = 1 causes a *privileged_action* trap. An attempted write by privileged software causes an *illegal_instruction* trap. An attempted write by nonprivileged software causes a *privileged_opcode* trap.
2. An attempted write to this register causes an *illegal_instruction* trap.
3. An attempted access in nonprivilged mode causes a *privileged_opcode* trap.
4. An attempted access in nonprivilged mode with PCR.priv = 1 causes a *privileged_action* trap.
5. Read accesses cause an *illegal_instruction* trap. An attempted write access in nonprivilged mode causes a *privileged_opcode* trap.
6. Nonprivileged software may read this register if the npt bit is 0. An attempt to read this register by nonprivileged software with npt = 1 causes a *privileged_action* trap. A write by privileged or user software causes an *illegal_instruction* trap.

3.1.1 Tick Register (TICK)

The TICK register contains two fields: npt and counter. The npt field is replicated per strand, while the counter field is shared by the eight strands on a physical 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 processor core clock.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.1.2 Program Counter (PC)

Each strand has a read-only program counter register. The PC contains a 48-bit virtual address and VA{63:48} is sign-extended from VA{47}. The format of this register is shown in TABLE 3-2.

TABLE 3-2 Program Counter – PC (ASR 05_{16})

Bit	Field	R/W	Description
63:48	va_high	RO	Sign-extended from VA{47}.
47:2	va	RO	Virtual address contained in the program counter.
1:0	—	RO	The lower 2 bits of the program counter always read as 0.

3.1.3 Floating-Point State Register (FSR)

Each virtual processor has a Floating-Point State register. This register follows the UltraSPARC Architecture 2007 specification, with the ver field permanently set to 0 and the qne field permanently set to 0 (UltraSPARC T2 does not support a FQ).

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.1.4 General Status Register (GSR)

Each virtual processor has a nonprivileged general status register (GSR). When PSTATE.pef or FPRS.fef is zero, accesses to this register cause an *fp_disabled* trap.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.1.5 Software Interrupt Register (SOFTINT)

Each virtual processor has a privileged software interrupt register. Nonprivileged accesses to this register cause a *privileged_opcode* trap. The TICK_CMPR register contains three fields: sm, int_level, and tm. Note that while setting the sm (bit 16), tm (bit 0), and SOFTINT{14} bits all generate *interrupt_level_14*, these bits are considered completely independent of each other. Thus a STICK compare will only set bit 16 and generate *interrupt_level_14*, not also set bit 14.

TABLE 3-3 specifies how *interrupt_level_14* will be shared between SOFTINT writes, STICK compares, and TICK compares.

TABLE 3-3 Sharing of *interrupt_level_14*

Event	tm	SOFTINT{14}	sm	Action
STICK compare when sm = 0	Unchanged	Unchanged	1	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 14
Set sm = 1 when sm = 0	Unchanged	Unchanged	1	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 4
Set SOFTINT{14} = 1 when SOFTINT{14} = 0.	Unchanged	1	Unchanged	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 4
TICK compare when tm = 0	1	Unchanged	Unchanged	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 4
Set tm=1 when tm = 0	1	Unchanged	Unchanged	*interrupt_level_14* if PSTATE.ie = 1 and PIL < 4

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.1.6 Tick Compare Register (TICK_CMPR)

Each virtual processor has a privileged Tick compare register. Nonprivileged accesses to this register cause a *privileged_opcode* trap. The TICK_CMPR register contains two fields: int_dis and tick_cmpr. A full 63-bit tick_cmpr field is implemented in the register, but the bottom seven bits are ignored when comparing against the TICK counter field. The int_dis bit controls whether a TICK *interrupt_level_14* interrupt is posted in the SOFTINT register when tick_cmpr bits 62:7 match TICK bits 62:7.

Caution To reliably create *interrupt_level_14* interrupts using the tick compare register, software should ensure that the value written to bits 62:7 of the Tick Compare Register is larger than the value subsequently read from bits 62:7 of the TICK Register.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.1.7 System Tick Register (STICK)

STICK and TICK are derived from the same register. Writes to STICK affect TICK and vice versa.

Writes by user-level code to TICK generate a *privileged_opcode* trap, while writes by user-level code to STICK generate an *illegal_instruction* trap.

Reads of STICK.counter{6:0} are tied to $7F_{16}$. This prevents software from setting the System Tick Compare Register or Hyperprivileged System Tick Compare Register to a value that should cause a subsequent interrupt but that would not be detected due to the System Tick Compare Register and Hyperprivileged System Tick Compare implementation. The compare registers are not continuously compared to STICK, but are compared periodically (at least once every 128 cycles).

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.1.8 System Tick Compare Register (STICK_CMPR)

Each virtual processor has a privileged System Tick Compare (STICK_CMPR register. Nonprivileged accesses to this register cause a *privileged_opcode* trap. STICK_CMPR contains two fields: int_dis and stick_cmpr. A full 63-bit stick_cmpr field is implemented in the register, but the bottom seven bits are ignored when comparing against the STICK counter field. To assist software in reliably creating interrupt_level_14 interrupts using the system tick compare register, UltraSPARC T2 always returns reads of the system tick register with bits 6:0 set to 7'h7F. This ensures that if software writes a value to the system tick compare register that is greater than the value subsequently read from the system tick register that a match will occur in the future.

The int_dis bit controls whether a STICK *interrupt_level_14* interrupt is posted in the SOFTINT register when stick_cmpr bits 62:7 match STICK bits 62:7.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.2 Privileged PR State Registers

TABLE 3-4 lists the privileged registers.

TABLE 3-4 Privileged Registers

Register	Register Name	Access	Description
00_{16}	TPC	RW	Trap PC[1]
01_{16}	TNPC	RW	Trap Next PC[1]
02_{16}	TSTATE	RW	Trap State
03_{16}	TT	RW	Trap Type
04_{16}	TICK	RW	Tick

TABLE 3-4 Privileged Registers

Register	Register Name	Access	Description
05_{16}	TBA	RW	Trap Base Address[1]
06_{16}	PSTATE	RW	Process State
07_{16}	TL	RW	Trap Level
08_{16}	PIL	RW	Processor Interrupt Level
09_{16}	CWP	RW	Current Window Pointer
$0A_{16}$	CANSAVE	RW	Savable Windows
$0B_{16}$	CANRESTORE	RW	Restorable Windows
$0C_{16}$	CLEANWIN	RW	Clean Windows
$0D_{16}$	OTHERWIN	RW	Other Windows
$0E_{16}$	WSTATE	RW	Window State
10_{16}	GL	RW	Global Level

1. UltraSPARC T2 only implements bits 47:0 of the TPC, TNPC, and TBA registers. Bits 63:48 are always sign-extended from bit 47.

3.2.1 Trap State Register (TSTATE)

Each virtual processor has *MAXPTL* (2) Trap State registers. These registers hold the state values from the previous trap level. The format of one element the TSTATE register array (corresponding to one trap level) is shown in TABLE 3-5.

TABLE 3-5 Trap State Register

Bit	Field	R/W	Description
63:42	—	RO	*Reserved*.
41:40	gl	RW	Global level at previous trap level
39:32	ccr	RW	CCR at previous trap level
31:24	asi	RW	ASI at previous trap level
23:21	—	RO	*Reserved*
20	pstate tct	RW	PSTATE.tct at previous trap level
19:18	—	RO	*Reserved* (corresponds to bits 11:10 of PSTATE)
17	pstate cle	RW	PSTATE.cle at previous trap level
16	pstate tle	RW	PSTATE.tle at previous trap level
15:13	—	RO	*Reserved* (corresponds to bits 7:5 of PSTATE)
12	pstate pef	RW	PSTATE.pef at previous trap level
11	pstate am	RW	PSTATE.am at previous trap level
10	pstate priv	RW	PSTATE.priv at previous trap level

TABLE 3-5 Trap State Register *(Continued)*

Bit	Field	R/W	Description
9	pstate ie	RW	PSTATE.ie at previous trap level
8	—	RO	*Reserved* (corresponds to bit 0 of PSTATE)
7:3	—	RO	*Reserved*
2:0	cwp	RW	CWP from previous trap level

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.2.2 Processor State Register (PSTATE)

Each virtual processor has a Processor State register. More details on PSTATE can be found in the UltraSPARC Architecture 2007 specification. The format of this register is shown in TABLE 3-6; note that the memory model selection field (mm) mentioned in UltraSPARC Architecture 2007 is not implemented in UltraSPARC T2.

TABLE 3-6 Processor State Register

Bit	Field	R/W	Description
63:13	—	RO	*Reserved*
12	tct	RW	Trap on control transfer
11:10	—	RO	*Reserved*
9	cle	RW	Current little endian
8	tle	RW	Trap little endian
7:6	—	RO	*Reserved* (mm; not implemented in UltraSPARC T2)
5	—	RO	*Reserved*
4	pef	RW	Enable floating-point
3	am	RW	Address mask
2	priv	RW	Privileged mode
1	ie	RW	Interrupt enable
0	—	RO	*Reserved* (was ag)

Implementation Note | Traps to hyperprivileged space will set PSTATE.priv to 0. PSTATE.priv could be set to either a 0 or 1 for this case.

Programming Note | Hyperprivileged changes to translation in delay slots of delayed control transfer instructions should be avoided.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.2.3 Trap Level Register (TL)

Each virtual processor has a Trap Level register. Writes to this register saturate at *MAXPTL* (2). This saturation is based on bits 2:0 of the write data; bits 63:3 of the write data are ignored.

Note | Hyperprivileged software can set TL to greater than *MAXPTL* for user or supervisor code by writing to TSTATE followed by a DONE/RETRY, etc.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.2.4 Current Window Pointer (CWP) Register

Since *N_REG_WINDOWS* = 8 on UltraSPARC T2, the CWP register in each virtual processor is implemented as a 3-bit register.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

3.2.5 Global Level Register (GL)

Each virtual processor has a Global Level register, which controls which set of global register windows is in use. The maximum global level (*MAXPGL*) for UltraSPARC T2 is 2, so GL is implemented as a 2-bit register on UltraSPARC T2. On a trap, GL is set to **min**(GL + 1,*MAXPTL*).

Writes to the GL register saturate at *MAXPTL*. This saturation is based on bits 3:0 of the write data; bits 63:4 of the write data are ignored.

The format of the GL register is shown in TABLE 3-7.

TABLE 3-7 Global Level Register

Bit	Field	R/W	Description
63:2	—	RO	*Reserved*
1:0	gl	RW	Global level.

For more information on this register, see the UltraSPARC Architecture 2007 specification.

CHAPTER 4

Instruction Format

Instruction formats are described in the UltraSPARC Architecture 2006 specification.

CHAPTER 5

Instruction Definitions

5.1 Instruction Set Summary

The UltraSPARC T2 CPU implements the UltraSPARC Architecture 2007*UltraSPARC Architecture 2007* instruction set.

TABLE 5-1 lists the complete UltraSPARC T2 instruction set supported in hardware. All instructions are documented in the *UltraSPARC Architecture 2007* specification.

TABLE 5-1 Complete UltraSPARC T2 Hardware-Supported Instruction Set *(1 of 6)*

Opcode	Description
ADD (ADDcc)	Add (and modify condition codes)
ADDC (ADDCcc)	Add with carry (and modify condition codes)
ALIGNADDRESS	Calculate address for misaligned data access
ALIGNADDRESSL	Calculate address for misaligned data access (little-endian)
ALLCLEAN	Mark all windows as clean
AND (ANDcc)	And (and modify condition codes)
ANDN (ANDNcc)	And not (and modify condition codes)
ARRAY{8,16,32}	3-D address to blocked byte address conversion
Bicc	Branch on integer condition codes
BMASK	Writes the GSR.mask field
BPcc	Branch on integer condition codes with prediction
BPr	Branch on contents of integer register with prediction
BSHUFFLE	Permutes bytes as specified by the GSR.mask field
CALL[1]	Call and link
CASA	Compare and swap word in alternate space
CASXA	Compare and swap doubleword in alternate space
DONE	Return from trap
EDGE{8,16,32}{L}{N}	Edge boundary processing {little-endian} {non-condition-code altering}

TABLE 5-1 Complete UltraSPARC T2 Hardware-Supported Instruction Set *(2 of 6)*

Opcode	Description
FABS(s,d)	Floating-point absolute value
FADD(s,d)	Floating-point add
FALIGNDATA	Perform data alignment for misaligned data
FANDNOT1{s}	Negated *src1* **and** *src2* (single precision)
FANDNOT2{s}	*Src1* **and** negated *src2* (single precision)
FAND{s}	Logical **and** (single precision)
FBPfcc	Branch on floating-point condition codes with prediction
FBfcc	Branch on floating-point condition codes
FCMP(s,d)	Floating-point compare
FCMPE(s,d)	Floating-point compare (exception if unordered)
FCMPEQ{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* = *src2*
FCMPGT{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* > *src2*
FCMPLE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* ≤ *src2*
FCMPNE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if *src1* ≠ *src2*
FDIV(s,d)	Floating-point divide
FEXPAND	Four 8-bit to 16-bit expand
FiTO(s,d)	Convert integer to floating-point
FLUSH	Flush instruction memory
FLUSHW	Flush register windows
FMOV(s,d)	Floating-point move
FMOV(s,d)cc	Move floating-point register if condition is satisfied
FMOV(s,d)R	Move floating-point register if integer register contents satisfy condition
FMUL(s,d)	Floating-point multiply
FMUL8SUX16	Signed upper 8- x 16-bit partitioned product of corresponding components
FMUL8ULX16	Unsigned lower 8- x 16-bit partitioned product of corresponding components
FMUL8X16	8- x 16-bit partitioned product of corresponding components
FMUL8X16AL	Signed lower 8- x 16-bit lower α partitioned product of four components
FMUL8X16AU	Signed upper 8- x 16-bit lower α partitioned product of four components
FMULD8SUX16	Signed upper 8- x 16-bit multiply → 32-bit partitioned product of components
FMULD8ULX16	Unsigned lower 8- x 16-bit multiply → 32-bit partitioned product of components
FNAND{s}	Logical **nand** (single precision)
FNEG(s,d)	Floating-point negate
FNOR{s}	Logical **nor** (single precision)
FNOT1{s}	Negate (1's complement) *src1* (single precision)
FNOT2{s}	Negate (1's complement) *src2* (single precision)
FONE{s}	One fill (single precision)
FORNOT1{s}	Negated *src1* **or** *src2* (single precision)

TABLE 5-1 Complete UltraSPARC T2 Hardware-Supported Instruction Set *(3 of 6)*

Opcode	Description
FORNOT2{s}	*src1* **or** negated *src2* (single precision)
FOR{s}	Logical **or** (single precision)
FPACKFIX	Two 32-bit to 16-bit fixed pack
FPACK{16,32}	Four 16-bit/two 32-bit pixel pack
FPADD{16,32}{s}	Four 16-bit/two 32-bit partitioned add (single precision)
fPMERGE	Two 32-bit to 64-bit fixed merge
FPSUB{16,32}{s}	Four 16-bit/two 32-bit partitioned subtract (single precision)
FsMULd	Floating-point multiply single to double
FSQRT(s,d)	Floating-point square root
FSRC1{s}	Copy *src1* (single precision)
FSRC2{s}	Copy *src2* (single precision)
F(s,d)TO(s,d)	Convert between floating-point formats
F(s,d)TOi	Convert floating point to integer
F(s,d)TOx	Convert floating point to 64-bit integer
FSUB(s,d)	Floating-point subtract
FXNOR{s}	Logical **xnor** (single precision)
FXOR{s}	Logical **xor** (single precision)
FxTO(s,d)	Convert 64-bit integer to floating-point
FZERO{s}	Zero fill (single precision)
ILLTRAP	Illegal instruction
INVALW	Mark all windows as CANSAVE
JMPL	Jump and link
LDBLOCKF	64-byte block load
LDDF	Load double floating-point
LDDFA	Load double floating-point from alternate space
LDF	Load floating-point
LDFA	Load floating-point from alternate space
LDFSR	Load floating-point state register lower
LDSB	Load signed byte
LDSBA	Load signed byte from alternate space
LDSH	Load signed halfword
LDSHA	Load signed halfword from alternate space
LDSTUB	Load-store unsigned byte
LDSTUBA	Load-store unsigned byte in alternate space
LDSW	Load signed word
LDSWA	Load signed word from alternate space
LDTW	Load twin words

TABLE 5-1 Complete UltraSPARC T2 Hardware-Supported Instruction Set *(4 of 6)*

Opcode	Description
LDTWA	Load twin words from alternate space
LDUB	Load unsigned byte
LDUBA	Load unsigned byte from alternate space
LDUH	Load unsigned halfword
LDUHA	Load unsigned halfword from alternate space
LDUW	Load unsigned word
LDUWA	Load unsigned word from alternate space
LDX	Load extended
LDXA	Load extended from alternate space
LDXFSR	Load extended floating-point state register
MEMBAR	Memory barrier
MOVcc	Move integer register if condition is satisfied
MOVr	Move integer register on contents of integer register
MULScc	Multiply step (and modify condition codes)
MULX	Multiply 64-bit integers
NOP	No operation
NORMALW	Mark other windows as restorable
OR (ORcc)	Inclusive-or (and modify condition codes)
ORN (ORNcc)	Inclusive-or not (and modify condition codes)
OTHERW	Mark restorable windows as other
PDIST	Distance between 8 8-bit components
POPC	Population count
PREFETCH	Prefetch data
PREFETCHA	Prefetch data from alternate space
PST	Eight 8-bit/4 16-bit/2 32-bit partial stores
RDASI	Read ASI register
RDASR	Read ancillary state register
RDCCR	Read condition codes register
RDFPRS	Read floating-point registers state register
RDPC	Read program counter
RDPR	Read privileged register
RDTICK	Read TICK register
RDY	Read Y register
RESTORE	Restore caller's window
RESTORED	Window has been restored
RETRY	Return from trap and retry
RETURN	Return

TABLE 5-1 Complete UltraSPARC T2 Hardware-Supported Instruction Set *(5 of 6)*

Opcode	Description
SAVE	Save caller's window
SAVED	Window has been saved
SDIV (SDIVcc)	32-bit signed integer divide (and modify condition codes)
SDIVX	64-bit signed integer divide
SETHI	Set high 22 bits of low word of integer register
SIAM	Set interval arithmetic mode
SLL	Shift left logical
SLLX	Shift left logical, extended
SMUL (SMULcc)	Signed integer multiply (and modify condition codes)
SRA	Shift right arithmetic
SRAX	Shift right arithmetic, extended
SRL	Shift right logical
SRLX	Shift right logical, extended
STB	Store byte
STBA	Store byte into alternate space
STBAR	Store barrier
STBLOCKF	64-byte block store
STDF	Store double floating-point
STDFA	Store double floating-point into alternate space
STF	Store floating-point
STFA	Store floating-point into alternate space
STFSR	Store floating-point state register
STH	Store halfword
STHA	Store halfword into alternate space
STTW	Store twin words
STTWA	Store twin words into alternate space
STW	Store word
STWA	Store word into alternate space
STX	Store extended
STXA	Store extended into alternate space
STXFSR	Store extended floating-point state register
SUB (SUBcc)	Subtract (and modify condition codes)
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)
SWAP	Swap integer register with memory
SWAPA	Swap integer register with memory in alternate space
TADDcc (TADDccTV)	Tagged add and modify condition codes (trap on overflow)

TABLE 5-1 Complete UltraSPARC T2 Hardware-Supported Instruction Set *(6 of 6)*

Opcode	Description
TSUBcc (TSUBccTV)	Tagged subtract and modify condition codes (trap on overflow)
Tcc	Trap on integer condition codes (with 8-bit sw_trap_number, if bit 7 is set trap to hyperprivileged)
UDIV (UDIVcc)	Unsigned integer divide (and modify condition codes)
UDIVX	64-bit unsigned integer divide
UMUL (UMULcc)	Unsigned integer multiply (and modify condition codes)
WRASI	Write ASI register
WRASR	Write ancillary state register
WRCCR	Write condition codes register
WRFPRS	Write floating-point registers state register
WRPR	Write privileged register
WRY	Write Y register
XNOR (XNORcc)	Exclusive-nor (and modify condition codes)
XOR (XORcc)	Exclusive-or (and modify condition codes)

1. The PC format saved by the CALL instruction is the same as the format of the PC register specified in Section 3.1.2, *Program Counter (PC)*, on page 11.

TABLE 5-2 lists the SPARC V9 and sun4v instructions that are not directly implemented in hardware by UltraSPARC T2, and the exception that occurs when an attempt is made to execute it.

TABLE 5-2 UltraSPARC Architecture 2007 Instructions Not Directly Implemented by UltraSPARC T2 Hardware *(1 of 2)*

Opcode	Description	Exception
FABSq	Floating-point absolute value quad	*illegal_instruction*
FADDq	Floating-point add quad	*illegal_instruction*
FCMPq	Floating-point compare quad	*illegal_instruction*
FCMPEq	Floating-point compare quad (exception if unordered)	*illegal_instruction*
FDIVq	Floating-point divide quad	*illegal_instruction*
FdMULq	Floating-point multiply double to quad	*illegal_instruction*
FiTOq	Convert integer to quad floating-point	*illegal_instruction*
FMOVq	Floating-point move quad	*illegal_instruction*
FMOVqcc	Move quad floating-point register if condition is satisfied	*illegal_instruction*
FMOVqr	Move quad floating-point register if integer register contents satisfy condition	*illegal_instruction*
FMULq	Floating-point multiply quad	*illegal_instruction*
FNEGq	Floating-point negate quad	*illegal_instruction*

TABLE 5-2 UltraSPARC Architecture 2007 Instructions Not Directly Implemented by UltraSPARC T2 Hardware *(2 of 2)*

Opcode	Description	Exception
FSQRTq	Floating-point square root quad	*illegal_instruction*
F(s,d,q)TO(q)	Convert between floating-point formats to quad	*illegal_instruction*
FQTOI	Convert quad floating point to integer	*illegal_instruction*
FQTOX	Convert quad floating point to 64-bit integer	*illegal_instruction*
FSUBq	Floating-point subtract quad	*illegal_instruction*
FxTOq	Convert 64-bit integer to floating-point	*illegal_instruction*
IMPDEP1 (not listed in TABLE 5-1)	Implementation-dependent instruction	*illegal_instruction*
IMPDEP2 (not listed in TABLE 5-1)	Implementation-dependent instruction	*illegal_instruction*
LDQF	Load quad floating-point	*illegal_instruction*
LDQFA	Load quad floating-point into alternate space	*illegal_instruction*
STQF	Store quad floating-point	*illegal_instruction*
STQFA	Store quad floating-point into alternate space	*illegal_instruction*

5.2 UltraSPARC T2-Specific Instructions

5.3 Block Load and Store Instructions

See the LDBLOCKF and STBLOCKF instruction descriptions in the *UltraSPARC Architecture 2007* specification for the standard definitions of these instructions.

.

A block load to IO space generates a *DAE_nc_page* trap.

Block stores to IO space are permitted.

Block store commits in UltraSPARC T2 do NOT force the data to be written to memory as specified in the *UltraSPARC Architecture 2007* specification. Block store commits are implemented the same as block stores in UltraSPARC T2. As with all stores, block stores and block store commits will maintain coherency with all I-caches, but will not flush any modified instructions executing down a pipeline. Flushing those instructions requires the pipeline to execute a FLUSH instruction.

Notes If LDBLOCKF is used with an `ASI_BLK_COMMIT_{P,S}` and a destination register number rd is specified which is not a multiple of 8 (a misaligned `rd`), UltraSPARC T2 generates an *illegal_instruction* exception (impl. dep. #255-U3-Cs10).

If LDBLOCKF is used with an `ASI_BLK_COMMIT_{P,S}` and a memory address is specified with less than 64-byte alignment, UltraSPARC T2 generates a *mem_address_not_aligned* exception (impl. dep. #256-U3)

These instructions are used for transferring large blocks of data (more than 256 bytes); for example, `bcopy()` and `bfill()`. On UltraSPARC T2, a block load forces a miss in the primary cache and will not allocate a line in the primary cache, but does allocate in L2.

UltraSPARC T2 treats block loads as interlocked with respect to following instructions. That is, all floating-point registers are updated before any subsequent instruction issues.

STBLOCKF source data registers are interlocked against completion of previous instructions, including block load instructions.

LDBLOCKF does not follow memory model ordering with respect to stores. In particular, read-after-write hazards to overlapping addresses are not detected. The side-effect bit associated with the access is ignored (see *Translation Table Entry (TTE)* on page 73). 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 the LDBLOCKF overlaps a previous store and there is no intervening MEMBAR or data reference, the LDBLOCKF may return data from before or after the store.

Compatibility Note Prior UltraSPARC implementations may have provided the first two registers at the same time. If code depends upon this unsupported behavior it must be modified for UltraSPARC T2.

STBLOCKF does not follow memory model ordering with respect to loads, previous block stores, or subsequent stores. (UltraSPARC T2 orders block stores with respect to previous nonblock stores). In particular, read-after-write hazards to overlapping addresses are not detected. The side-effects bit associated with the access is ignored. If ordering with respect to later loads is important then there must be an intervening MEMBAR instruction. If the STBLOCKF overlaps a later load and there is no intervening MEMBAR #`StoreLoad` instruction, the contents of the block are undefined.

Compatibility Notes

Block load and store operations do not obey the ordering restrictions of the currently selected processor memory model (TSO, PSO, or RMO); block operations always execute under an RMO memory ordering model. In general, 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 generally conform to dependence order on the issuing virtual processor; that is, no read-after-write or write-after-read checking occurs between block loads and stores. Explicit MEMBARs are required to enforce dependence ordering between block operations that reference the same address. However, UltraSPARC T2 partially orders some block operations.

TABLE 5-3 describes the synchronization primitives required in UltraSPARC T2, if any, to guarantee TSO ordering between various sequences of memory reference operations. The first column contains the reference type of the first or earlier instruction; the second column contains the reference type of the second or the later instruction. UltraSPARC T2 orders loads and block loads against all subsequent instructions.

TABLE 5-3 UltraSPARC T2 Synchronization Requirements for Memory Reference Operations

First reference	Second reference	Synchronization Required
Load	Load	—
	Block load	—
	Store	—
	Block store	—
Block load	Load	—
	Block load	—
	Store	—
	Block store	—
Store	Load	—
	Block load	MEMBAR `#StoreLoad` or `#Sync`
	Store	—
	Block store	—

TABLE 5-3 UltraSPARC T2 Synchronization Requirements for Memory Reference Operations

First reference	Second reference	Synchronization Required
Block store	Load	MEMBAR #StoreLoad or #Sync
	Block load	MEMBAR #StoreLoad or #Sync
	Store	MEMBAR #Sync
	Block store	MEMBAR #Sync

Block Initializing Store ASIs

Instruction	imm_asi	ASI Value	Operation
ST[B,H,W,TW,X]A	ASI_ST_BLKINIT_AS_IF_USER_PRIMARY (ASI_STBI_AIUP)	22_{16}	64-byte block initialing store to primary address space, user privilege
	ASI_ST_BLKINIT_AS_IF_USER_SECONDARY (ASI_STBI_AIUS)	23_{16}	64-byte block initialing store to secondary address space, user privilege
	ASI_ST_BLKINIT_NUCLEUS (ASI_STBI_N)	27_{16}	64-byte block initialing store to nucleus address space
	ASI_ST_BLKINIT_AS_IF_USER_PRIMARY_LITTLE (ASI_STBI_AIUPL)	$2A_{16}$	64-byte block initialing store to primary address space, user privilege, little-endian
	ASI_ST_BLKINIT_AS_IF_USER_SECONDARY_LITTLE (ASI_STBI_AIUS_L)	$2B_{16}$	64-byte block initialing store to secondary address space, user privilege, little-endian
	ASI_ST_BLKINIT_NUCLEUS_LITTLE (ASI_STBI_NL)	$2F_{16}$	64-byte block initialing store to nucleus address space, little-endian
	ASI_ST_BLKINIT_PRIMARY (ASI_STBI_P)	$E2_{16}$	64-byte block initialing store to primary address space
	ASI_ST_BLKINIT_SECONDARY (ASI_STBI_S)	$E3_{16}$	64-byte block initialing store to secondary address space
	ASI_ST_BLKINIT_PRIMARY_LITTLE (ASI_STBI_PL)	EA_{16}	64-byte block initialing store to primary address space, little-endian
	ASI_ST_BLKINIT_SECONDARY_LITTLE (ASI_STBI_SL)	EB_{16}	64-byte block initialing store to secondary address space, little-endian

Description Block initializing store instructions are selected by using one of the block initializing store ASIs with integer store 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 (to alternate space) are allowed to use these ASIs

All stores to these ASIs operate under relaxed memory ordering (RMO), regardless of the PSTATE.mm setting, and software must follow a sequence of these stores with a MEMBAR #Sync to ensure ordering with respect to subsequent loads and stores. Stores to these ASIs where the least-significant 6 bits of the address are non-zero (that is, not the first word in the cache line) behave the same as a normal RMO store. A store to these ASIs where the least-significant 6 bits are zero will load a cache line in the L2 cache with either all zeros or the existing data, and then update that line with the new store data. This special store will make sure the line maintains coherency when it is loaded into the cache, but will not generally fetch the line from memory (initializing it with zeros instead). Stores using these ASIs to a noncacheable address will behave the same as a normal store.

Note | These instructions are used for transferring large blocks of data (more than 256 bytes); for example, `bcopy()` and `bfill()`. On UltraSPARC T2, a quad load forces a miss in the primary cache and will not allocate a line in the primary cache, but does allocate in L2.

Access to these ASIs by a floating-point store (STFA, STDFA) will result in a *DAE_invalid_ASI* trap (or *mem_address_not_aligned* trap if not properly aligned for the store size).

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, guaranteeing 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]
   }
   ldtxa [%l0] #ASI_TWINX_P, %l2
   add %l0, 16, %l0
   stxa %l2, [%l1] #ASI_ST_BLKINIT_PRIMARY
   add %l1, 8, %l1
   stxa %l3, [%l1] #ASI_ST_BLKINIT_PRIMARY
   add %l1, 8, %l1
}
for (i = 0; i < 4; i++) {
   ldtxa [%l0] #ASI_TWINX_P, %l2
   add %l0, 16, %l0
   stx %l2, [%l1]
   stx %l3,d [%l1+8]
   add %l1, 16, %l1
```

```
}
membar #Sync
```

Programming Notes

These ASIs are specific to UltraSPARC T2 to provide a high-performance bcopy alternative to block load and store (which fetch the lined stored to from memory to the L2 cache, requiring three memory operations for `bcopy()` and two memory operations for a `bfill()`). These ASIs are of Class "N" and are only allowed in dynamically linked, platform-specific, OS-enabled libraries.

These ASIs provide a higher-performance `bcopy()` or `bfill()` than LDBLOCKF and STBLOCKF, due to their ability to avoid the unnecessary fetch from memory of the data that is overwritten by the store.

CHAPTER 6

Traps

6.1 Trap Levels

Each virtual processor supports two trap levels (*MAXPTL* = 2).

6.2 Trap Behavior

TABLE 6-1 specifies the codes used in the tables below.

TABLE 6-1 Table Codes

Code	Meaning
H	Trap is taken in Hyperprivileged mode
P	Trap is taken via the Privileged trap table, in Privileged mode (PSTATE.priv = 1)
-x-	Not possible. Hardware cannot generate this trap in the indicated running mode. For example, all privileged instructions can be executed in privileged mode, therefore a *privileged_opcode* trap cannot occur in privileged mode.
—	This trap can only legitimately be generated by hyperprivileged software, not by the CPU hardware. So, for the purposes of sun4v, the trap vector has to be correct, but for a hardware CPU implementation these trap types are not generated by the hardware, therefore the resultant running mode is irrelevant.

TABLE 6-2 Trap Behavior *(1 of 2)*

TT #	Hardware Trap Name	Priority	From privilege level: Nonprivileged	Privileged	
0_{16}	*Reserved*	—	—	—	
6_{16}	*Reserved*	—	—	—	
8_{16}	*IAE_privilege_violation*	3.1	H	-x-	-
B_{16}	*IAE_unauth_access*	2.9[1]	H	H	
C_{16}	*IAE_nfo_page*	3.3	H	H	
F_{16}	*Reserved*	—	—	—	
10_{16}	*illegal_instruction*	6.1[2]	H	H	
11_{16}	*privileged_opcode*	7	P	-x-	
12_{16}	*unimplemented_LDTW*	—	—	—	
13_{16}	*unimplemented_STTW*	—	—	—	—
14_{16}	*DAE_invalid_asi*	12.1	H	H	
15_{16}	*DAE_privilege_violation*	12.4	H	H	
16_{16}	*DAE_nc_page*	12.5	H	H	
17_{16}	*DAE_nfo_page*	12.6	H	H	
18_{16}–$1F_{16}$	*Reserved*	—	—	—	
20_{16}	*fp_disabled*	8.1	P	P	
21_{16}	*fp_exception_ieee_754*	11.1	P	P	
22_{16}	*fp_exception_other*	11.1	P	P	
23_{16}	*tag_overflow*	14	P	P	
24_{16}–27_{16}	*clean_window*	10.1	P	P	
28_{16}	*division_by_zero*	15	P	P	
$2C_{16}$	*Reserved*	—	—	—	
$2F_{16}$	*Reserved*	—	—	—	
30_{16}	*DAE_so_page*	12.6	H	H	
34_{16}	*mem_address_not_aligned*	10.2	H	H	
35_{16}	*LDDF_mem_address_not_aligned*	10.1	H	H	
36_{16}	*STDF_mem_address_not_aligned*	10.1	H	H	
37_{16}	*privileged_action*	11.1	H	H	
38_{16}	*LDQF_mem_address_not_aligned*	—	—	—	
39_{16}	*STQF_mem_addess_not_aligned*	—	—	—	
$3A_{16}$	*Reserved*	—	—	—	
41_{16}–$4F_{16}$	*interrupt_level_n*	32 – *n*	P	P	
50_{16}–$5D_{16}$	*Reserved*	—	—	—	
62_{16}	*VA_watchpoint*	11.2	H	H	
70_{16}	*Reserved*	—	—	—	—
73_{16}	*Reserved*	—	—	—	—

TABLE 6-2 Trap Behavior *(2 of 2)*

TT #	Hardware Trap Name	Priority	From privilege level: Nonprivileged	From privilege level: Privileged
74_{16}	*control_transfer_instruction*	11.1	P	P
75_{16}	*instruction_VA_watchpoint*	2.5	H	H
77_{16}–$7B_{16}$	*Reserved*	—	—	—
$7C_{16}$	*cpu_mondo_trap*	16.6	P	P
$7D_{16}$	*dev_mondo_trap*	16.7	P	P
$7E_{16}$	*resumable_error*	33.3	P	P
$7F_{16}$	*nonresumable_error* (generated by software only)	—	—	—
80_{16}–$9C_{16}$[1]	*spill_n_normal* (*n* = 0–7)	9	P	P
A0–BC_{16}[1]	*spill_n_other* (*n* = 0–7)	9	P	P
$C0_{16}$–DC_{16}[1]	*fill_n_normal* (*n* = 0–7)	9	P	P
$E0_{16}$–FC_{16}[1]	*fill_n_other* (*n* = 0–7)	9	P	P
100_{16}–$17F_{16}$	*trap_instruction*	16.2	P	P
180_{16}–$1FF_{16}$	*htrap_instruction*	16.2	-x-	H

1. UltraSPARC T2 deviates from the 3.2 priority in UltraSPARC Architecture 2007 for *IAE_unauth_access*.
2. UltraSPARC T2 deviates from UltraSPARC Architecture 2007 and swaps the priority of *illegal_instruction* (6.2 in UltraSPARC Architecture 2007) and *instruction_breakpoint* (6.1 in UltraSPARC Architecture 2007)

6.3 Trap Masking

TABLE 6-3 specifies the codes used inTABLE 6-3.

TABLE 6-3 Codes

Code	Meaning
(nm)	Never Masked — when the condition occurs in this running mode, it is never masked out and the trap is always taken.
(ie)	When the outstanding disrupting trap condition occurs in this privilege mode, it may be conditioned (masked out) by PSTATE.ie = 0 (but remains pending).
PIL	Masked by PSTATE.ie and PIL
tct	Masked by PSTATE.tct

TABLE 6-3 Codes *(Continued)*

Code	Meaning
M	Always masked
—	This trap can only legitimately be generated by hyperprivileged software, not by the CPU hardware. So, for the purposes of sun4v, the trap vector has to be correct, but for a hardware CPU implementation these trap types are not generated by the hardware, therefore the resultant running mode is irrelevant.

For example, trap $7C_{16}$ ("cpu mondo") in TABLE 6-4 is masked by PSTATE.ie in nonprivileged and privileged mode.

TABLE 6-4 lists the trap mask behavior.

TABLE 6-4 Trap Mask Behavior *(1 of 2)*

			From privilege level:	
TT #	**Hardware Trap Name**	**Type**	**Nonprivileged**	**Privileged**
0_{16}	*Reserved*	Reset	(nm)	(nm)
6_{16}	*Reserved*	—	—	—
8_{16}	*IAE_privilege_violation*	Precise	(nm)	(nm)
B_{16}	*IAE_unauth_access*	Precise	(nm)	(nm)
C_{16}	*IAE_nfo_page*	Precise	(nm)	(nm)
F_{16}	*Reserved*	—	—	—
10_{16}	*illegal_instruction*	Precise	(nm)	(nm)
11_{16}	*privileged_opcode*	Precise	(nm)	—
12_{16}	*unimplemented_LDTW*	—	—	—
13_{16}	*unimplemented_STTW*	—	—	—
14_{16}	*DAE_invalid_asi*	Precise	(nm)	(nm)
15_{16}	*DAE_privilege_violation*	Precise	(nm)	(nm)
16_{16}	*DAE_nc_page*	Precise	(nm)	(nm)
17_{16}	*DAE_nfo_page*	Precise	(nm)	(nm)
18_{16}–$1F_{16}$	*Reserved*	—	—	—
20_{16}	*fp_disabled*	Precise	(nm)	(nm)
21_{16}	*fp_exception_ieee_754*	Precise	(nm)	(nm)
22_{16}	*fp_exception_other*	Precise	(nm)	(nm)
23_{16}	*tag_overflow*	Precise	(nm)	(nm)
24_{16}–27_{16}	*clean_window*	Precise	(nm)	(nm)
28_{16}	*division_by_zero*	Precise	(nm)	(nm)
$2C_{16}$	*Reserved*	—	—	—
$2F_{16}$	*Reserved*	—	—	—
30_{16}	*DAE_so_page*	Precise	(nm)	(nm)

TABLE 6-4 Trap Mask Behavior *(2 of 2)*

TT #	Hardware Trap Name	Type	From privilege level: Nonprivileged	From privilege level: Privileged
34_{16}	*mem_address_not_aligned*	Precise	(nm)	(nm)
35_{16}	*LDDF_mem_address_not_aligned*	Precise	(nm)	(nm)
36_{16}	*STDF_mem_address_not_aligned*	Precise	(nm)	(nm)
37_{16}	*privileged_action*	Precise	(nm)	—
38_{16}	*LDQF_mem_address_not_aligned*	—	—	—
39_{16}	*STQF_mem_addess_not_aligned*	—	—	—
$3A_{16}$	*Reserved*	—	—	—
41_{16}–$4F_{16}$	*interrupt_level_n*	Disrupting	PIL	PIL
50_{16}–$5D_{16}$	*Reserved*	—	—	—
62_{16}	*VA_watchpoint*	Precise	(nm)	(nm)
70_{16}	*Reserved*	—	—	—
73_{16}	*Reserved*	—	—	—
74_{16}	*control_transfer_instruction*	Precise	tct	tct
75_{16}	*instruction_VA_watchpoint*	Precise	(nm)	(nm)
77_{16}–$7B_{16}$	*Reserved*	—	—	—
$7C_{16}$	*cpu_mondo_trap*	Disrupting	(ie)	(ie)
$7D_{16}$	*dev_mondo_trap*	Disrupting	(ie)	(ie)
$7E_{16}$	*resumable_error*	Disrupting	(ie)	(ie)
$7F_{16}$	*nonresumable_error* (generated by software only)	—	—	—
80_{16}–$9C_{16}$[1]	*spill_n_normal* (n = 0–7)	Precise	(nm)	(nm)
A0–BC_{16}[1]	*spill_n_other* (n = 0–7)	Precise	(nm)	(nm)
$C0_{16}$–DC_{16}[1]	*fill_n_normal* (n =0–7)	Precise	(nm)	(nm)
$E0_{16}$–FC_{16}[1]	*fill_n_other* (n = 0–7)	Precise	(nm)	(nm)
100_{16}–$17F_{16}$	*trap_instruction*	Precise	(nm)	(nm)
180_{16}–$1FF_{16}$	*htrap_instruction*	Precise	—	(nm)

CHAPTER 7

Interrupt Handling

The chapter describes the hardware interrupt delivery mechanism for the UltraSPARC T2 chip.

Hyperprivileged code notifies privileged code about some types of interrupts software recoverable errors, and hardware-corrected errors (and precise error traps) through the *cpu_mondo*, *dev_mondo*, and *resumable_error* traps as described in *Interrupt Queue Registers* on page 37. Software interrupts are delivered to each virtual processor using the *interrupt_level_n* traps. Software interrupts are described in the UltraSPARC Architecture 2006 Specification.

The second type of interrupts are external "mondo" interrupts, such as those generated by PCI-Express. These interrupts follow the standard mondo interrupt ACK/NACK flow control. Only the first two 64-bit words of mondo data are supported by UltraSPARC T2.

7.1 CPU Interrupt Registers

7.1.1 Interrupt Queue Registers

Each virtual processor has eight ASI_QUEUE registers at ASI = 25_{16}, VA{63:0} = $3C0_{16}$-$3F8_{16}$ that are used for communicating interrupts to the operating system. These registers contain the head and tail pointers for four supervisor interrupt queues: *cpu_mondo*, *dev_mondo*, *resumable_error*, *nonresumable_error*. The tail registers are read-only by supervisor, and read/write by hypervisor. Writes to the tail registers by the supervisor generate a *DAE_invalid_ASI* trap. The head registers are read/write by both supervisor and hypervisor.

Whenever the CPU_MONDO_HEAD register does not equal the CPU_MONDO_TAIL register, a *cpu_mondo* trap is generated. Whenever the DEV_MONDO_HEAD register does not equal the DEV_MONDO_TAIL register, a *dev_mondo* trap is generated. Whenever the RESUMABLE_ERROR_HEAD register does not equal the RESUMABLE_ERROR_TAIL register, a *resumable_error* trap is generated. Unlike

the other queue register pairs, the *nonresumable_error* trap is *not* automatically generated whenever the NONRESUMABLE_ERROR_HEAD register does not equal the NONRESUMABLE_ERROR_TAIL register; instead, the hypervisor will need to generate the *nonresumable_error* trap.

TABLE 7-1 through TABLE 7-8 define the format of the eight ASI_QUEUE registers.

TABLE 7-1 CPU Mondo Head Pointer – ASI_QUEUE_CPU_MONDO_HEAD (ASI 25_{16}, VA $3C0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	head	X	RW	Head pointer for CPU mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-2 CPU Mondo Tail Pointer – ASI_QUEUE_CPU_MONDO_TAIL (ASI 25_{16}, VA $3C8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	tail	X	RW (hyperpriv) RO (priv)	Tail pointer for CPU mondo interrupt queue.
5:0	—	0	RO	*Reserved*

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

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	head	X	RW	Head pointer for device mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-4 Device Mondo Tail Pointer – ASI_QUEUE_DEV_MONDO_TAIL (ASI 25_{16}, VA $3D8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	tail	X	RW (hyperpriv) RO (priv)	Tail pointer for device mondo interrupt queue.
5:0	—	0	RO	*Reserved*

TABLE 7-5 Resumable Error Head Pointer – ASI_QUEUE_RESUMABLE_HEAD (ASI 25_{16}, VA $3E0_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	Reserved.
17:6	head	X	RW	Head pointer for resumable error queue.
5:0	—	0	RO	*Reserved*

TABLE 7-6 Resumable Error Tail Pointer – `ASI_QUEUE_RESUMABLE_TAIL` (ASI 25_{16}, VA $3E8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	tail	X	RW (hyperpriv) RO (priv)	Tail pointer for resumable error queue.
5:0	—	0	RO	*Reserved*

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

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	head	X	RW	Head pointer for nonresumable error queue.
5:0	—	0	RO	*Reserved*

TABLE 7-8 Nonresumable Error Tail Pointer – `ASI_QUEUE_NONRESUMABLE_TAIL` (ASI 25_{16}, VA $3F8_{16}$)

Bit	Field	Initial Value	R/W	Description
63:18	—	0	RO	*Reserved*
17:6	tail	X	RW (hyperpriv) RO (priv)	Tail pointer for nonresumable error queue.
5:0	—	0	RO	*Reserved*

CHAPTER 8

Memory Models

SPARC V9 defines the semantics of memory operations for three memory models. From strongest to weakest, they are Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Order (RMO). The differences in these models lie in the freedom an implementation is allowed in order to obtain higher performance during program execution. The purpose of the memory models is to specify any constraints placed on the ordering of memory operations in uniprocessor and shared-memory multiprocessor environments. UltraSPARC T2 supports only TSO, with the exception that certain ASI accesses (such as block loads and stores) may operate under RMO.

Although a program written for a weaker memory model potentially benefits from higher execution rates, it may require explicit memory synchronization instructions to function correctly if data is shared. MEMBAR is a SPARC V9 memory synchronization primitive that enables a programmer to control explicitly the ordering in a sequence of memory operations. Processor consistency is guaranteed in all memory models.

The current memory model is indicated in the PSTATE.mm field. It is unaffected by normal traps. UltraSPARC T2 ignores the value set in this field and always operates under TSO.

A memory location is identified by an 8-bit address space identifier (ASI) and a 64-bit virtual address. The 8-bit ASI may be obtained from a ASI register or included in a memory access instruction. The ASI is used to distinguish between and provide an attribute for different 64-bit address spaces. For example, the ASI is used by the UltraSPARC T2 MMU to control access to implementation-dependent control and data registers and for access protection. Attempts by nonprivileged software (PSTATE.priv = 0) to access restricted ASIs (ASI{7} = 0) cause a *privileged_action* trap.

Real memory spaces can be accessed without side effects. For example, a read from real memory space returns the information most recently written. In addition, an access to real memory space does not result in program-visible side effects.

8.1 Supported Memory Models

The following sections contain brief descriptions of the two memory models supported by UltraSPARC T2. These definitions are for general illustration. Detailed definitions of these models can be found in *The SPARC Architecture Manual-Version 9*. The definitions in the following sections apply to system behavior as seen by the programmer.

Notes Stores to UltraSPARC T2 internal ASIs, block loads, and block stores and block initializing stores are outside the memory model; that is, they need MEMBARs to control ordering.

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

8.1.1 TSO

UltraSPARC T2 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 to I/O space are all strongly ordered with respect to each other.
- 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.

8.1.2 RMO

UltraSPARC T2 implements the following programmer-visible properties for special ASI accesses that operate under Relaxed Memory Order (RMO) mode:

- There is no implicit order between any two memory references, either cacheable or noncacheable, except that noncacheable accesses to I/O space) are all strongly ordered with respect to each other.
- A MEMBAR must be used between cacheable memory references if stronger order is desired. A MEMBAR `#MemIssue` is needed for ordering of cacheable after noncacheable accesses. A MEMBAR `#Lookaside` should be used between a store and a subsequent load at the same noncacheable address.

CHAPTER 9

Address Spaces and ASIs

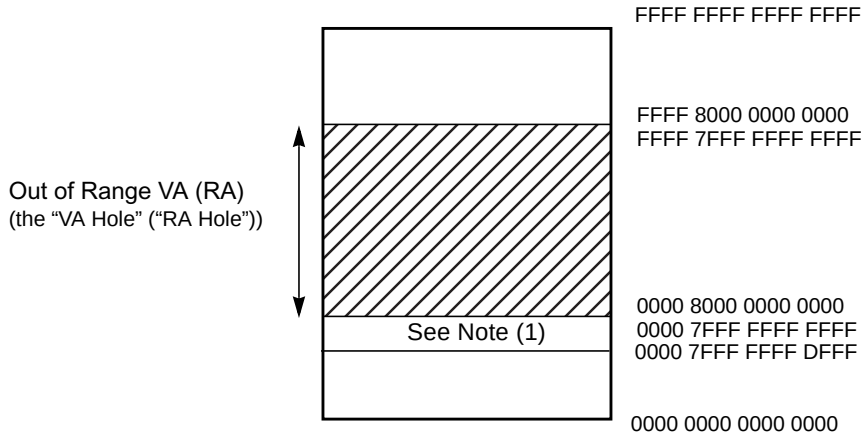
9.1 Address Spaces

UltraSPARC T2 supports a 48-bit virtual address space.

9.1.1 48-bit Virtual and Real Address Spaces

UltraSPARC T2 supports a 48-bit subset of the full 64-bit virtual and real address spaces. 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 (real) address space. Virtual (real) addresses between $0000\ 8000\ 0000\ 0000_{16}$ and $FFFF\ 7FFF\ FFFF\ FFFF_{16}$ inclusive lie within a "VA hole" ("RA hole"), are termed "out-of-range"[1], and are illegal. Prior UltraSPARC implementations introduced the additional restriction on software to not use pages within 4 Gbytes of the VA (RA) hole as instruction pages to avoid problems with prefetching into the VA (RA) hole. UltraSPARC T2 implements a hardware check for instruction fetching near the VA (RA) hole and generates a trap when instructions are executed from a location in the address range $0000\ 7FFF\ FFFF\ FFE0_{16}$ to $0000\ 7FFF\ FFFF\ FFFF_{16}$, inclusive. However, even though UltraSPARC T2 provides this hardware checking, it is still recommended that software should *not* use the 8-Kbyte page before the VA (RA) hole for instructions. Address translation and MMU related descriptions can be found in *Translation* on page 80.

1. Another way to view an out-of-range address is as any address where bits {63:48} are not all equal to bit {47}.



Note (1): Use of this region restricted to data only.

FIGURE 9-1 UltraSPARC T2's 48-bit Virtual and Real Address Spaces, With Hole

Throughout this document, when virtual (real) address fields are specified as 64-bit quantities, they are assumed to be sign-extended based on VA{47} (RA{47}).

A number of state registers are affected by the reduced virtual and real address spaces. The PC register is 48 bits, sign-extended to 64-bits on read accesses. TBA, TPC, and TNPC, registers are 48-bits and their values are *not* sign-extended when read. 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 virtual (real) address during an instruction access, caused by execution into the VA (RA) hole or into 0000 7FFF FFFF $FFE0_{16}$ to 0000 7FFF FFFF $FFFF_{16}$ inclusive, results in a trap if PSTATE.am = 0. In addition, UltraSPARC T2 hardware detects when a branch target is in the VA hole, and PSTATE.am changes from being set ('1') for the branch, DONE, or RETRY to being cleared ('0') for the target instruction (via the branch delay slot instruction or TSTATE) and generates an exception.

If the target virtual (real) address of a JMPL, RETURN, branch, or CALL instruction is an out-of-range address and PSTATE.am = 0, a trap is generated with TPC equal to the address of the JMPL, RETURN, branch, or CALL instruction.

An out-of-range virtual (real) address during a data access results in a trap if PSTATE.am = 0.

9.2 Alternate Address Spaces

TABLE 9-1 summarizes the ASI usage in UltraSPARC T2. The Section/Page column contains a reference to the detailed explanation of the ASI (the page number refers to this chapter). For internal ASIs, the legal VAs are listed (or the field contains "Any" if all VAs are legal). Only bits 47:0 are checked when determining the legal VA range. An access outside the legal VA range will generate a *DAE_invalid_asi* trap.

Notes | All internal, nontranslating ASIs in UltraSPARC T2 can only be accessed using LDXA and STXA.

ASIs 80_{16}–FF_{16} are unrestricted (access allowed in all modes -- nonprivileged, privileged). ASIs 00_{16}–$2F_{16}$ are restricted to privileged and hyperprivileged modes.

TABLE 9-1 UltraSPARC T2 ASI Usage *(1 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
00_{16}–03_{16}			Any	—	*DAE_invalid_asi*	
04_{16}	ASI_NUCLEUS	RW	Any	—	Implicit address space, nucleus context, TL > 0	(See UA-2007)
05_{16}–$0B_{16}$			Any	—	*DAE_invalid_asi*	
$0C_{16}$	ASI NUCLEUS_LITTLE	RW	Any	—	Implicit address space, nucleus context, TL > 0 (LE)	(See UA-2007)
$0D_{16}$–$0F_{16}$			Any	—	*DAE_invalid_asi*	
10_{16}	ASI_AS_IF_USER_PRIMARY	RW	Any	—	Primary address space, user privilege	(See UA-2007)
11_{16}	ASI_AS_IF_USER_SECONDARY	RW	Any	—	Secondary address space, user privilege	(See UA-2007)
12_{16}–13_{16}			Any	—		
14_{16}	ASI_REAL	RW	Any	—	Real address (normally used as cacheable)	page 53
15_{16}	ASI_REAL_IO	RW	Any	—	Real address (normally used as noncacheable, with side effect)	page 53
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY	RW	Any	—	64-byte block load/store, primary address space, user privilege	5.3

TABLE 9-1 UltraSPARC T2 ASI Usage *(2 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY	RW	Any	—	64-byte block load/store, secondary address space, user privilege	5.3
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE	RW	Any	—	Primary address space, user privilege (LE)	(See UA-2007)
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE	RW	Any	—	Secondary address space, user privilege (LE)	(See UA-2007)
$1A_{16}$–$1B_{16}$			Any	—		
$1C_{16}$	ASI_REAL_LITTLE	RW	Any	—	Real address (normally used as cacheable) (LE)	page 53
$1D_{16}$	ASI_REAL_IO_LITTLE	RW	Any	—	Real address (normally used as noncacheable, with side effect) (LE)	page 53
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE	RW	Any	—	64-byte block load/store, primary address space, user privilege (LE)	5.3
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE	RW	Any	—	64-byte block load/store, secondary address space, user privilege (LE)	5.3
20_{16}	ASI_SCRATCHPAD	RW	0_{16}–18_{16}	Y	Scratchpad registers	page 53
20_{16}	ASI_SCRATCHPAD	RW	20_{16}–28_{16}	—		page 53
20_{16}	ASI_SCRATCHPAD	RW	30_{16}–38_{16}	Y	Scratchpad registers	page 53
21_{16}	ASI_MMU	RW	8_{16}	Y	I/DMMU Primary Context register 0	12.7.2
21_{16}	ASI_MMU	RW	10_{16}	Y	DMMU Secondary Context register 0	12.7.2
21_{16}	ASI_MMU	RW	108_{16}	Y	I/DMMU Primary Context register 1	12.7.2
21_{16}	ASI_MMU	RW	110_{16}	Y	DMMU Secondary Context register 1	12.7.2
22_{16}	ASI_TWINX_AIUP, ASI_STBI_AIUP	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space, user privilege Store: Block initializing store, primary address space, user privilege	5.7.4

TABLE 9-1 UltraSPARC T2 ASI Usage *(3 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
23_{16}	`ASI_TWINX_AIUS, ASI_STBI_AIUS`	RW	Any	—	Load: 128-bit atomic load twin extended word, secondary address space, user privilege Store: Block initializing store	(See UA-2007)
24_{16}	`ASI_TWINX`	RO	Any	—	128-bit atomic load twin extended word	(See UA-2007)
25_{16}	`ASI_QUEUE`	RW	$3C0_{16}$	Y	CPU Mondo Queue head pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW (hyperpriv) RO (priv)	3C8	Y	CPU Mondo Queue tail pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW	$3D0_{16}$	Y	Device Mondo Queue head pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW (hyperpriv) RO (priv)	$3D8_{16}$	Y	Device Mondo Queue tail pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW	$3E0_{16}$	Y	Resumable Error Queue head pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW (hyperpriv) RO (priv)	$3E8_{16}$	Y	Resumable Error Queue tail pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW	$3F0_{16}$	Y	Nonresumable Error Queue head pointer	7.1.1
25_{16}	`ASI_QUEUE`	RW (hyper-priv) RO (priv)	$3F8_{16}$	Y	Nonresumable Error Queue tail pointer	7.1.1
26_{16}	`ASI_TWINX_REAL`	R	Any	—	128-bit atomic LDDA, real address	(See UA-2007)
27_{16}	`ASI_TWINX_NUCLEUS, ASI_STBI_N`	RW	Any	—	Load: 128-bit atomic load twin extended word from nucleus context Store: Block initializing store from nucleus context	(See UA-2007)
28_{16}–29_{16}			Any	—	*DAE_invalid_asi*	

TABLE 9-1 UltraSPARC T2 ASI Usage *(4 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
$2A_{16}$	ASI_TWINX_AIUPL, ASI_STBI_AIUPL	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space, user privilege, little endian Store: Block initializing store, primary address space, user privilege, little endian	(See UA-2007)
$2B_{16}$	ASI_TWINX_AIUSL, ASI_STBI_AIUSL	RW	Any	—	Load: 128-bit atomic load twin extended word, secondary address space, user privilege, little endian Store: Block initializing store, secondary address space, user privilege, little endian	(See UA-2007)
$2C_{16}$	ASI_TWINX_LITTLE	RO	Any	—	128-bit atomic load twin extended word, little endian	(See UA-2007)
$2D_{16}$			Any	—	*DAE_invalid_asi*	
$2E_{16}$	ASI_TWINX_REAL_LITTLE	RO	Any	—	128-bit atomic LDDA, real address (LE)	(See UA-2007)
$2F_{16}$	ASI_TWINX_NL, ASI_STBI_NL	RW	Any	—	Load: 128-bit atomic load twin extended word from nucleus context, little endian Store: Block initializing store from nucleus context, little endian	(See UA-2007)
80_{16}	ASI_PRIMARY	RW	Any	—	Implicit primary address space	(See UA-2007)
81_{16}	ASI_SECONDARY	RW	Any	—	Implicit secondary address space	(See UA-2007)
82_{16}	ASI_PRIMARY_NO_FAULT	RO	Any	—	Primary address space, no fault	(See UA-2007)
83_{16}	ASI_SECONDARY_NO_ FAULT	RO	Any	—	Secondary address space, no fault	(See UA-2007)
84_{16}–87_{16}			Any	—	*DAE_invalid_asi*	
88_{16}	ASI_PRIMARY_LITTLE	RW	Any	—	Implicit primary address space (LE)	(See UA-2007)
89_{16}	ASI_SECONDARY_LITTLE	RW	Any	—	Implicit secondary address space (LE)	(See UA-2007)

TABLE 9-1 UltraSPARC T2 ASI Usage *(5 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE	RO	Any	—	Primary address space, no fault (LE)	(See UA-2007)
$8B_{16}$	ASI_SECONDARY_NO_FAULT_LITTLE	RO	Any	—	Secondary address space, no fault (LE)	(See UA-2007)
$8C_{16}$–BF_{16}			Any	—	*DAE_invalid_asi*	
$C0_{16}$	ASI_PST8_P		Any	—	Eight 8-bit conditional stores, primary address	(See UA-2007)
$C1_{16}$	ASI_PST8_S		Any	—	Eight 8-bit conditional stores, secondary address	(See UA-2007)
$C2_{16}$	ASI_PST16_P		Any	—	Four 16-bit conditional stores, primary address	(See UA-2007)
$C3_{16}$	ASI_PST16_S		Any	—	Four 16-bit conditional stores, secondary address	(See UA-2007)
$C4_{16}$	ASI_PST32_P		Any	—	Two 32-bit conditional stores, primary address	(See UA-2007)
$C5_{16}$	ASI_PST32_S		Any	—	Two 32-bit conditional stores, secondary address	(See UA-2007)
$C6_{16}$–$C7_{16}$			Any	—	*DAE_invalid_asi*	
$C8_{16}$	ASI_PST8_PL		Any	—	Eight 8-bit conditional stores, primary address, little endian	(See UA-2007)
$C9_{16}$	ASI_PST8_SL		Any	—	Eight 8-bit conditional stores, secondary address, little endian	(See UA-2007)
CA_{16}	ASI_PST16_PL		Any	—	Four 16-bit conditional stores, primary address, little endian	(See UA-2007)
CB_{16}	ASI_PST16_SL		Any	—	Four 16-bit conditional stores, secondary address, little endian	(See UA-2007)
CC_{16}	ASI_PST32_PL		Any	—	Two 32-bit conditional stores, primary address, little endian	(See UA-2007)
CD_{16}	ASI_PST32_SL		Any	—	Two 32-bit conditional stores, secondary address, little endian	(See UA-2007)
CE_{16}–CF_{16}			Any	—	*DAE_invalid_asi*	
$D0_{16}$	ASI_FL8_P		Any	—	8-bit load/store, primary address	(See UA-2007)
$D1_{16}$	ASI_FL8_S		Any	—	8-bit load/store, secondary address	(See UA-2007)

TABLE 9-1 UltraSPARC T2 ASI Usage *(6 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
$D2_{16}$	ASI_FL16_P		Any	—	16-bit load/store, primary address	(See UA-2007)
$D3_{16}$	ASI_FL16_S		Any	—	16-bit load/store, secondary address	(See UA-2007)
$D4_{16}$–$D7_{16}$			Any	—	*DAE_invalid_asi*	
$D8_{16}$	ASI_FL8_PL		Any	—	8-bit load/store, primary address, little endian	(See UA-2007)
$D9_{16}$	ASI_FL8_SL		Any	—	8-bit load/store, secondary address, little endian	(See UA-2007)
DA_{16}	ASI_FL16_PL		Any	—	16-bit load/store, primary address, little endian	(See UA-2007)
DB_{16}	ASI_FL16_SL		Any	—	16-bit load/store, secondary address, little endian	(See UA-2007)
DC_{16}–DF_{16}			Any	—	*DAE_invalid_asi*	
$E0_{16}$	ASI_BLK_COMMIT_PRIMARY	RW	Any	—	64-byte block commit store, primary address	5.3
$E1_{16}$	ASI_BLK_COMMIT_SECONDARY	RW	Any	—	64-byte block commit store, secondary address	5.3
$E2_{16}$	ASI_TWINX_P, ASI_STBI_P	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space Store: Block initializing store, primary address space	(See UA-2007)
$E3_{16}$	ASI_TWINX_S, ASI_STBI_S	RW	Any	—	Load: 128-bit atomic load twin extended word, sedondary address space Store: Block initializing store, sedondary address space	(See UA-2007)
$E4_{16}$–$E9_{16}$			Any	—	*DAE_invalid_ASI*	
EA_{16}	ASI_TWINX_PL, ASI_STBI_PL	RW	Any	—	Load: 128-bit atomic load twin extended word, primary address space, little endian Store: Block initializing store, primary address space, little endian	(See UA-2007)

TABLE 9-1 UltraSPARC T2 ASI Usage *(7 of 7)*

ASI	ASI Name	R/W	VA	Copy per Strand	Description	Section/Page
EB_{16}	ASI_TWINX_PL, ASI_STBI_PL	RW	Any	—	Load: 128-bit atomic load twin extended word, sedondary address space, little endian Store: Block initializing store, sedondary address space, little endian	(See UA-2007)
EC_{16}–EF_{16}			Any	—	*DAE_invalid_asi*	
$F0_{16}$	ASI_BLK_P	RW	Any	—	64-byte block load/store, primary address	5.3
$F1_{16}$	ASI_BLK_S	RW	Any	—	64-byte block load/store, secondary address	5.3
$F2_{16}$–$F7_{16}$			Any	—	*DAE_invalid_asi*	
$F8_{16}$	ASI_BLK_PL	RW	Any	—	64-byte block load/store, primary address (LE)	5.3
$F9_{16}$	ASI_BLK_SL	RW	Any	—	64-byte block load/store, secondary address (LE)	5.3
FA_{16}–FF_{16}			Any	—	*DAE_invalid_asi*	

9.2.1 ASI_REAL, ASI_REAL_LITTLE, ASI_REAL_IO, and ASI_REAL_IO_LITTLE

These ASIs are used to bypass the VA-to-RA translation. For these ASIs, the real address is set equal to the truncated virtual address (that is, RA{39:0} ← VA{39:0}), and the attributes used are those present in the matching TTE. The hypervisor will normally set the TTE attributes for ASI_REAL and ASI_REAL_LITTLE to cacheable (cp = 1) and for ASI_REAL_IO and ASI_REAL_IO_LITTLE to noncacheable, with side effect (cp = 0, e = 1).

9.2.2 ASI_SCRATCHPAD

Each virtual processor has a set of privileged ASI_SCRATCHPAD registers at ASI 20_{16} with VA{63:} = 0_{16}–18_{16}, 30_{16}–38_{16}. These registers are for scratchpad use by privileged software.

UltraSPARC T2 Implementation Note Accesses to VA 20_{16} and 28_{16} are much slower than to the other six scratchpad registers.

CHAPTER **10**

Performance Instrumentation

10.1 SPARC Performance Control Register

Each virtual processor has a privileged Performance Control register. Nonprivileged accesses to this register cause a *privileged_opcode* trap. The Performance Control register contains thirteen fields: hold_ov1, hold_ov0, ov1, sl1, mask1, ov0, sl0, mask0, toe, ht, ut, st, and priv. hold_ov1 and hold_ov0 read as 0 and control whether ov1 and ov0, respectively, are updated on a write. All bits except ov1 and ov0 are always updated on a Performance Control register write. ov1 and ov0 are state bits associated with the PIC.h and PIC.l overflow traps and are provided to allow software to determine which PIC counter has overflowed. sl1 and sl0 controls which events are counted in PIC.h and PIC.l, respectively. mask1 (mask0) is used in conjunction with sl1 (sl0) in determining which set of subevents are counted in PIC.h (PIC.l). toe controls whether a trap is generated when the PIC counter overflows. ut controls whether user-level events are counted. st controls whether supervisor-level events are counted. ht controls whether hypervisor level events are 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.

Note | As the ht bit controls the counting of hyperprivileged events, writes to this bit while privileged are ignored.

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

Bit	Field	Initial Value	R/W	Description
63	hold_ov1	0	Write to 0 or 1, reads as 0	If set to 0 on a write, update ov1 from bit 31 of the write data; else, don't update ov1. In this case ov1 holds its previous value.
62	hold_ov0	0	Write to 0 or 1, reads as 0	If set to 0 on a write, update ov0 from bit 18 of the write data; else, don't update ov0. In this case ov0 holds its previous value.
61:32	—	0	RO	*Reserved*
31	ov1	0	RW	Set to 1 when PIC.h wraps from 2^{32} –1 to 0, or when PIC.h is within --16..-1 inclusively, and an event occurs which causes PIC.h to increment. Once set, ov1 remains set until reset by software.
30:27	sl1	0	RW	Selects 1 of 16 events to be counted for PIC.h as per the following table.
26:19	mask1	0	RW	Mask event for PIC.h as listed in TABLE 10-2.
18	ov0	0	RW	Set to 1 when PIC.l wraps from 2^{32} –1 to 0, or when PIC.l is within --16..-1 inclusively, and an event occurs which causes PIC.l to increment. Once set, ov0 remains set until reset by software.
17:14	sl0	0	RW	Selects one of sixteen events to be counted for PIC.l as per the following table.
13:6	mask0	0	RW	Mask event for PIC.l as listed in TABLE 10-2.
5:4	toe	0	RW	Trap-on-Event: This field controls whether a disrupting trap to hyperprivileged software will occur if the corresponding counter overflows. toe{1} corresponds to ov1, and toe{0} to ov0. Hardware will **and** the value of toe{*i*} with ov{*i*} to produce a trap. Events in event groups 2) and 3) are "precisely" trapped, assuming that PCR.toe = 1 -- TPC will contain the address of an instruction that generated a count event. If PCR.toe = 0 when the counter overflows, TPC will contain the address of the instruction to be executed next when the trap is eventually taken. Events in other event groups are not directly related to the instruction stream; therefore, the TPC may be some number of instructions later than when the overflow event occurred.
3	ht	0	RO	If ht = 1, count events in hyperprivileged mode; otherwise, ignore hyperprivileged mode events.
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 privileged mode; otherwise, ignore privileged 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.

Note that hold_ov1 and hold_ov0 control whether ov1 and ov0, respectively, are updated when a write occurs. All of the 4 combinations of the hold_ov1 and hold_ov0 fields are supported: both, either, or none of the ov1 and ov0 bits can be

updated independently. This allows software to avoid a race condition that may occur, for example, if ov1 is set when the trap handler is entered, then ov0 is set by a counter overflow between the time software reads PCR and resets ov1 prior to leaving the trap handler.

TABLE 10-2 describes the settings of the sl0 and sl1 fields. Note that with the exception of sl = 0, all events correspond to a given strand. Most sl fields have a mask associated with them. Setting multiple mask bits at the same time can lead to multiple events being counted as one event. More details are described in TABLE 10-2.

TABLE 10-2 sl Field Settings *(1 of 4)*

sl	mask	Event	Description
0	—	All strands idle	Count cycles when no strand can be picked for the physical core on which the monitoring strand resides.[2]
1	—	—	*Reserved*
2	01_1	Completed branches	
	02_{16}	Taken branches	Taken branches are always mispredicted[3]
	04_{16}	FGU arithmetic instructions	All FADD, FSUB, FCMP, convert, FMUL, FDIV, FNEG, FABS, FSQRT, FMOV, FPADD, FPSUB, FPACK, FEXPAND, FPMERGE, FMUL8, FMULD8, FALIGNDATA, BSHUFFLE, FZERO, FONE, FSRC, FNOT1, FNOT2, FOR, FNOR, FAND, FNAND, FXOR, FXNOR, FORNOT1, FORNOT2, FANDNOT1, FANDNOT2, PDIST, SIAM.
	08_{16}	Load instructions	
	10_{16}	Store instructions	
	20_{16}	`sethi %hi(fc000`$_{16}$`), %g0`	Software count instructions.
	40_{16}	Other instructions	
	80_{16}	Atomics	Atomics are LDSTUB/A, CASA/XA, SWAP/A
	Any other value 03_{16}– FF_{16}	Any subset of instructions	Count instruction types identified by a 1 in the corresponding mask register bit; e.g., FD_{16} counts all instructions. Certain instructions (e.g., LDSTUB, CAS, SWAP) are decoded as both Load and Store instructions.

TABLE 10-2 sl Field Settings *(2 of 4)*

sl	mask	Event	Description
3	01_{16}	Icache misses	**Note:** This counts only primary instruction cache misses, and does not count duplicate instruction cache misses.[4] Also, only "true" misses are counted. If a thread encounters an I$ miss, but the thread is redirected (due to a branch misprediction or trap, for example) before the line returns from L2 and is loaded into the I$, then the miss is not counted.
	02_{16}	Dcache misses	Note: This counts both primary and duplicate data cache misses.[4]
	04_{16}	—	Undefined operation.
	08_{16}	—	Undefined operation.
	10_{16}	L2 cache instruction misses	
	20_{16}	L2 cache load misses	**Note:** Block loads are treated as one L2 miss event. In reality, each individual load can hit or miss in the L2 since the block load is not atomic.
	03_{16}, 11_{16}, 12_{16}, 13_{16}, 21_{16}, 22_{16}, 23_{16}, 30_{16}, 31_{16}, 32_{16}, 33_{16}	Subset of misses	Count subset of misses identified by a '1' in corresponding mask bit; e.g., 23_{16} counts I-cache, D-cache, and L2 load misses; this counter can advance at most 1 per cycle. **Note:** Instructions that get both an I-Cache miss (or an L2 cache instruction miss) and a D-Cache miss (or L2 cache load miss) count as one event.
	Any other value	—	*Reserved*, Undefined operation.
4	01_{16}	—	*Reserved*
	02_{16}	—	*Reserved*
	04_{16}	ITLB references to L2	For each ITLB miss with hardware tablewalk enabled, count each access the ITLB hardware tablewalk makes to L2.
	08_{16}	DTLB references to L2	For each DTLB miss with hardware tablewalk enabled, count each access the DTLB hardware tablewalk makes to L2.
	10_{16}	ITLB references to L2 which miss in L2	For each ITLB miss with hardware tablewalk enabled, count each access the ITLB hardware tablewalk makes to L2 which misses in L2. **Note:** Depending upon the hardware table walk configuration, each ITLB miss may issue from 1 to 4 requests to L2 to search TSBs.
	20_{16}	DTLB references to L2 which miss in L2	For each DTLB miss with hardware tablewalk enabled, count each access the DTLB hardware tablewalk makes to L2 which misses in L2. **Note:** Depending upon the hardware tablewalk configuration, each DTLB miss may issue from 1 to 4 requests to L2 to search TSBs.
	C_{16}, 14_{16}, 18_{16}, $1C_{16}$, 24_{16}, 28_{16}, $2C_{16}$, 34_{16}, 38_{16} $3C_{16}$	Subset of above events	Count subset of misses identified by a 1 in corresponding mask bit; e.g., 14_{16} counts ITLB and DTLB hardware tablewalk references to L2; this counter can advance at most 1 per cycle. Certain combinations (14_{16}, 28_{16}, 34_{16}, 38_{16}, $3C_{16}$) are likely not useful.
	Any other value	—	*Reserved*. Undefined operation.

TABLE 10-2 sl Field Settings *(3 of 4)*

sl	mask	Event	Description
5			
	01_{16}	Streaming Unit Loads to PCX	Count SPU load operations to L2.
	02_{16}	Streaming Unit Stores to PCX	Count SPU store operations to L2.
	04_{16}	CPU Load to PCX	Count CPU loads to L2.
	08_{16}	CPU I-fetch to PCX	Count I-fetches to L2.
	10_{16}	CPU Store to PCX	Count CPU stores to L2.
	20_{16}	MMU Load to PCX	Count MMU loads to L2.
	Any other value 03_{16}–$3F_{16}$	Subset of PCX requests	Count subset of PCX requests identified by a '1' in corresponding mask bit; e.g., $3F_{16}$counts all PCX requests; this counter increments at most one per cycle.
	40_{16}–FF_{16}	—	*Reserved*
6[1]			
	01_{16}	DES/3DES operations	Increment for each CWQ or ASI operation that uses DES/3DES unit.
	02_{16}	AES operations	Increment for each CWQ or ASI operation that uses AES unit.
	04_{16}	RC4 operations	Increment for each CWQ or ASI operation that uses RC4.
	08_{16}	MD5/SHA-1/SHA-256 operations	Increments for each CWQ or ASI operation that uses MD5, SHA-1, or SHA-256.
	10_{16}	MA operations	Increment for each CWQ or ASI modular arithmetic operation.
	20_{16}	CRC or TCP/IP checksum	Increment for each iSCSI CRC or TCP/IP checksum operation.
	Any other value 03_{16}–$3F_{16}$	Subset of SPU operations	Count the SPU operations whose corresponding mask bit is a 1. Hardware can initiate an MA operation and one of the other operations simultaneously. Note: If MA operations are counted with any other operations, and hardware initiates an MA operation and one of the other operations at the same cycle, the counter increments by 1.
	40_{16}–FF_{16}	—	*Reserved*
7[1]			
	01_{16}	DES/3DES busy cycles	Increment each cycle DES/3DES unit is busy.
	02_{16}	AES busy cycles	Increment each cycle AES unit is busy.
	04_{16}	RC4 busy cycles	Increment each cycle RC4 unit is busy.
	08_{16}	MD5/SHA-1/SHA-256 busy cycles	Increment each cycle MD5, SHA-1, or SHA-256 unit is busy.
	10_{16}	MA busy cycles	Increment each cycle modular arithmetic unit is busy.
	20_{16}	CRC/MPA/checksum	Increment each cycle CRC/MPA/checksum unit is busy.
	Any other value 03_{16}–$3F_{16}$	Unit busy subset	Increment count if any unit whose corresponding mask bit is a 1 is busy; this counter increments at most one per cycle.
	40_{16}–FF_{16}	—	*Reserved*
8-10	—	—	*Reserved*

TABLE 10-2 sl Field Settings *(4 of 4)*

sl	mask	Event	Description
11	04_{16}	ITLB misses	Includes all misses (successful and unsuccessful tablewalks).
	08_{16}	DTLB misses	Includes all misses (successful and unsuccessful tablewalks).
	$0C_{16}$	TLB misses	Count both ITLB and DTLB misses, including successful and unsuccessful tablewalks.
	Any other value	—	*Reserved*. Undefined operation.
12-15	—	—	*Reserved*

1. PCR.UT, PCR.HT, and PCR.ST must all be set in order to properly count events in groups 6 and 7.
2. Unrestricted access to performance events for sl field setting 0 may have security implications since they contain information about other strands. OS software can protect against unrestricted access by setting the PCR.priv bit. Hypervisor software can protect against unrestricted access by not having partitions span an eight-strand boundary.
3. In conjunction with the completed branch count, the taken branch count can be used to compute not-taken prediction accuracy. Also it can be used to sum idle cycles in single-strand mode by assuming a fixed number of pipeline bubble cycles per mispredicted branch.
4. A duplicate miss is a miss for which another thread has already missed in the cache for the line, and the cache fill is pending. UltraSPARC {N2} does not count duplicate I-cache misses but does count duplicate D-cache misses.

10.2 SPARC Performance Instrumentation Counter

Each virtual processor has a Performance Instrumentation Counter register. Access privilege is controlled by the setting of PCR.priv. When PCR.priv = 1 an attempt to access this register in nonprivileged mode causes a *privileged_action* trap.

The PIC counter contains two fields: h and l. The h field counts the event select by PCR.sl1. The l field counts the event selected by PCR.sl0. The ut, st, and ht fields for PCR control which combination of user, supervisor, and/or hypervisor events are counted.

Counter overflow is recorded in the ov0 or ov1 bit of the counter as well as in bit 15 of the SOFTINT register.

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	Programmable event counter, event controlled by PCR.sl1.
31:0	l	0	RW	Programmable event counter, event controlled by PCR.sl0.

CHAPTER 11

Implementation Dependencies

11.1 SPARC V9 General Information

11.1.1 Level-2 Compliance (Impdep #1)

UltraSPARC T2 is designed to meet Level-2 SPARC V9 compliance. It

- Correctly interprets all nonprivileged operations, and
- Correctly interprets all privileged elements of the architecture.

Note | System emulation routines (for example, quad-precision floating-point operations) shipped with UltraSPARC T2 also must be Level-2 compliant.

11.1.2 Unimplemented Opcodes, ASIs, and ILLTRAP

SPARC V9 unimplemented, *reserved*, ILLTRAP opcodes, and instructions with invalid values in *reserved* fields (other than *reserved* FPops) encountered during execution cause an *illegal_instruction* trap. Unimplemented and *reserved* ASI values cause a *DAE_invalid_ASI* trap.

11.1.3 Trap Levels (Impdep #37, 38, 39, 40, 114, 115)

UltraSPARC T2 supports two trap levels; that is, *MAXPTL* = 2. Normal execution is at TL = 0.

A virtual processor normally executes at trap level 0 (`execute_state`, TL = 0). Per SPARC V9, a trap causes the virtual processor to enter the next higher trap level, which is a very fast and efficient process because there is one set of trap state

registers for each trap level. After saving the most important machine states (PC, NPC, PSTATE) on the trap stack at this level, the trap (or error) condition is processed.

11.1.4 Trap Handling (Impdep #16, 32, 33, 35, 36, 44)

UltraSPARC T2 supports precise trap handling for all operations except for deferred and disrupting traps from hardware failures and interrupts. UltraSPARC T2 implements precise traps, interrupts, and exceptions for all instructions, including long-latency floating-point operations. Multiple traps levels are supported, allowing graceful recovery from faults. UltraSPARC T2 can efficiently execute kernel code even in the event of multiple nested traps, promoting strand efficiency while dramatically reducing the system overhead needed for trap handling.

Multiple sets of global registers are provided. 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 T2 are listed in TABLE 6-2 on page 32.

11.1.5 Secure Software

To establish an enhanced security environment, it may be necessary to initialize certain virtual processor states between contexts. Examples of such states are the contents of integer and floating-point register files, condition codes, and state registers. See also *Clean Window Handling (Impdep #102)*.

11.1.6 Operation in Nonprivileged Mode with TL > 0

Operation with PSTATE.priv = 0 and TL > 0 is invalid and will result in an *IAE_privilege_violation* trap on UltraSPARC T2.

11.1.7 Address Masking (Impdep #125)

UltraSPARC T2 follows UltraSPARC Architecture 2007 for PSTATE.am masking. In addition to the masking required by UltraSPARC Architecture 2007, addresses to non-translating ASIs and *REAL* ASIs are masked if PSTATE.am = 1. Translating accesses that bypass translation are also masked if PSTATE.am = 1.

11.2 SPARC V9 Integer Operations

11.2.1 Integer Register File and Window Control Registers (Impdep #2)

UltraSPARC T2 implements an eight-window 64-bit integer register file; that is, *N_REG_WINDOWS* = 8. UltraSPARC T2 truncates values stored in the CWP, CANSAVE, CANRESTORE, CLEANWIN, and OTHERWIN registers to three bits. This includes implicit updates to these registers by SAVE, SAVED, RESTORE, and RESTORED instructions. The most significant two bits of these registers read as zero.

11.2.2 Clean Window Handling (Impdep #102)

SPARC V9 introduced the concept of "clean window" to enhance security and integrity during program execution. A clean window is defined to be a register window that contains either all zeroes or addresses and data that belong to the current context. The CLEANWIN register records the number of available clean windows.

When a SAVE instruction requests a window and there are no more clean windows, a *clean_window* trap is generated. System software needs to clean one or more windows before returning to the requesting context.

11.2.3 Integer Multiply and Divide

Integer multiplications (MULScc, SMUL{cc}, MULX) and divisions (SDIV{cc}, UDIV{cc}, UDIVX) are executed directly in hardware.

11.2.4 MULScc

SPARC V9 does not define the value of xcc and rd{63:32] for MULScc. UltraSPARC T2 sets xcc.n to 0, xcc.z to 1 if rd{63:0} is zero and to 0 if rd{63:0} is not zero, xcc.v to 0, and xcc.c to 0. UltraSPARC T2 sets rd{63:33} to zeros, and sets rd{32} to icc.c (that is, rd{32} is set if there is a carry-out of rd{31}; otherwise, it is cleared).

11.3 SPARC V9 Floating-Point Operations

11.3.1 Subnormal Operands and Results; Nonstandard Operation

UltraSPARC T2 handles some cases of subnormal operands or results directly in hardware and traps on the rest. In the trapping cases, an *fp_exception_other* [fft = unfinished_FPop] trap is signaled and these operations are handled in system software.

Because trapping on subnormal operands and results can be quite costly, UltraSPARC T2 supports the nonstandard result option of the SPARC-V9 architecture. When the FSR.ns bit is set, subnormal operands or results encountered in trapping cases are flushed to zero and the unfinished_FPop floating-point trap is not taken.

11.3.2 Overflow, Underflow, and Inexact Traps (Impdep #3, 55)

UltraSPARC T2 implements precise floating-point exception handling. Underflow is detected before rounding. Prediction of overflow, underflow, and inexact traps for operations as well as prediction of invalid operation is used to simplify the hardware.

> Significant performance degradation may be observed while running with the inexact exception enabled.

11.3.3 Quad-Precision Floating-Point Operations (Impdep #3)

All quad-precision floating-point instructions, listed in TABLE 11-1, cause an *illegal_instruction* trap. These operations are then emulated by system software.

TABLE 11-1 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.

11.3.4 Floating-Point Upper and Lower Dirty Bits in FPRS Register

The FPRS_dirty_upper (du) and FPRS_dirty_lower (dl) bits in the Floating-Point Registers State (FPRS) register are set when an instruction that modifies the corresponding upper or 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.

While SPARC V9 allows FPRS.du and FPRS.dl to be set pessimistically, UltraSPARC T2 only sets FPRS.du or FPRS.dl when an instruction that updates the floating-point register file successfully completes. This implies that floating-point instructions that do not update a floating-point register (for example, an FMOVcc that does not meet the condition or a floating-point operate instruction that takes a trap) leave FPRS.du and FPRS.dl unchanged.

11.3.5 Floating-Point Status Register (FSR) (Impdep #13, 19, 22, 23, 24)

UltraSPARC T2 supports precise-traps and implements all three exception fields (tem, cexc, and aexc) conforming to IEEE Standard 754-1985.

UltraSPARC T2 implements the FSR register according to the definition in UltraSPARC Architecture 2007, with the following implementation-specific clarifications:

- UltraSPARC T2 does not contain an FQ, therefore FSR.qne always reads as 0 and an attempt to read the FQ with an RDPR instruction causes an *illegal_instruction* trap.
- UltraSPARC T2 does not detect the unimplemented_FPop, sequence_error, hardware_error or invalid_fp_register floating-point trap types directly in hardware, therefore does not generate a trap when those conditions occur.

11.4 SPARC V9 Memory-Related Operations

11.4.1 Load/Store Alternate Address Space (Impdep #5, 29, 30)

Supported ASI accesses are listed in *Alternate Address Spaces* on page 47.

11.4.2 Read/Write ASR (Impdep #6, 7, 8, 9, 47, 48)

Supported ASRs are listed in Chapter 3, *Registers*.

11.4.3 MMU Implementation (Impdep #41)

UltraSPARC T2 memory management is based on in-memory Translation Storage Buffers (TSBs) backed by a Software Translation Table. See Chapter 12, *Memory Management Unit* for more details.

11.4.4 FLUSH and Self-Modifying Code (Impdep #122)

FLUSH is needed to synchronize code and data spaces after code space is modified during program execution. FLUSH is described in *Memory Synchronization: MEMBAR and FLUSH* on page 123. On UltraSPARC T2, the FLUSH effective address is ignored, and as a result, FLUSH cannot cause a *DAE_invalid_ASI* trap.

Note | SPARC V9 specifies that the FLUSH instruction has no latency on the issuing virtual processor. In other words, a store to instruction space prior to the FLUSH instruction is visible immediately after the completion of FLUSH. When a flush is performed, UltraSPARC T2 guarantees that earlier code modifications will be visible across the whole system.

11.4.5 PREFETCH{A} (Impdep #103, 117)

For UltraSPARC T2, PREFETCH{A} instructions follow TABLE 11-2 based on the fcn value. All prefetches in UltraSPARC T2 are of the "weak" variety (that is, on an MMU miss, the prefetch is dropped) so the only trap generated by prefetch is *illegal_instruction* (for fcn = 5_{16}–F_{16}).

TABLE 11-2 PREFETCH{A} Variants in UltraSPARC T2

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	
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.

TABLE 11-2 PREFETCH{A} Variants in UltraSPARC T2

fcn	Prefetch Function	Action
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}	Invalidate cache entry	No operation for PREFETCHA. For Prefetch, if executed in user or privileged mode, no operation. If executed while hyperprivileged, invalidate cache line from Level 2 cache (writing back to memory if dirty) leaving Level 2 cache line invalid.
19_{16}–$1F_{16}$	—	No operation

11.4.6 LDD/STD Handling (Impdep #107, 108)

LDD and STD instructions are directly executed in hardware.

Note | LDD/STD are deprecated in SPARC V9. In UltraSPARC T2 it is more efficient to use LDX/STX for accessing 64-bit data. LDD/STD take longer to execute than two 32- or 64-bit loads/stores.

11.4.7 FP mem_address_not_aligned (Impdep #109, 110, 111, 112)

LDDF{A}/STDF{A} cause an *LDDF_/STDF_ mem_address_not_aligned* trap if the effective address is 32-bit aligned but not 64-bit (doubleword) aligned.

LDQF{A}/STQF{A} are not directly executed in hardware; they cause an *illegal_instruction* trap.

11.4.8 Supported Memory Models (Impdep #113, 121)

UltraSPARC T2 supports only the TSO memory model, although certain specific operations such as block loads and stores operate under the RMO memory model. See Chapter 8, Section 8.2. Supported Memory Models.”.

11.4.9 Implicit ASI When TL > 0 (Impdep #124)

UltraSPARC T2 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.

11.5 Non-SPARC V9 Extensions

11.5.1 Cache Subsystem

UltraSPARC T2 contains one or more levels of cache. The cache subsystem architecture is described in Appendix D, *Caches and Cache Coherency*.

11.5.2 Block Memory Operations

UltraSPARC T2 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 25.

11.5.3 Partial Stores

UltraSPARC T2 supports 8-/16-/32-bit partial stores to memory. See *Block Load and Store Instructions* on page 25.

11.5.4 Short Floating-Point Loads and Stores

UltraSPARC T2 supports 8-/16-bit loads and stores to the floating-point registers.

11.5.5 Load Twin Extended Word

UltraSPARC T2 supports 128-bit atomic load operations to a pair of integer registers.

11.5.6 UltraSPARC T2 Instruction Set Extensions (Impdep #106)

The UltraSPARC T2 processor supports VIS 2.0. VIS instructions are designed to enhance graphics functionality and improve the efficiency of memory accesses.

Unimplemented IMPDEP1 and IMPDEP2 opcodes encountered during execution cause an *illegal_instruction* trap.

11.5.7 Performance Instrumentation

UltraSPARC T2 performance instrumentation is described in Chapter 10, *Performance Instrumentation*.

CHAPTER 12

Memory Management Unit

This chapter provides detailed information about the UltraSPARC T2 Memory Management Unit. It describes the internal architecture of the MMU and how to program it.

12.1 Translation Table Entry (TTE)

The Translation Table Entry 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.

. TABLE 12-1 shows the sun4v TTE tag format.

TABLE 12-1 TTE Tag Format

Bit	Field	Description
63:61	—	*Reserved*
60:48	context	The 13-bit context identifier associated with the TTE.
47:42	—	*Reserved*
41:0	va	Virtual Address Tag{63:22}. The virtual page number. Bits 21 through 13 are not maintained in the tag, since these bits are used to index the smallest TSB (512 entries). **NOTE:** Hardware only supports a 48-bit VA.

The sun4v TTE data format is shown in TABLE 12-2.

TABLE 12-2 TTE Data Format

Bit	Field	Description
63	v	Valid. If the Valid bit is set, the remaining fields of the TTE are meaningful.
62	nfo	No-fault-only. If this bit is set, loads with ASI_PRIMARY_NO_FAULT{_LITTLE}, ASI_SECONDARY_NO_FAULT{_LITTLE} are translated. Any other DMMU access will trap with a *DAE_nfo_page* trap. For the IMMU, if the nfo bit is set, an *iae_nfo_page* trap will be taken.
61:56	soft2	soft2 and soft are software-defined fields, provided for use by the operating system. Software fields are not implemented in the UltraSPARC T2 TLB. soft and soft2 fields may be written with any value; they read from the TLB as zero, with the exception of soft{61}, which contains the TLB data parity bit.
55:13	ra	The real page[1] number. For UltraSPARC T2, a 40-bit real address range is supported by the hardware tablewalker, and bits {55:40} should always be zero.
12	ie	Invert endianess. If this bit is set, accesses to the associated page are processed with inverse endianness from what is specified by the instruction (big-for-little and little-for-big). 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. **Note:** This bit is intended to be set primarily for noncacheable accesses.
11	e	Side effect. If this bit is set, noncacheable memory accesses other than block loads and stores are strongly ordered against other e bit accesses, and noncacheable stores are not merged. This bit should be set for pages that map I/O devices having side effects. Note, however, that the e bit does not prevent normal instruction prefetching. For the IMMU, the e bit in the TTE is written into the ITLB, but ignored during ITLB operation. The value of the e bit written into the ITLB will be read out on an ITLB Data Access read. **NOTE:** The e bit does not force an uncacheable access. It is expected, but not required, that the cp and cv bits will be set to zero when the e bit is set.
10:9	cp, cv	The cacheable-in-physically-indexed-cache and cacheable-in-virtually-indexed-cache (cp, cv) bits determine the placement of data in UltraSPARC T2 caches, according to TABLE 12-3. The MMU does not operate on the cacheable bits, but merely passes them through to the cache subsystem. The cv bit is ignored by UltraSPARC T2, and is not written into the TLBs and returns zero on a Data Access read.

TABLE 12-3 Cacheable Field Encoding (from TSB)

Cacheable (cp:cv)	Meaning of TTE When Placed in: iTLB (I-cache PA-Indexed)	Meaning of TTE When Placed in: dTLB (D-cache PA-Indexed)
0x	Cacheable L2 cache only	Cacheable L2 cache only
1x	Cacheable L2 cache, I-cache	Cacheable L2 cache, D-cache

TABLE 12-2 TTE Data Format *(Continued)*

Bit	Field	Description
8	p	Privileged. If the p bit is set, only privileged software can access the page mapped by the TTE. If the p bit is set and an access to the page is attempted when PSTATE.priv = 0, the MMU will signal an *IAE_privilege_violation* or *DAE_privilege_violation* trap.
7	ep	Executable. If the ep bit is set, the page mapped by this TTE has execute permission granted. Otherwise, execute permission is not granted and the hardware table-walker will not load the ITLB with a TTE with ep = 0. For the IMMU and DMMU, the ep bit in the TTE is not written into the TLB, and returns zero on a Data Access read.
6	w	Writable. If the w bit is set, the page mapped by this TTE has write permission granted. Otherwise, write permission is not granted and the MMU will cause a trap if a write is attempted. 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.
5:4	soft	(see soft2, above)
3:0	size	The page size of this entry, encoded as shown in TABLE 12-4.

TABLE 12-4 Size Field Encoding (from TTE)

Size{2:0}	Page Size
0000	8 KB
0001	64 KB
0010	Reserved
0011	4 MB
0100	Reserved
0101	256 MB
0110-1111	Reserved

1. sun4v supports translation from virtual addresses (VA) to real addresses (RA. Privileged code manages the VA-to-RA translations..

12.2 Translation Storage Buffer (TSB)

A TSB is an array of TTEs managed entirely by software. It serves as a cache of the Software Translation table

A TSB is arranged as a direct-mapped cache of TTEs.

The TSB exists as a normal data structure in memory and therefore may be cached. 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 12-1 shows the 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.

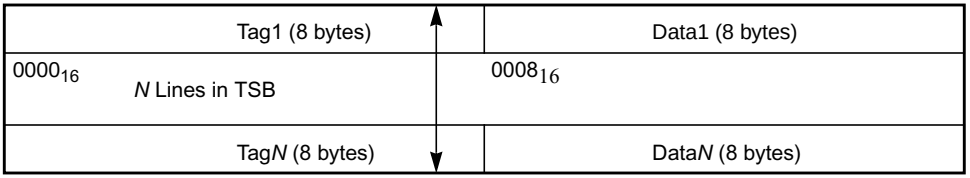


FIGURE 12-1 TSB Organization

12.3 MMU-Related Faults and Traps

12.3.1 *IAE_privilege_violation* Trap

The I-MMU detects a privilege violation for an instruction fetch; that is, an attempted access to a privileged page when PSTATE.priv = 0.

12.3.2 *IAE_nfo_page* Trap

The I-MMU detects an access to a page marked with the nfo (no-fault-only) bit.

12.3.3 *instruction_address_range* Trap

The *instruction_address_range* occurs when the virtual address out of range and PSTATE.am = 0 (see *48-bit Virtual and Real Address Spaces* on page 45).

12.3.4 *instruction_real_range* Trap

The *instruction_real_range* trap occurs when the Real address out of range (see *48-bit Virtual and Real Address Spaces* on page 45).

12.3.5 *DAE_privilege_violation* Trap

The D-MMU detects a privilege violation for a data access; that is, an attempted access to a privileged page when PSTATE.priv = 0.

12.3.6 *DAE_side_effect_page* Trap

A speculative (nonfaulting) load instruction issued to a page marked with the side-effect (e) bit = 1.

12.3.7 *DAE_nc_page* Trap

An atomic instruction (including 128-bit atomic load) issued to a memory address marked uncacheable; for example,, with cp = 0.

Implementation Note For UltraSPARC T2, cp only controls cacheability in the primary cache, not the shared secondary, and thus the hardware supports the ability to complete an atomic operation for pages with the cp bit = 0 as long as the secondary cache is enabled. However, to keep UltraSPARC T2 compliant with the UltraSPARC Architecture 2006 specification, the *DAE_nc_page* trap is generated when an atomic is issued to a memory address marked with cp = 0.

12.3.8 *DAE_invalid_asi* Trap

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).

12.3.9 *DAE_nfo_page* Trap

An access with an ASI other than ASI_{PRIMARY,SECONDARY}_NO_FAULT{_LITTLE} to a page marked with the nfo (no-fault-only) bit.

12.3.10 *privileged_action* Trap

This trap occurs when an access is attempted using a *restricted* ASI while in non-privileged mode (PSTATE.priv = 0).

12.3.11 **_mem_address_not_aligned* Traps

The *lddf_mem_address_not_aligned*, *stdf_mem_address_not_aligned*, and *mem_address_not_aligned* traps occur when a load, store, atomic, or JMPL/RETURN instruction with a misaligned address is executed.

12.4 MMU Operation Summary

TABLE 12-7 summarizes the behavior of the D-MMU for noninternal ASIs using tabulated abbreviations. TABLE 12-8 summarizes the behavior of the I-MMU. In each case, and for all conditions, the behavior of the MMU is given by one of the abbreviations in TABLE 12-5. TABLE 12-6 lists abbreviations for ASI types.

TABLE 12-5 Abbreviations for MMU Behavior

Abbreviation	Meaning
ok	Normal translation
dasi	*DAE_invalid_asi* trap
dpriv	*DAE_privilege_violation* trap
dse	*DAE_side_effect_page* trap
dprot	*fast_data_access_protection* trap
iexc	*IAE_privilege_violation* trap

TABLE 12-6 Abbreviations for ASI Types

Abbreviation	Meaning
NUC	`ASI_NUCLEUS*`
PRIM	Any ASI with PRIMARY translation, except `*NO_FAULT`
SEC	Any ASI with SECONDARY translation, except `*NO_FAULT`
PRIM_NF	`ASI_PRIMARY_NO_FAULT*`
SEC_NF	`ASI_SECONDARY_NO_FAULT*`
U_PRIM	`ASI_*_AS_IF_USER_PRIMARY*`
U_SEC	`ASI_*_AS_IF_USER_SECONDARY*`
U_PRIV	`ASI_*_AS_IF_PRIV_*`
REAL	`ASI_*REAL*`

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.

TABLE 12-7 and TABLE 12-8 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 T2 internal registers other than LDXA, LDFA, STDFA or STXA; the MMU signals a *DAE_invalid_asi* trap for this case.

- Attempted access using a restricted ASI in nonprivileged mode; the MMU signals a *privileged_action* trap for this case. Attempted use of a hyperprivileged ASI in privileged mode; the MMU also signals *privileged_action* trap for this case.
- An atomic instruction (including 128-bit atomic load) issued to a memory address marked uncacheable in a physical cache (that is, with cp = 0 or pa{39} = 1); the MMU signals a *DAE_nc_page* trap for this case.
- A data access with an ASI other than ASI_{PRIMARY,SECONDARY}_NO_FAULT{_LITTLE} or an instruction access to a page marked with the nfo (no-fault-only) bit; the MMU signals a *DAE_nfo_page* or *IAE_nfo_page* trap for this case.
- An instruction fetch to a memory address marked non-executable (ep = 0). This is checked when Hardware Tablewalk attempts to load the I-MMU, and an *IAE_unauth_access* trap is taken instead.
- Real address out of range; the MMU signals an *instruction_real_range* trap for this case.
- Virtual address out of range and PSTATE.am is not set; the MMU signals an *instruction_address_range* trap for this case.

TABLE 12-7 D-MMU Operations for Normal ASIs

Condition					Behavior			
Opcode	**priv Mode**	**ASI**	**w**		**e = 0 p = 0**	**e = 0 p = 1**	**e = 1 p = 0**	**e = 1 p = 1**
Load	non-privileged	PRIM, SEC	—		ok	dpriv	ok	dpriv
		PRIM_NF, SEC_NF	—		ok	dpriv	dse	dpriv
	privileged	PRIM, SEC, NUC	—		ok			
		PRIM_NF, SEC_NF	—		ok		dse	
		U_PRIM, U_SEC	—		ok	dpriv	ok	dpriv
		REAL	—		ok			
FLUSH	non-privileged		—		ok			
	privileged		—		ok			
Store or Atomic	non-privileged	PRIM, SEC	0		dprot	dpriv	dprot	dpriv
			1		ok	dpriv	ok	dpriv
	privileged	PRIM, SEC, NUC	0		dprot			
			1		ok			
		U_PRIM, U_SEC	0		dprot	dpriv	dprot	dpriv
			1		ok	dpriv	ok	dpriv
		REAL	0		dprot			
			1		ok			

TABLE 12-8 I-MMU Operations

Condition	Behavior	
priv Mode	P = 0	P =
nonprivileged	ok	iexc
privileged	ok	

See *Alternate Address Spaces* on page 47 for a summary of the UltraSPARC T2 ASI map.

12.5 Translation

12.5.1 Instruction Translation

12.5.1.1 Instruction Prefetching

UltraSPARC T2 fetches instructions sequentially (including delay slots). UltraSPARC T2 fetches delay slots before the branch is resolved (before whether the delay slot will be annulled is known). UltraSPARC T2 also fetches the target of a DCTI before the delay slot executes.

12.5.2 Data Translation

TABLE 12-9 DMMU Translation *(1 of 4)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
00_{16}–03_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
04_{16}	ASI_NUCLEUS	*privileged_action*	VA → PA	
05_{16}–$0B_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$0C_{16}$	ASI_NUCLEUS_LITTLE	*privileged_action*	VA → PA	
$0D_{16}$–$0F_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
10_{16}	ASI_AS_IF_USER_PRIMARY	*privileged_action*	VA → PA	
11_{16}	ASI_AS_IF_USER_SECONDARY	*privileged_action*	VA → PA	

TABLE 12-9 DMMU Translation *(2 of 4)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
12_{16}–13_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
14_{16}	ASI_REAL	*privileged_action*	RA → PA	
15_{16}	ASI_REAL_IO	*privileged_action*	RA → PA	
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY	*privileged_action*	VA → PA	
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY	*privileged_action*	VA → PA	
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE	*privileged_action*	VA → PA	
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE	*privileged_action*	VA → PA	
$1A_{16}$–$1B_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$1C_{161}$	ASI_REAL_LITTLE	*privileged_action*	RA → PA	
$1D_{16}$	ASI_REAL_IO_LITTLE	*privileged_action*	RA → PA	
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE	*privileged_action*	VA → PA	
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE	*privileged_action*	VA → PA	
20_{16}	ASI_SCRATCHPAD	*privileged_action*	nontranslating	
21_{16}	ASI_MMU	*privileged_action*	nontranslating	
22_{16}	ASI_TWINX_AIUP, ASI_STBI_AIUP	*privileged_action*	VA → PA	
23_{16}	ASI_TWINX_AIUS, ASI_STBI_AIUS	*privileged_action*	VA → PA	
24_{16}	ASI_TWINX	*privileged_action*	VA → PA	
25_{16}	ASI_QUEUE	*privileged_action*	nontranslating	
26_{16}	ASI_TWINX_REAL	*privileged_action*	RA → PA	
27_{16}	ASI_TWINX_NUCLEUS, ASI_STBI_N	*privileged_action*	VA → PA	
28_{16}–29_{16}	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$2A_{16}$	ASI_TWINX_AIUPL, ASI_STBI_AIUPL	*privileged_action*	VA → PA	
$2B_{16}$	ASI_TWINX_AIUSL, ASI_STBI_AIUSL	*privileged_action*	VA → PA	
$2C_{16}$	ASI_TWINX_LITTLE	*privileged_action*	VA → PA	
$2D_{16}$	*Reserved*	*privileged_action*	*DAE_invalid_asi*	
$2E_{16}$	ASI_TWINX_REAL_LITTLE	*privileged_action*	RA → PA	

TABLE 12-9 DMMU Translation *(3 of 4)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
$2F_{16}$	ASI_TWINX_NL, ASI_STBI_NL	*privileged_action*	VA → PA	
80_{16}	ASI_PRIMARY	VA → PA	VA → PA	
81_{16}	ASI_SECONDARY	VA → PA	VA → PA	
82_{16}	ASI_PRIMARY_NO_FAULT	VA → PA	VA → PA	
83_{16}	ASI_SECONDARY_NO_FAULT	VA → PA	VA → PA	
84_{16}–87_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
88_{16}	ASI_PRIMARY_LITTLE	VA → PA	VA → PA	
89_{16}	ASI_SECONDARY_LITTLE	VA → PA	VA → PA	
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE	VA → PA	VA → PA	
$8B_{16}$	ASI_SECONDARY_NO_FAULT_ LITTLE	VA → PA	VA → PA	
$8C_{16}$–BF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$C0_{16}$	ASI_PST8_P	VA → PA	VA → PA	
$C1_{16}$	ASI_PST8_S	VA → PA	VA → PA	
$C2_{16}$	ASI_PST16_P	VA → PA	VA → PA	
$C3_{16}$	ASI_PST16_S	VA → PA	VA → PA	
$C4_{16}$	ASI_PST32_P	VA → PA	VA → PA	
$C5_{16}$	ASI_PST32_S	VA → PA	VA → PA	
$C6_{16}$–$C7_{16}$	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$C8_{16}$	ASI_PST8_PL	VA → PA	VA → PA	
$C9_{16}$	ASI_PST8_SL	VA → PA	VA → PA	
CA_{16}	ASI_PST16_PL	VA → PA	VA → PA	
CB_{16}	ASI_PST16_SL	VA → PA	VA → PA	
CC_{16}	ASI_PST32_PL	VA → PA	VA → PA	
CD_{16}	ASI_PST32_SL	VA → PA	VA → PA	
CE_{16}–CF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$D0_{16}$	ASI_FL8_P	VA → PA	VA → PA	
$D1_{16}$	ASI_FL8_S	VA → PA	VA → PA	
$D2_{16}$	ASI_FL16_P	VA → PA	VA → PA	
$D3_{16}$	ASI_FL16_S	VA → PA	VA → PA	
$D4_{16}$–$D7_{16}$		*DAE_invalid_asi*	*DAE_invalid_asi*	
$D8_{16}$	ASI_FL8_PL	VA → PA	VA → PA	

TABLE 12-9 DMMU Translation *(4 of 4)*

ASI Value (hex)	ASI NAME	Translation		
		Nonprivileged	Privileged	Hypervisor
$D9_{16}$	ASI_FL8_SL	VA → PA	VA → PA	
DA_{16}	ASI_FL16_PL	VA → PA	VA → PA	
DB_{16}	ASI_FL16_SL	VA → PA	VA → PA	
DC_{16}–DF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$E0_{16}$	ASI_BLK_COMMIT_PRIMARY	VA → PA	VA → PA	
$E1_{16}$	ASI_BLK_COMMIT_SECONDARY	VA → PA	VA → PA	
$E2_{16}$	ASI_TWINX_P, ASI_STBI_P	VA → PA	VA → PA	
$E3_{16}$	ASI_TWINX_S, ASI_STBI_S	VA → PA	VA → PA	
$E4_{16}$–$E9_{16}$	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
EA_{16}	ASI_TWINX_PL, ASI_STBI_PL	VA → PA	VA → PA	
EB_{16}	ASI_TWINX_PL, ASI_STBI_PL	VA → PA	VA → PA	
EC_{16}-EF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$F0_{16}$	ASI_BLK_PRIMARY	VA → PA	VA → PA	
$F1_{16}$	ASI_BLK_SECONDARY	VA → PA	VA → PA	
$F2_{16}$-$F7_{16}$	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	
$F8_{16}$	ASI_BLK_PRIMARY_LITTLE	VA → PA	VA → PA	
$F9_{16}$	ASI_BLK_SECONDARY_LITTLE	VA → PA	VA → PA	
FA_{16}-FF_{16}	*Reserved*	*DAE_invalid_asi*	*DAE_invalid_asi*	

12.6 Compliance With the SPARC V9 Annex F

The UltraSPARC T2 MMU complies completely with the SPARC V9 MMU Requirements described in Annex F of the *The SPARC Architecture Manual, Version 9*. TABLE 12-10 shows how various protection modes can be achieved, if necessary, through the presence or absence of a translation in the I- or D-MMU.

TABLE 12-10 MMU Compliance With SPARC V9 Annex F Protection Mode

Condition			
TTE in D-MMU	**TTE in I-MMU**	**Writable Attribute Bit**	**Resultant Protection Mode**
Yes	No	0	Read-only
No	Yes	Don't Care	Execute-only
Yes	No	1	Read/Write
Yes	Yes	0	Read-only/Execute
Yes	Yes	1	Read/Write/Execute

12.7 MMU Internal Registers and ASI Operations

12.7.1 Accessing MMU Registers

All internal MMU registers can be accessed directly by the virtual processor through ASIs defined by UltraSPARC T2.

See Section 12.5 for details on the behavior of the MMU during all other UltraSPARC T2 ASI accesses.

Note | STXA to an MMU register *does not* require any subsequent instructions such as a MEMBAR #Sync, FLUSH, DONE, or RETRY before the register effect will be visible to load / store / atomic accesses. UltraSPARC T2 resolves all MMU register hazards via an automatic synchronization on all MMU register writes.

If the low order three bits of the VA are non-zero in an LDXA/STXA to/from these registers, a *mem_address_not_aligned* trap occurs. Writes to read-only, reads to write-only, illegal ASI values, or illegal VA for a given ASI may cause a *DAE_invalid_asi* trap.

Caution | UltraSPARC T2 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.

TABLE 12-11 UltraSPARC T2 MMU Internal Registers and ASI Operations

I-MMU ASI	D-MMU ASI	VA{63:0}	Access	Register or Operation Name
21_{16}		8_{16}	Read/Write	Primary Context 0 register
—	21_{16}	10_{16}	Read/Write	Secondary Context 0 register
21_{16}		108_{16}	Read/Write	Primary Context 1 register
—	21_{16}	110_{16}	Read/Write	Secondary Context 1 register

12.7.2 Context Registers

UltraSPARC T2 supports a pair of primary and a pair of secondary context registers per strand, which are shared by the I- and D-MMUs. Primary Context 0 and Primary Context 1 are the primary context registers, and a TLB entry for a translating primary ASI can match the context field with either Primary Context 0 or Primary Context 1 to produce a TLB hit. Secondary Context 0 and Secondary Context 1 are the secondary context registers, and a TLB entry for a translating secondary ASI can match the context field with either Secondary Context 0 or Secondary Context 1 to produce a TLB hit.

Compatibility Note | To maintain backward compatibility with software designed for a single primary and single secondary context register, writes to Primary (Secondary) Context 0 Register also update Primary (Secondary) Context 1 Register.

The Primary Context 0 and Primary Context 1 registers are defined as shown in FIGURE 12-2, where pcontext is the context value for the primary address space.

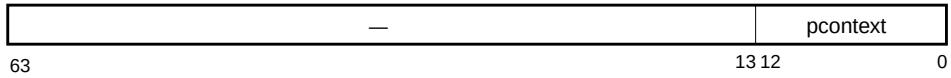


FIGURE 12-2 Primary Context 0/1 register

The Secondary Context 0 and Secondary Context 1 Registers are defined in FIGURE 12-3, where scontext is the context value for the secondary address space.

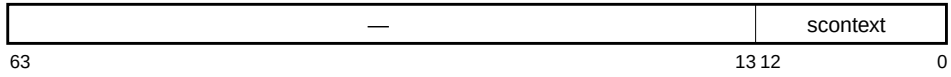


FIGURE 12-3 Secondary Context 0/1 Register

The contents of the Nucleus Context register are hardwired to the value zero:

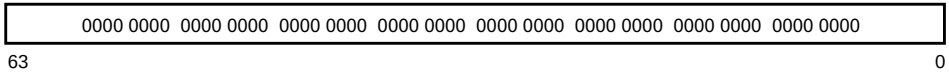


FIGURE 12-4 Nucleus Context Register

APPENDIX **A**

Programming Guidelines

A.1 Multithreading

In UltraSPARC T2, each physical core contains eight strands. The strands are divided into two thread groups, with strands 0–3 occupying one thread group and strands 4–7 occupying the other.

Within a thread group, among the available strands, the least recently picked strand is selected for execution every cycle. Thus, up to two instructions can be picked each cycle.

Since each physical core has only one load/store unit and one floating-point and graphics unit, only one load/store or FGU instruction may be picked each cycle. One thread group can issue a load/store instruction while the other thread group issues an FGU instruction. Arbitrating between the two thread groups is done with a least-recently-picked mechanism, to ensure fairness.

Since context switching is built into the UltraSPARC T2 pipeline (via the D and P stages), strands are switched each cycle with no pipeline stall penalty (except when resource collisions occur, such as when both thread groups require the load/store unit or the FGU).

In normal operation, UltraSPARC T2 speculates that most control-transfer instructions will be “not taken” and that loads hit in the L1 data cache. An enable bit, accessible to hyperprivileged software, controls whether UltraSPARC T2 speculates on these instructions or not.

The following instructions change a strand from available to unavailable until hardware determines that their input/execution requirements can be satisfied:

- CALL, DONE, RETRY, JMPL
- LDFSR, LDXFSR, STFSR, STXFSR
- All WRPR, WR
- All RDPR, RD
- SAVE(D), RESTORE(D), RETURN, FLUSHW (all register-window management)

- All MUL and DIV
- MULX, UMUL, SMUL, POPC
- MEMBAR#, FLUSH
- FCMP
- All memory operations to/from alternate space
- All atomic (load-store) operations
- PREFETCH

If speculation is not enabled, the following instruction types also change a strand from available to unavailable until hardware determines that their execution requirements can be satisfied:

- All control transfer instructions
- All loads

A.2 Instruction Latency

TABLE A-1 lists the minimum single-strand instruction latencies for UltraSPARC T2. When multiple strands are executing, some or much of the additional latency for multicycle instructions will be overlapped with execution of the additional strands.

TABLE A-1 UltraSPARC T2 Instruction Latencies *(1 of 8)*

Opcode	Description	Latency	Notes
ADD (ADDcc)	Add (and modify condition codes)	1	
ADDC (ADDCcc)	Add with carry (and modify condition codes)	1	
ALIGNADDRESS	Calculate address for misaligned data access	1	
ALIGNADDRESSL	Calculate address for misaligned data access (little-endian)	1	
ALLCLEAN	Mark all windows as clean	25	
AND (ANDcc)	Logical **and** (and modify condition codes)	1	
ANDN (ANDNcc)	Logical **and not** (and modify condition codes)	1	
ARRAY{8,16,32}	3-D address to blocked byte address conversion	6	
Bicc	Branch on integer condition codes	1 not-taken, 6 taken	
BMASK	Write the GSR.mask field	25	
BPcc	Branch on integer condition codes with prediction	1 not-taken, 6 taken	

TABLE A-1 UltraSPARC T2 Instruction Latencies *(2 of 8)*

Opcode	Description	Latency	Notes
BPr	Branch on contents of integer register with prediction	1 not-taken, 6 taken	
BSHUFFLE	Permute bytes as specified by the GSR.mask field	6	
CALL	Call and link	6	
CASA	Compare and swap word in alternate space	20-30	Done in L2 cache
CASXA	Compare and swap doubleword in alternate space	20-30	Done in L2 cache
DONE	Return from trap	6	
EDGE{8,16,32}{L}{N}	Edge boundary processing {little-endian} {non-condition-code altering}	6	
FABS(s,d)	Floating-point absolute value	6	
FADD(s,d)	Floating-point add	6	
FALIGNDATA	Perform data alignment for misaligned data	6	
FANDNOT1{s}	Negated src1 **and** src2 (single precision)	6	
FANDNOT2{s}	src1 **and** negated src2 (single precision)	6	
FAND{s}	Logical **and** (single precision)	6	
FBPfcc	Branch on floating-point condition codes with prediction	1 not-taken, 6 taken	
FBfcc	Branch on floating-point condition codes	1 not-taken, 6 taken	
FCMP(s,d)	Floating-point compare	6	
FCMPE(s,d)	Floating-point compare (exception if unordered)	6	
FCMPEQ{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 = src2	6	
FCMPGT{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 > src2	6	
FCMPLE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 ≤ src2	6	
FCMPNE{16,32}	Four 16-bit / two 32-bit compare: set integer dest if src1 ≠ src2	6	
FDIV(s,d)	Floating-point divide	19 SP, 33 DP	
FEXPAND	Four 8-bit to 16-bit expand	6	
FiTO(s,d)	Convert integer to floating-point	6	

TABLE A-1 UltraSPARC T2 Instruction Latencies *(3 of 8)*

Opcode	Description	Latency	Notes
FLUSH	Flush instruction memory	variable	
FLUSHW	Flush register windows	25	
FMOV(s,d)	Floating-point move	6	
FMOV(s,d)cc	Move floating-point register if condition is satisfied	6	
FMOV(s,d)R	Move floating-point register if integer register contents satisfy condition	6	
FMUL(s,d)	Floating-point multiply	6	
FMUL8SUx16	Signed upper 8- x 16-bit partitioned product of corresponding components	6	
FMUL8ULx16	Unsigned lower 8- x 16-bit partitioned product of corresponding components	6	
FMUL8x16	8- x 16-bit partitioned product of corresponding components	6	
FMUL8x16AL	Signed lower 8- x 16-bit lower α partitioned product of 4 components	6	
FMUL8x16AU	Signed upper 8- x 16-bit lower α partitioned product of 4 components	6	
FMULD8SUx16	Signed upper 8- x 16-bit multiply → 32-bit partitioned product of components	6	
FMULD8ULx16	Unsigned lower 8- x 16-bit multiply → 32-bit partitioned product of components	6	
FNAND{s}	Logical **nand** (single precision)	6	
FNEG(s,d)	Floating-point negate	6	
FNOR{s}	Logical **nor** (single precision)	6	
FNOT1{s}	Negate (1's complement) src1 (single precision)	6	
FNOT2{s}	Negate (1's complement) src2 (single precision)	6	
FONE{s}	One fill (single precision)	6	
FORNOT1{s}	Negated src1 **or** src2 (single precision)	6	
FORNOT2{s}	src1 **or** negated src2 (single precision)	6	
FOR{s}	Logical **or** (single precision)	6	
FPACKFIX	Two 32-bit to 16-bit fixed pack	6	
FPACK{16,32}	Four 16-bit/two 32-bit pixel pack	6	
FPADD{16,32}{s}	Four 16-bit/two 32-bit partitioned add (single precision)	6	
FPMERGE	Two 32-bit to 64-bit fixed merge	6	

TABLE A-1 UltraSPARC T2 Instruction Latencies *(4 of 8)*

Opcode	Description	Latency	Notes
FPSUB{16,32}{s}	Four 16-bit/two 32-bit partitioned subtract (single precision)	6	
FsMULd	Floating-point multiply single to double	6	
FSQRT(s,d)	Floating-point square root	19 SP, 33 DP	
FSRC1{s}	Copy src1 (single precision)	6	
FSRC2{s}	Copy src2 (single precision)	6	
F(s,d)TO(s,d)	Convert between floating-point formats	6	
F(s,d)TOi	Convert floating point to integer	6	
F(s,d)TOx	Convert floating point to 64-bit integer	6	
FSUB(s,d)	Floating-point subtract	6	
FXNOR{s}	Logical **xnor** (single precision)	6	
FXOR{s}	Logical **xor** (single precision)	6	
FxTO(s,d)	Convert 64-bit integer to floating-point	6	
FZERO{s}	Zero fill (single precision)	6	
ILLTRAP	Illegal instruction		
INVALW	Mark all windows as CANSAVE	6	
JMPL	Jump and link	6	
LDBLOCKF	64-byte block load	32	
LDD	Load doubleword	3	
LDDA	Load doubleword from alternate space	variable	
LDDF	Load double floating-point	3	
LDDFA	Load double floating-point from alternate space	variable	
LDF	Load floating-point	3	
LDFA	Load floating-point from alternate space	variable	
LDFSR	Load floating-point state register lower	1-8	pre-sync to previous FGU op from that thread
LDSB	Load signed byte	3	
LDSBA	Load signed byte from alternate space	variable	
LDSH	Load signed halfword	3	

TABLE A-1 UltraSPARC T2 Instruction Latencies *(5 of 8)*

Opcode	Description	Latency	Notes
LDSHA	Load signed halfword from alternate space	variable	
LDSTUB	Load-store unsigned byte	3	
LDSTUBA	Load-store unsigned byte in alternate space	variable	
LDSW	Load signed word	3	
LDSWA	Load signed word from alternate space	variable	
LDUB	Load unsigned byte	3	
LDUBA	Load unsigned byte from alternate space	variable	
LDUH	Load unsigned halfword	3	
LDUHA	Load unsigned halfword from alternate space	variable	
LDUW	Load unsigned word	3	
LDUWA	Load unsigned word from alternate space	variable	
LDX	Load extended	3	
LDXA	Load extended from alternate space	variable	
LDXFSR	Load extended floating-point state register	1-8	pre-sync to previous FGU op from that thread
MEMBAR	Memory barrier	variable	
MOVcc	Move integer register if condition is satisfied	1	
MOVr	Move integer register on contents of integer register	1	
MULScc	Multiply step (and modify condition codes)		
MULX	Multiply 64-bit integers	5	
NOP	No operation	1	
NORMALW	Mark other windows as restorable	25	
OR (ORcc)	Inclusive-**or** (and modify condition codes)	1	
ORN (ORNcc)	Inclusive-**or not** (and modify condition codes)	1	
OTHERW	Mark restorable windows as other	6	
PDIST	Distance between eight 8-bit components	6	1 per 2 cycles
POPC	Population count	5	
PREFETCH	Prefetch data	variable	>6

TABLE A-1 UltraSPARC T2 Instruction Latencies *(6 of 8)*

Opcode	Description	Latency	Notes
PREFETCHA	Prefetch data from alternate space	variable	>6
RDASI	Read ASI register	variable	
RDASR	Read ancillary state register	variable	
RDCCR	Read condition codes register	variable	
RDFPRS	Read floating-point registers state register	variable	
RDPC	Read program counter	variable	
RDPR	Read privileged register	variable	
RDTICK	Read TICK register	variable	
RDY	Read Y register	variable	
RESTORE	Restore caller's window	6	
RESTORED	Window has been restored	6	
RETRY	Return from trap and retry		
RETURN	Return	7	
SAVE	Save caller's window	6	
SAVED	Window has been saved	6	
SDIV (SDIVcc)	32-bit signed integer divide (and modify condition codes)	12-41	
SDIVX	64-bit signed integer divide	12-41	
SETHI	Set high 22 bits of low word of integer register	1	
SIAM	Set interval arithmetic mode	6	
SLL	Shift left logical	1	
SLLX	Shift left logical, extended	1	
SMUL (SMULcc)	Signed integer multiply (and modify condition codes)	5	
SRA	Shift right arithmetic	1	
SRAX	Shift right arithmetic, extended	1	
SRL	Shift right logical	1	
SRLX	Shift right logical, extended	1	
STB	Store byte	1	
STBA	Store byte into alternate space		
STBAR	Store barrier	variable	

TABLE A-1 UltraSPARC T2 Instruction Latencies *(7 of 8)*

Opcode	Description	Latency	Notes
STBLOCKF	64-byte block store	16	Assuming store buffer empty when STBLOCKF decodes
STD	Store doubleword	1	
STDA	Store doubleword into alternate space		
STDF	Store double floating-point	1	
STDFA	Store double floating-point into alternate space		
STF	Store floating-point	1	
STFA	Store floating-point into alternate space		
STFSR	Store floating-point state register	1-8	pre-sync to previous FGU op from that thread
STH	Store halfword	1	
STHA	Store halfword into alternate space		
STPARTIALF	Eight 8-bit/4 16-bit/2 32-bit partial stores	1	
STW	Store word	1	
STWA	Store word into alternate space		
STX	Store extended	1	
STXA	Store extended into alternate space	variable	
STXFSR	Store extended floating-point state register		
SUB (SUBcc)	Subtract (and modify condition codes)	1	
SUBC (SUBCcc)	Subtract with carry (and modify condition codes)	1	
SWAP	Swap integer register with memory	20-30	Done in L2 cache
SWAPA	Swap integer register with memory in alternate space	20-30	Done in L2 cache
TADDcc (TADDccTV)	Tagged add and modify condition codes (trap on overflow)	1	If no trap, 6 if trap
TSUBcc (TSUBccTV)	Tagged subtract and modify condition codes (trap on overflow)	1	If no trap, 6 if trap

TABLE A-1 UltraSPARC T2 Instruction Latencies *(8 of 8)*

Opcode	Description	Latency	Notes
Tcc	Trap on integer condition codes (with 8-bit sw_trap_number, if bit 7 is set, trap to hyperprivileged)	1	If no trap, 6 if trap
UDIV (UDIVcc)	Unsigned integer divide (and modify condition codes)	12-41	
UDIVX	64-bit unsigned integer divide	12-41	
UMUL (UMULcc)	Unsigned integer multiply (and modify condition codes)	5	
WRASI	Write ASI register		
WRASR	Write ancillary state register	variable	
WRCCR	Write condition codes register	25	
WRFPRS	Write floating-point registers state register	25	
WRPR	Write privileged register	variable	
WRY	Write Y register	25	
XNOR (XNORcc)	Exclusive-**nor** (and modify condition codes)	1	
XOR (XORcc)	Exclusive-**or** (and modify condition codes)	1	

APPENDIX **B**

IEEE 754 Floating-Point Support

UltraSPARC T2 conforms to the SPARC V9 Appendix B (IEEE Std 754-1985 Requirements for SPARC-V9) recommendation.

Note | UltraSPARC T2 detects tininess before rounding.

B.1 Special Operand Handling

The UltraSPARC T2 FGU follows the UltraSPARC I/UltraSPARC II handling of special operands instead of that used in UltraSPARC T1. While UltraSPARC T1 provides full hardware support for subnormal operands and results, UltraSPARC T2 generates an *fp_exception_other* exception (with FSR.ftt = unfinished_FPop) in some cases. In addition, UltraSPARC T2 implements a nonstandard floating-point mode (enabled when FSR.ns = 1), whereas UltraSPARC T1 does not.

The FGU generates +∞, −∞, +largest number, −smallest 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 for subnormal or gross underflow results: (see *Subnormal Handling* on page 106).

For conversion to integer instructions {F<s | d>TOi, F<s | d>TOx}: UltraSPARC T2 follows *The SPARC Architecture Manual-Version 9* (appendix B.5, pg 246.

For NaN's: UltraSPARC T2 follows *The SPARC Architecture Manual-Version 9* 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 II/UltraSPARC III implementations, all Fpops as defined in V9 will update cexc. All other instructions will leave cexc unchanged.

The remainder of this section gives examples of special cases to be aware of that could generate various exceptions.

B.1.1 Infinity Arithmetic

Let "num" be defined as unsigned in the following tables.

B.1.1.1 One Infinity Operand Arithmetic

- Do not generate exceptions.

TABLE B-1 One-Infinity Operations That Do Not Generate Exceptions

Cases
+∞ plus +num = +∞ +∞ plus -num = +∞ -∞ plus +num = –∞ -∞ plus -num = –∞
+∞ minus +num = +∞ +∞ minus -num = +∞ -∞ minus +num = –∞ -∞ minus –num = –∞
+∞ multiplied by +num = +∞ +∞ multiplied by -num = –∞ –∞ multiplied by +num = –∞ –∞ multiplied by –num = +∞
+∞ divided by +num = +∞ +∞ divided by –num = –∞ –∞ divided by +num = –∞ –∞ divided by –num = +∞

TABLE B-1 One-Infinity Operations That Do Not Generate Exceptions *(Continued)*

Cases
+num divided by +∞ = +0 +num divided by -∞ = –0 –num divided by +∞ = –0 –num divided by –∞ = +0
FsTOd, FdTOs (+∞) = +∞ FsTOd, FdTOs (–∞) = –∞
sqrt(+∞) = +∞
+∞ divided by +0 = +∞ +∞ divided by –0 = –∞ –∞ divided by +0 = –∞ –∞ divided by –0 = +∞
Any arithmetic operation involving infinity as one operand and a QNaN as the other operand: V9, B.2.2, Table 27. (± ∞) OPERATOR (QNaN2) = QNaN2 (QNaN1) OPERATOR (± ∞) = QNaN1
Compares when other operand is not a NaN treat infinity just like a regular number: +∞ = +∞, +∞ > anything else; –∞ = –∞, –∞ < anything else. Effects following instructions: V9 fp compares (rs1 and/or rs2 could be ± ∞): * FCMPE * FCMP
Compares when other operand is a QNaN, SPARC V9 A.13, B.2.1; fcc value = unordered = 11_2 FCMP(s,d) (± ∞) with (QNaN2) – no invalid exception FCMP(s,d) (QNaN1) with (±∞) – no invalid exception

- Could generate exceptions

TABLE B-2 One Infinity Operations That Could Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
V9, Appendix B.5[1]	IEEE_754 7.1	
F<s | d>TOi (+∞) = invalid	IEEE_754 invalid	$2^{31}-1$
F<s | d>TOx (+∞) = invalid	IEEE_754 invalid	$2^{63}-1$
F<s | d>TOi (–∞) = invalid	IEEE_754 invalid	-2^{31}
F<s | d>TOx (–∞) = invalid	IEEE_754 invalid	-2^{63}
V9, B.2.2	IEEE_754 7.1	(No NaN operand result)
sqrt(–∞) = invalid	IEEE_754 invalid	QNaN
+∞ multiplied by +0 = invalid	IEEE_754 invalid	QNaN
+∞ multiplied by –0 = invalid	IEEE_754 invalid	QNaN
–∞ multiplied by +0 = invalid	IEEE_754 invalid	QNaN
–∞ multiplied by –0 = invalid	IEEE_754 invalid	QNaN
V9, B.2.2, Table 27[2] Any arithmetic operation involving infinity as one operand and SNaN as the other operand except copying/moving data	IEEE_754 7.1	(One operand, a SNaN)
(± ∞) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (± ∞)	IEEE_754 invalid	QSNaN1
V9, A.13, B.2.1[2] Any compare operation involving infinity as one operand and a SNaN as the other operand:	IEEE_754 7.1	
FCMP<s | d> (± ∞) with (SNaN2)	IEEE_754 invalid	fcc value = unordered = 11_2
FCMP<s | d> (SNaN1) with (± ∞)	IEEE_754 invalid	fcc value = unordered = 11_2
FCMPE<s | d> (± ∞) with (SNaN2)	IEEE_754 invalid	fcc value = unordered = 11_2
FCMPE<s | d> (SNaN1) with (± ∞)	IEEE_754 invalid	fcc value = unordered = 11_2
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> (± ∞) with (QNaN2)	IEEE_754 invalid	fcc value = unordered = $2'b11_2$
FCMPE<s | d> (QNaN1) with (± ∞)	IEEE_754 invalid	fcc value = unordered = $2'b11_2$

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 resulting number that cannot fit in 32-bit[64-bit] integer format: (large positive argument $\geq 2^{31}[2^{63}]$ or large negative argument $\leq -(2^{31}+1)[-(2^{63}+1)]$

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).

B.1.1.2 Two Infinity Operand Arithmetic

- Do not generate exceptions

TABLE B-3 Two Infinity Operations That Do Not Generate Exceptions

Cases
+∞ plus +∞ = +∞ –∞ plus –∞ = –∞
+∞ minus –∞ = +∞ –∞ minus +∞ = –∞
+∞ multiplied by +∞ = +∞ +∞ multiplied by –∞ = –∞ –∞ multiplied by +∞ = –∞ –∞ multiplied by –∞ = +∞
Compares treat infinity just like a regular number: +∞ = +∞, +∞ > anything else; –∞ = –∞, –∞ < anything else. Affects following instructions: V9 fp compares (rs1 and/or rs2 could be ± ∞): * FCMPE * FCMP

■ Could generate exceptions

TABLE B-4 Two Infinity Operations That Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
V9, B.2.2	IEEE_754 7.1	(No NaN operand result)
+∞ plus –∞ = invalid	IEEE_754 invalid	QNaN
–∞ plus +∞ = invalid	IEEE_754 invalid	QNaN
+∞ minus +∞ = invalid	IEEE_754 invalid	QNaN
–∞ minus –∞ = invalid	IEEE_754 invalid	QNaN
V9, B.2.2	IEEE_754 7.1	(No NaN operand result)
+∞ divided by +∞ = invalid	IEEE_754 invalid	QNaN
+∞ divided by –∞ = invalid	IEEE_754 invalid	QNaN
–∞ divided by +∞ = invalid	IEEE_754 invalid	QNaN
–∞ divided by –∞ = invalid	IEEE_754 invalid	QNaN

B.1.2 Zero Arithmetic

TABLE B-5 Zero Arithmetic Operations that generate exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
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
V9. 5.1.7.10.4	IEEE_754 7.2	
+num divided by +0 = divide by zero	IEEE_754 div_by_zero	+∞
+num divided by –0 = divide by zero	IEEE_754 div_by_zero	–∞
–num divided by +0 = divide by zero	IEEE_754 div_by_zero	–∞
–num divided by –0 = divide by zero	IEEE_754 div_by_zero	+∞
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 B-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.
sqrt (–0) = –0

B.1.3 NaN Arithmetic

- Do not generate exceptions

TABLE B-7 NaN Arithmetic Operations That Do Not Generate Exceptions

Cases
V9, B.2.1: Fp convert to wider NaN transformation FsTOd (QNaN2) = QNaN2 widened FsTOd (7FD1 0000_{16}) = 7FFA 2000 0000 0000_{16} FsTOd (FFD1 0000_{16}) = FFFA 2000 0000 0000_{16}
V9, B.2.1: Fp convert to narrower NaN transformation FdTOs (QNaN2) = QNaN2 narrowed FdTOs (7FFA 2000 0000 0000_{16}) = 7FD 1000_{16} FdTOs (FFFA 2000 0000 0000_{16}) = FFD 1000_{16}
V9, B.2.2 Table 27 Any noncompare arithmetic operations. Result takes sign of QNaN pass through operand. [Note this rule is applicable to sqrt(QNaN2) = QNaN2 as well]. (± num) OPERATOR (QNaN2) = QNaN2 (QNaN1) OPERATOR (± num) = QNaN1 (QNaN1) OPERATOR (QNaN2) = QNaN2

- Could Generate Exceptions

TABLE B-8 NaN Arithmetic Operations That Could Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
V9, B.2.1: Fp convert to wider NaN transformation FsTOd (SNaN2) = QSNaN2 widened FsTOd (7F91 0000_{16}) = 7FFA 2000 0000 0000_{16} FsTOd (FF91 0000_{16}) = FFFA 2000 0000 0000_{16}	IEEE_754 7.1 IEEE_754 invalid	 QSNaN2 widened
V9, B.2.1: Fp convert to narrower NaN transformation FdTOs (SNaN2) = QSNaN2 narrowed FdTOs (7FF2 2000 0000 0000_{16}) = 7FD 1000_{16} FdTOs (FFF2 2000 0000 0000_{16}) = FFD 1000_{16}	IEEE_754 7.1 IEEE_754 invalid	 QSNaN2 narrowed

TABLE B-8 NaN Arithmetic Operations That Could Generate Exceptions *(Continued)*

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
V9, B.2.2 Table 27 Any noncompare arithmetic operations except copying/moving (FMOV, FABS, FNEG) [Note this rule applies to sqrt(SNaN2) = QNaN2 and invalid exception as well]	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
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}

B.1.4 Special Inexact Exceptions

UltraSPARC T2 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 B-9 Fp ↔ Int Conversions With Inexact Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
V9, A.14: Fp convert to 32-bit 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
V9, A.14: Fp convert to 64-bit 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
V9, A.15: Convert integer to fp format when 32-bit integer source operand magnitude is not exactly representable in single precision (23-bit 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	 An SP number
V9, A.15: Convert integer to fp format when 64-bit integer source operand magnitude is not exactly representable in single precision (23-bit 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	 An SP number
V9, A.15: Convert integer to fp format when 64-bit integer source operand magnitude is not exactly representable in double precision (52-bit 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

B.2 Subnormal Handling

The UltraSPARC T2 FGU follows the UltraSPARC I/UltraSPARC II subnormal handling instead of that used in UltraSPARC T1. While UltraSPARC T1 provides full hardware support for subnormal operands and results, UltraSPARC T2 generates an unfinished_FPop trap type in some cases. In addition, UltraSPARC T2 implements a nonstandard floating-point mode, whereas UltraSPARC T1 does not.

UltraSPARC T2 provides limited subnormal support in hardware when in standard mode (FSR.ns = 0) or interval arithmetic mode (GSR.im = 1) [Note that when GSR.im = 1, regardless of FSR.ns, UltraSPARC T2 operates in standard mode.]:

- UltraSPARC T2 supports full subnormal operand handling for single and double precision fp compares;
- UltraSPARC T2 supports gross underflow results for fp-to-fp conversions from higher to lower precision (FdTOs);
- UltraSPARC T2 supports gross underflow results in hardware for FMUL(s,d) and FDIV(s,d) which gives 90% of the optimal underflow performance at a fraction of the cost to completely support subnormal operands and results;
- For those instructions without any subnormal support, an unfinished trap is taken.

UltraSPARC T2 supports the following in nonstandard mode ((FSR.ns = 1) and (GSR.im = 0)):

- Subnormal operands and results are flushed to zero with the same sign, and execution is allowed to proceed without incurring the performance cost of an unfinished trap.

TABLE B-11 and TABLE B-12 show how each instruction type is explicitly handled.

Handling of the FMUL<s | d>, FDIV<s | d>, FdTOs instructions requires a few additional definitions:

- Let Signr = sign of result, RP = round to +infinity, RM = round to –infinity. Define RND as round mode bits. In standard mode, these can have two different sources:

  When in typical standard mode ((FSR.ns = 0) and (GSR.im = 0)), RND = FSR.rd

  When in interval arithmetic mode (GSR.im = 1), RND = GSR.irnd
- Let E(rs1) = biased exponent of rs1 operand, and E(rs2) = biased exponent of rs2 operand
- Let Er = unnormalized and unrounded biased exponent result

  For FMUL<s | d>: Er = E(rs1) + E(rs2) – EBIAS(P)

  For FDIV<s | d>: Er = E(rs1) – E(rs2) + EBIAS(P) – 1

  For FdTOs: Er = E(rs2) – EBIAS(P_rs2) + EBIAS(P_rd), where P_rs2 is the larger precision of the source and P_rd is the smaller precision of the destination
- Let Ef = final normalized and rounded biased exponent result

■ Define constants dependent on precision type (see TABLE B-10)

TABLE B-10 Subnormal Handling Constants Per Destination Precision Type

Precision (P)	Number of exponent field bits	Exponent Bias (EBIAS)	Exponent Max (EMAX)	Exponent Gross Underflow (EGUF)
Single	8	127	255	–25
Double	11	1023	2047	–54

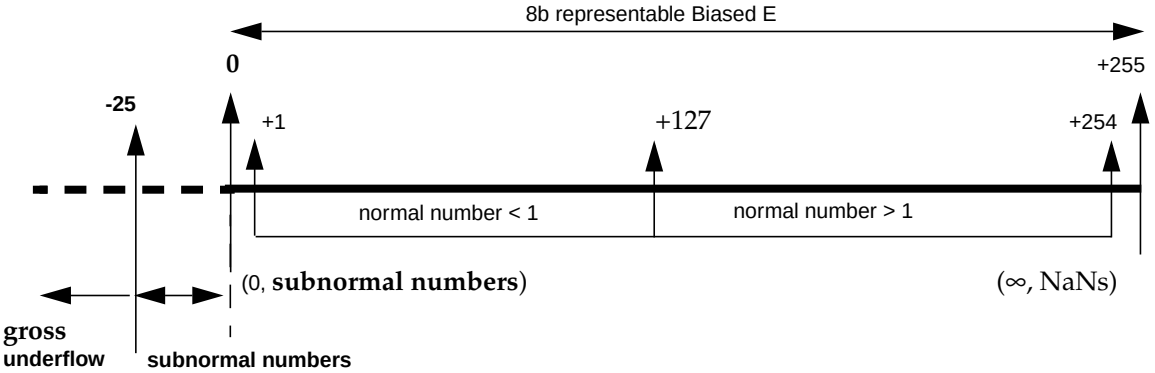


FIGURE B-1 Single Precision Unbiased vs. Biased Subnormals and Gross Underflow

■ Note that even though 0 ≤ [E(rs1) or E(rs2)] ≤ 255 for each single precision biased operand exponent, the *computed* Er can be 0 ≤ Er ≤ 255 or can even be negative. For example, for a FMULS instruction:

If E(rs1) = E(rs2) = +127, then Er = 127 + 127 –127 = +127

If E(rs1) = E(rs2) = 0, then Er = 0 + 0 - 127 = –127

■ Following the sections 5.1.7.6, 5.1.7.8, 5.1.7.9, and figures in 5.1.7.10 of *The SPARC Architecture Manual-Version 9*, Overflow Result is defined as follows:

> 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 previous UltraSPARC I/ UltraSPARC II implementations 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 the sections 5.1.7.6, 5.1.7.8, 5.1.7.9 and figures in 5.1.7.10 of *The SPARC Architecture Manual-Version 9*, Gross Underflow Zero Result is defined as follows:

  Result = 0 (with correct sign)

  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/ UltraSPARC II implementations 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.

- Subnormal handling is overridden for the following cases:
  - Result is a QNaN or SNaN — by *The SPARC Architecture Manual-Version 9* Appendix B.2.2 (Table 27).

    Define "OP_NaN" as instruction uses a SNaN or QNaN operand.

    Examples:

    subnormal + SNaN = QNaN with invalid exception (No unfinished trap in standard mode and no FSR.nx in nonstandard mode)

    subnormal + QNaN = QNaN, no exception (No unfinished trap in standard mode and no FSR.nx in nonstandard mode)

  - Result already generates an exception.

    Define "OP_lt_0" as instruction uses an operand less than zero.

    Examples:

    sqrt(number less than zero) = invalid

  - Result is infinity.

    Define "OP_inf" as instruction uses infinity operand.

    Examples:

    subnormal $+\infty = \infty$ (No unfinished trap in standard mode and no FSR.nx in nonstandard mode)

    subnormal $\times \infty = \infty$ in standard mode; subnormal $\times \infty$ = QNaN with invalid exception in nonstandard mode since subnormal is flushed to zero.

  - Result is zero.

    Define "OP_0" as instruction uses a zero operand

Example:

subnormal × 0 = 0 (No unfinished trap in standard mode, and no FSR.nx in nonstandard mode)

B.2.1 One or Both Subnormal Operands

TABLE B-11 One or Both Subnormal Operands Handling *(1 of 3)*

Instructions	(FSR.ns = 0) *or* (GSR.im = 1) Standard Mode	(FSR.ns = 1) *and* (GSR.im = 0) Nonstandard mode

Single/Double Precision add, subtract [FADD<s|d>, FSUB<s|d>]	**if** (**not** (OP_NaN **or** OP_inf)) {
	generate unfinished trap
	}
		if (**not**(OP_NaN **or** OP_inf)) {
		executes w/subnormal operand flushed to 0 with the same sign
		FSR.nx ← 1
		}

Single/Double Precision FPCOMPARE [FCMP<s|d>, FCMPE<s|d>]	**if** (**not** OP_NaN) {
	execute the compare using the subnormal operand(s)
	}
		if (**not** OP_NaN) {
		executes the compare using the subnormal operand(s)[3]
		}

Single/Double Precision multiply [FMUL<s|d>]	**if** (**not** (OP_NaN **or** OP_inf **or** OP_0)) {
	If ((Er > EGUF(P))
	or
	(Er ≤ EGUF(P) **and** Signr=0 **and** RND=RP)
	or
	(Er ≤ EGUF(P) **and** Signr=1 **and** RND=RM))
	{generate unfinished trap}
	else {
	generate gross underflow zero result[1] }
	}
		if (**not**(OP_NaN **or** OP_0) {
		if (**not**(OP_inf)) {
		executes w/subnormal operand flushed to 0 with the same sign
		FSR.nx ← 1
		} **else** { // 1 op is subnormal, other is ∞
		executes w/subnormal operand flushed to 0 with the same sign
		FSR.nv ← 1 & return QNaN
		}
		}

TABLE B-11 One or Both Subnormal Operands Handling *(Continued)* *(2 of 3)*

Instructions	(FSR.ns = 0) *or* (GSR.im = 1) Standard Mode	(FSR.ns = 1) *and* (GSR.im = 0) Nonstandard mode

Single/double precision divide [FDIV<s|d>]	**if** (**not** (OP_NaN **or** OP_inf **or** OP_0)) {
	  if (**not** (Ef > EMAX(P))) {
	  // not overflow
	  if ((Er > EGUF(P))
	  or
	  (Er ≤ EGUF(P) **and** (Signr=0)
	  and (RND=RP))
	  or
	  (Er ≤ EGUF(P) **and** (Signr=1)
	  and (RND=RM)))
	  {
	  generate unfinished trap
	  } **else** {
	  generate gross-underflow zero result[1]
	  }
	  } **else** {
	  generate overflow result[2]
	  }
	}
		if (**not**(OP_NaN) {
		  if (**not**(OP_inf **or** OP_0) {
		  executes w/subnormal operand flushed to 0 with the same sign
		  if (rs1 and rs2 are flushed to zero) {
		  FSR.nv ← 1 // 0 ÷ 0 is invalid
		  } **else if** (rs2 divisor is flushed to zero) {
		  FSR.dz ← 1
		  } **else** {
		  FSR.nx ← 1
		  }
		  } **else if** (OP_inf) { // 1 op is subnormal,
		  // other = ∞
		  executes w/subnormal operand flushed to 0 with the same sign
		  // 0 ÷ ∞ is 0, and ∞ ÷ 0 is ∞
		  No exceptions are set.
		  Even if divisor is flushed to zero: no need to set FSR.dz
		  } **else if** (OP_0) { // 1 op subnorm, other = 0
		  executes w/subnormal operand flushed to 0 with the same sign
		  // 0 ÷ 0 is invalid exception
		  FSR.nv ← 1 // overrides FSR.dz
		  }
		}

Single to double precision multiply [FSMULD]	**if** (**not** (OP_NaN **or** OP_inf **or** OP_0)) {
	  generate unfinished trap
	}
		if (**not**(OP_NaN **or** OP_0) {
		  if (**not**(OP_inf)) {
		  executes w/subnormal operand flushed to 0 with the same sign
		  Set FSR.nx
		  } **else** { // 1 op is subnormal, other is ∞
		  executes w/subnormal operand flushed to 0 with the same sign
		  Set FSR.nv & return QNaN
		  }
		}

TABLE B-11 One or Both Subnormal Operands Handling *(Continued) (3 of 3)*

Instructions	(FSR.ns = 0) *or* (GSR.im = 1) Standard Mode	(FSR.ns = 1) *and* (GSR.im = 0) Nonstandard mode

Single/Double Precision square root [FSQRT(s,d)] (*note: single operand instruction)	**if** (**not** (OP_NaN **or** OP_inf)) {
	  if (OP_lt_0) {
	  FSR.nv ← 1 & return QNaN
	  } **else** {
	  generate unfinished trap
	  }
	}
		if (**not** (OP_NaN **or** OP_inf)) {
		  executes w/subnormal operand flushed to 0 with the same sign
		  if (OP_lt_0) { //that is, subnormal
		  FSR.nx ← 1; return –0
		  } **else if** (OP_eq_0) { //that is, 0
		  No exception; return 0 with same sign
		  } **else** {
		  FSR.nx ← 1
		  }
		}

Fp to Int and Fp Single to Double conversion [FsTOx, FdTOx, FsTOi, FdTOi, FsTOd] (*note single operand instructions)	**if** (**not** (OP_NaN **or** OP_inf)) {
	  generate unfinished trap
	}
		if (**not** (OP_NaN **or** OP_inf)) {
		  executes w/subnormal operand flushed to 0 with the same sign
		  FSR.nx ← 1
		}

Fp Double to Single conversion [FdTOs] (*note single operand instruction)	**if** (**not** (OP_NaN **or** OP_inf)) {
	  if ((Signr = 0 **and** RND = RP)
	  or
	  (Signr = 1 **and** RND = RM)) {
	  generate unfinished trap
	  } **else** {
	  generate gross underflow zero result[1]
	  }
	}
		if (**not** (OP_NaN **or** OP_inf)) {
		  executes w/subnormal operand flushed to 0 with the same sign
		  FSR.nx ← 1
		}

1. See gross underflow zero result definition on page 109.
2. See overflow definition on page 109.
3. This errata (not flushing subnormal operands to zero for only single and double precision fpcompares) ensures UltraSPARC T2 is backward compatible with UltraSPARC I/UltraSPARC II and UltraSPARC III.

B.2.2 Normal Operand(s) Giving Subnormal Result

TABLE B-12 Subnormal Result Handling for Two Normal Operands

Instructions	(FSR.ns = 0) *or* (GSR.im = 1)	(FSR.ns = 1) *and* (GSR.im = 0)
Single/Double Precision add, subtract [FADD<s|d>), FSUB<s|d>]	Generate unfinished trap	Generate gross underflow zero result[1]

Single/Double Precision multiply [FMUL<s|d>] and divide [FDIV<s|d>], Fp double-to-single conversion [FdTOs] (*note single operand instruction)	**if** (**not** (Ef >= 1)) { // that is, not subnormal intermediate result that rounded to normalized result
	if ((1 > Er > EGUF(P))
	or
	(Er ≤ EGUF(P) **and** Signr = 0 **and** RND = RP)
	or
	(Er ≤ EGUF(P) **and** Signr = 1 **and** RND = RM)) {
	generate unfinished trap
	}
	else {
	generate gross underflow zero result[1]
	}
	} **else** {
	generate normalized result
	}
		Generate gross underflow zero result[1]

1. See gross underflow zero result definition on page 109.

- For those instructions found in TABLE B-11 but not in TABLE B-12, TABLE B-11 is sufficient for their subnormal handling; so the additional rules in TABLE B-12 need not be applied.
- Multiplies that include a conversion from a smaller to larger precision (FSMULD) are not included in TABLE B-12 along with FMUL<s|d> because the larger precision result's exponent range is sufficient to represent a number that would have underflowed in the smaller precision's exponent range.

APPENDIX C

Differences From UltraSPARC T1

C.1 General Architectural and Microarchitectural Differences

UltraSPARC T2 follows the CMT philosophy of UltraSPARC T1, but adds more execution and cryptography capability to each physical core, as well as significant system-on-a-chip components and an enhanced L2 cache. The following lists the microarchitectural differences:

- Physical core consists of two integer execution pipelines and a single floating-point pipeline. UltraSPARC T1 had a single integer execution pipeline and all cores shared a single floating-point pipeline.
- Each physical core in UltraSPARC T2 supports eight strands, which all share the floating-point pipeline. The eight strands are partitioned into two groups of four strands, each of which shares an integer pipeline. UltraSPARC T1 shared the single integer pipeline among four strands.
- Pipeline in UltraSPARC T2 is eight stages, two stages longer than UltraSPARC T1.
- Instruction cache is 8-way associative, compared to 4-way in UltraSPARC T1.
- The L2 cache is 4 Mbyte, 8-banked and 16-way associative, compared to 3 Mbyte, 4-banked and 12-way associative in UltraSPARC T1.

C.2 ISA Differences

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

- UltraSPARC T2 fully supports all VIS 2.0 instructions. UltraSPARC T1 supports a subset of VIS 1.0 plus SIAM (the remainder of VIS 1.0 and 2.0 instructions trap to software for emulation, on UltraSPARC T1).

- UltraSPARC T2 supports ALLCLEAN, INVALW, NORMALW, OTHERW, POPC, and FSQRT<s|d> in hardware.
- UltraSPARC T2 has a floating-point unit that generates *fp_unfinished_operation* for most denorm cases and supports a nonstandard mode that flushes denorms to zero, as described in Appendix B. UltraSPARC T1 handles denorms in hardware, never generates an *FP_unfinished_operation* trap and does not support a nonstandard mode.
- UltraSPARC T2 generates an *illegal_instruction* trap on any quad-precision FP instruction, while UltraSPARC T1 generates an *fp_exception_other* trap on numeric and move-FP-quad instructions. See Table 5-2 on page 24.
- UltraSPARC T2 generates a *privileged_action* exception upon attempted access to hyperprivileged ASIs by privileged software whereas, in such cases, UltraSPARC T1 takes a *data_access_exception* exception.
- UltraSPARC T2 supports PSTATE.tct; UltraSPARC T1 did not.
- UltraSPARC T2 implements SAVE similar to all previous UltraSPARC processors. UltraSPARC T1 implements a SAVE that updates the locals in the new window to be the same as the locals in the old window, and swaps the *in*s (*out*s) of the old window with the *out*s (*in*s) of the new widow.
- PSTATE.am masking details differ between UltraSPARC T1 and UltraSPARC T2, as described in Section 11.1.7, *Address Masking (Impdep #125)*, on page 64.

C.3 MMU Differences

The UltraSPARC T2 MMU is described in Chapter 12. The UltraSPARC T2 and UltraSPARC T1 MMUs differ as follows:

- UltraSPARC T2 supports a pair of primary context registers and a pair of secondary context registers. UltraSPARC T1 supports a single primary context and single secondary context register.
- UltraSPARC T2 supports only the sun4v TTE format. UltraSPARC T1 supports both the sun4v and the sun4u TTE formats.
- UltraSPARC T2 is compatible with UltraSPARC Architecture 2007 with regard to multiple flavors of data access exception (*DAE_**) and instruction access exception (*IAE_**). UltraSPARC T1 uses the UltraSPARC single flavor of *data_access_exception* and *instruction_access_exception*.
- UltraSPARC T1 and UltraSPARC T2 support the same four page sizes (8 Kbyte, 64 Kbyte, 4 Mbyte, 256 Mbyte). UltraSPARC T2 supports an *unsupported_page_size* trap when an illegal page size is programmed into TSB registers or attempted to be loaded into the TLB. UltraSPARC T1 forces an illegal page size being

programmed into TSB registers to be 256 Mbytes and generates a *data_access_exception* trap when a page with an illegal size is loaded into the TLB.

C.4 Performance Instrumentation Differences

Both UltraSPARC T1 and UltraSPARC T2 provide access to hardware performance counters through the PIC and PCR registers. However, the events captured by the hardware differ significantly between UltraSPARC T1 and UltraSPARC T2, with UltraSPARC T2 capturing a much larger set of events, as described in Chapter 10.

APPENDIX **D**

Caches and Cache Coherency

D.1 Cache and Memory Interactions

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:

- 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.

D.2 Cache Flushing

Data in the level-1 (read-only or writethrough) caches can be flushed by invalidating the entry in the cache (in a way that also leaves the L2 directory in a consistent state). 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. This is done with the FLUSH instruction.
- **D-cache:** Flush is needed when a physical page is changed from (physically) cacheable to (physically) noncacheable. This is done with a displacement flush (*Displacement Flushing*, below).

- **L2 cache:** Flush is needed for stable storage. Examples of stable storage include battery-backed memory and transaction logs. The recommended way to perform this is by using a service call to hyperprivileged software. Alternatively, this can be done by a displacement flush (see the next section). Flushing the L2 cache flushes the corresponding blocks from the I- and D-caches, because UltraSPARC T2 maintains inclusion between the L2 and L1 caches.

D.2.1 Displacement Flushing

Cache flushing of the L2 cache or the D-cache 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.)

D.2.2 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.

In UltraSPARC T2, all memory accesses are cached in the L2 cache (as long as the L2 cache is enabled). The `cp` bit in the `TTE` corresponding to the access controls whether the memory access will be cached in the primary caches (if `cp` = 1, the access is cached in the primary caches; if `cp` = 0 the access is not cached in the primary caches). Atomic operations are always performed at the L2 cache.

D.2.3 Coherence Domains

Two types of memory operations are supported in UltraSPARC T2: cacheable and noncacheable accesses, as indicated by the page translation. Cacheable accesses are inside the coherence domain; noncacheable accesses are outside the coherence domain.

SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses. UltraSPARC T2 maintains TSO ordering, regardless of the cacheability of the accesses, relative to other access by processors. (Ordering of processor accesses relative to DMA accesses roughly follows PCI ordering rules.)

See the *The SPARC Architecture Manual-Version 9* for more information about the SPARC V9 memory models.

On UltraSPARC T2, a MEMBAR #`Lookaside` is effectively a NOP and is not needed for forcing order of stores vs. loads to noncacheable addresses.

D.2.3.1 Cacheable Accesses

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

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

D.2.3.2 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 T2 with the following properties:

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

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

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

D.2.3.3 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 D-1 illustrates the issues involved in mixing cacheable and noncacheable accesses.

CODE EXAMPLE D-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 for RMO cacheable accesses.

Due to load and store buffers implemented in UltraSPARC T2, CODE EXAMPLE D-1 may not work for RMO accesses without the MEMBARs shown in the program segment.

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

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

D.2.4 Memory Synchronization: MEMBAR and FLUSH

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

D.2.4.1 MEMBAR #`LoadLoad`

All loads on UltraSPARC T2 switch a strand out until the load completes. Thus, MEMBAR #`LoadLoad` is treated as a NOP on UltraSPARC T2.

D.2.4.2 MEMBAR #`StoreLoad`

MEMBAR #`StoreLoad` forces all loads after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility. MEMBAR #`StoreLoad` behaves the same as MEMBAR #`Sync` on UltraSPARC T2.

D.2.4.3 MEMBAR #`LoadStore`

All loads on UltraSPARC T2 switch a strand out until the load completes. Thus, MEMBAR #`LoadStore` is treated as a NOP on UltraSPARC T2

D.2.4.4 MEMBAR #`StoreStore` and STBAR

Stores on UltraSPARC T2 maintain order in the store buffer. Thus Membar #`StoreStore` is treated as a NOP on UltraSPARC T2.

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

UltraSPARC T2 block stores and block-init stores are RMO. If a program needs to maintain order between RMO stores to different L2 cache lines, it should use a MEMBAR #Sync.

D.2.4.5 MEMBAR #Lookaside

Loads and stores to noncacheable addresses are "self-synchronizing" on UltraSPARC T2. Thus MEMBAR #Lookaside is treated as a NOP on UltraSPARC T2.

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

D.2.4.6 MEMBAR #MemIssue

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 (as viewed by other strands) before the cacheable stores after the MEMBAR.

Since loads are already self-synchronizing, Membar #MemIssue just needs to drain the store buffer (and receive all the store ACKs) before allowing memory operations to issue again. This is the same operation as UltraSPARC T2's Membar #Sync.

D.2.4.7 MEMBAR #Sync (Issue Barrier)

Membar #Sync 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.

D.2.4.8 Self-Modifying Code (FLUSH)

The SPARC V9 instruction set architecture does not guarantee consistency between code and data spaces. A problem arises when code space is dynamically modified by a program writing to memory locations containing instructions. Dynamic

optimizers, 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 T2, FLUSH behaves like a store instruction for the purpose of memory ordering. In addition, all instruction fetch (or prefetch) buffers are invalidated. The issue of the FLUSH instruction is delayed until previous (cacheable) stores are completed. Instruction fetch (or prefetch) resumes at the instruction immediately after the FLUSH.

D.2.5 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 unprivileged mode (PSTATE.priv = 0) causes a *privileged_action* trap. An atomic access with a noncacheable address causes a *DAE_nc_page* trap. An atomic access with an unsupported ASI causes a *DAE_invalid_ASI* trap. TABLE D-1 lists the ASIs that support atomic accesses.

TABLE D-1 ASIs That Support SWAP, LDSTUB, and CAS

ASI Name
ASI_NUCLEUS{_LITTLE}
ASI_AS_IF_USER_PRIMARY{_LITTLE}
ASI_AS_IF_USER_SECONDARY{_LITTLE}
ASI_PRIMARY{_LITTLE}
ASI_SECONDARY{_LITTLE}
ASI_REAL{_LITTLE}

Notes

Atomic accesses with nonfaulting ASIs are not allowed, because these ASIs have the load-only attribute.

For all atomics, allocation is done to the L2 cache only and will invalidate the L1s.

D.2.5.1 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.

D.2.5.2 LDSTUB Instruction

LDSTUB behaves like SWAP, except that it loads a byte from memory into an integer register and atomically writes all 1's (FF_{16}) into the addressed byte.

D.2.5.3 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.

D.2.6 Nonfaulting Load

A nonfaulting load behaves like a normal load, except that

- It does not allow side-effect access. An access with the `e` bit set causes a *DAE_so_page* trap.
- It can be applied to a page with the `nfo` bit set; other types of accesses will cause a *DAE_NFO_page* trap.

Nonfaulting loads are issued with `ASI_PRIMARY_NO_FAULT{_LITTLE}` or `ASI_SECONDARY_NO_FAULT{_LITTLE}`. A store with a `NO_FAULT` ASI causes a *DAE_invalid_ASI* trap.

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

Typically, optimizers use nonfaulting loads to move loads before conditional control structures that guard their use. This technique potentially increases the distance between a load of data and the first use of that data, to hide latency; it allows for more flexibility in code scheduling. It also allows for improved performance in certain algorithms by removing address checking from the critical code path.

For example, when following a linked list, nonfaulting loads allow the null pointer to be accessed safely in a read-ahead fashion if the operating system can ensure that the page at virtual address 0_{16} is accessed with no penalty. The `nfo` (nonfault access only) bit in the MMU marks pages that are mapped for safe access by nonfaulting loads but can still cause a trap by other, normal accesses. This allows programmers to trap on wild pointer references (many programmers count on an exception being generated when accessing address 0_{16} to debug code) while benefitting from the acceleration of nonfaulting access in debugged library routines.

APPENDIX **E**

Glossary

This chapter defines concepts and terminology unique to the UltraSPARC T2 implementation. Definitions of terms common to all UltraSPARC Architecture implementations may be found in the Definitions chapter of *UltraSPARC Architecture 2007*.

ALU Arithmetic Logical Unit

architectural state Software-visible registers and memory (including caches).

ARF Architectural register file.

blocking ASI An ASI access that accesses its ASI register or array location once all older instructions in that strand have retired, no instructions in the other strand can issue, and the store queue, TSW, and LMB are all empty.

branch outcome A reference as 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.

commit An instruction commits when it modifies architectural state.

complex instruction A complex instruction is 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 Latency* on page 88 for a complete list of all complex instructions and their helper sequences.

consistency *See* **coherence**.

CPU Central Processing Unit. A synonym for **virtual processor**.

CSR Control Status register.

FP Floating point.

L2C (or L2$) 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.

nonblocking ASI A nonblocking ASI access 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.

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.

one hot An *n*-bit binary signal is one hot if and only if $n - 1$ of the bits are each zero and a single bit is a 1.

quadlet

SIAM Set interval arithmetic mode instruction.

younger instruction *See* **older instruction**.

writeback The process of writing a dirty cache line back to memory before it is refilled.

APPENDIX **F**

Bibliography

[contents of this appendix are TBD]

Index

M

N

O

P

V

W