UltraSPARC T1™ Supplement

to the *UltraSPARC Architecture 2005*

Draft D2.0, 17 Mar 2006

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: 819-3404-04
Revision: Draft D2.0, 17 Mar 2006

Copyright 2002-2006 Sun Microsystems, Inc., 4150 Network Circle • Santa Clara, CA 950540 USA. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. For Netscape Communicator™, the following notice applies: Copyright 1995 Netscape Communications Corporation. All rights reserved.

Sun, Sun Microsystems, the Sun logo, Solaris, and VIS are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2002–2006 Sun Microsystems, Inc., 4150 Network Circle • Santa Clara, CA 950540 Etats-Unis. Tous droits réservés.

Des parties de ce document est protégé par un copyright© 1994 SPARC International, Inc.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Solaris, et VIS sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

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

Contents

Preface

Welcome to the UltraSPARC T1 Processor Supplement, D2.0. This document contains information about the processor-specific aspects of the architecture and programming of the UltraSPARC T1 processor, one of Sun Microsystems' family processors compliant with UltraSPARC Architecture™. It is intended to supplement the *UltraSPARC Architecture 2005* with processor-specific information.

Target Audience

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

Fonts and Notational Conventions

Fonts are used as follows:

- *Italic* font is used for emphasis, book titles, and the first instance of a word that is defined.
- *Italic* font is also used for terms where substitution is expected, for example, "`fcc`*n*", "virtual processor *n*", or "*reg_plus_imm*".
- *Italic sans serif* font is used for exception and trap names. For example, "The *privileged_action* exception...."
- lowercase helvetica font is used for register field names (named bits) and instruction field names, for example: "The `rs1` field contains...."

- UPPERCASE HELVETICA font is used for register names; for example, FSR.
- TYPEWRITER (Courier) font is used for literal values, such as code (assembly language, C language, ASI names) and for state names. For example: %f0, ASI_PRIMARY, execute_state.
- When a register field is shown along with its containing register name, they are separated by a period ('.'), for example, FSR.cexc.
- UPPERCASE words are acronyms or instruction names. Some common acronyms appear in the glossary. **Note:** Names of some instructions contain both upper- and lower-case letters.
- An underscore character joins words in register, register field, exception, and trap names. **Note:** Such words may be split across lines at the underbar without an intervening hyphen. For example: "This is true whenever the integer_condition_ code field...."

The following notational conventions are used:

- The left arrow symbol (←) is the assignment operator. For example, "PC ← PC + 1" means that the Program Counter (PC) is incremented by 1.
- Square brackets ([ ]) are used in two different ways, distinguishable by the context in which they are used:
  - Square brackets indicate indexing into an array. For example, TT[TL] means the element of the Trap Type (TT) array, as indexed by the contents of the Trap Level (TL) register.
  - Square brackets are also used to indicate optional additions/extensions to symbol names. For example, "ST[D,Q]F" expands to all three of "STF", "STDF", and "STQF". Similarly, ASI_PRIMARY[_LITTLE] indicates two related address space identifiers, ASI_PRIMARY and ASI_PRIMARY_LITTLE. (Contrast with the use of angle brackets, below)
- Angle brackets (< >) indicate mandatory additions/extensions to symbol names. For example, "ST<D|Q>F" expands to mean "STDF" and "STQF". (Contrast with the second use of square brackets, above)
- Curly braces ({ }) indicate a bit field within a register or instruction. For example, CCR{4} refers to bit 4 in the Condition Code Register.
- A consecutive set of values is indicated by specifying the upper and lower limit of the set separated by a colon (:), for example, CCR{3:0} refers to the set of four least significant bits of register CCR. (Contrast with the use of double periods, below)
- A double period (..) indicates any *single* intermediate value between two given end values is possible. For example, NAME[2..0] indicates four forms of NAME exist: NAME, NAME2, NAME1, and NAME0; whereas NAME<2..0> indicates that three forms exist: NAME2, NAME1, and NAME0. (Contrast with the use of the colon, above)

- A vertical bar (|) separates mutually exclusive alternatives inside square brackets ([ ]), angle brackets (< >), or curly braces ({ }). For example, “NAME[A|B]” expands to “NAME, NAMEA, NAMEB” and “NAME<A|B>” expands to "NAMEA, NAMEB".
- The asterisk (*) is used as a wild card, encompassing the full set of valid values. For example, FCMP* refers to FCMP with all valid suffixes (in this case, FCMP<s|d|q> and FCMPE<s|d|q>). An asterisk is typically used when the full list of valid values either is not worth listing (because it has little or no relevance in the given context) or the valid values are too numerous to list in the available space.
- The slash (/) is used to separate paired or complementary values in a list, for example, “the LDBLOCKF/STBLOCKF instruction pair ....”
- The double colon (**::**) is an operator that indicates concatenation (typically, of bit vectors). Concatenation strictly strings the specified component values into a single longer string, in the order specified. The concatenation operator performs no arithmetic operation on any of the component values.

Notation for Numbers

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

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

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

Informational Notes

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

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

Programming Note | Programming notes contain incidental information about how software can use an architectural feature.

Implementation Note | An Implementation Note contains incidental information, describing how an UltraSPARC Architecture processor might implement an architectural feature.

V9 Compatibility Note | Note containing information about possible differences between UltraSPARC Architecture and SPARC V9 implementations. Such information may not pertain to other SPARC V9 implementations.

CHAPTER 1

UltraSPARC T1 Basics

1.1 Background

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

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

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

FIGURE 1-1 Differences Between TLP and ILP

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

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

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

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

1.2 UltraSPARC T1 Overview

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

Each SPARC physical core has a 16-Kbyte, 4-way associative instruction cache (32-byte lines), and 8K-byte, 4-way associative data cache (16-byte lines) that are shared by the four strands. The eight SPARC physical cores are connected through a crossbar to an on-chip unified 3-Mbyte, 12-way associative L2 cache (with 64-byte lines). The L2 cache is banked 4 ways to provide sufficient bandwidth for the eight SPARC physical cores.

1.3 UltraSPARC T1 Components

This section describes each component in UltraSPARC T1:

- SPARC physical core
- Floating-point unit
- L2 cache

1.3.1 SPARC Physical Core

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

FIGURE 1-2 illustrates SPARC physical core.

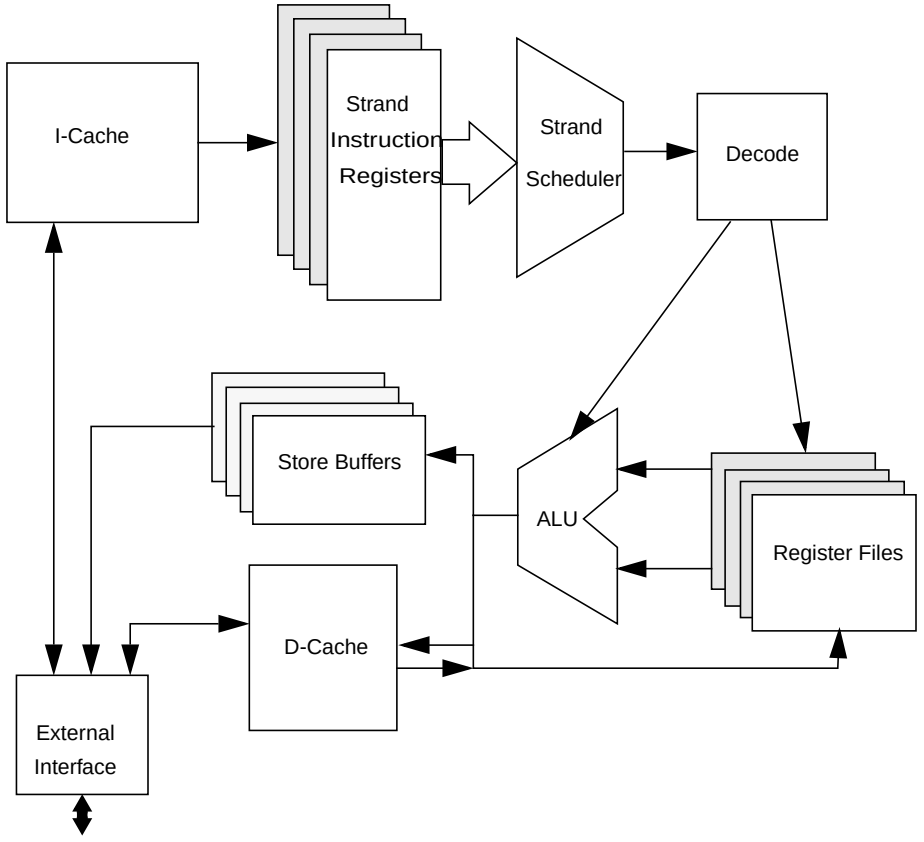


FIGURE 1-2 SPARC Core Block Diagram

1.3.2 Floating-Point Unit (FPU)

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

1.3.3 L2 Cache

The L2 cache is banked four ways, with the bank selection based on address bits 7:6. The cache is 3 Mbytes, 12-way set associative, and has a line size of 64 bytes.

CHAPTER 2

Data Formats

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

CHAPTER 3

Registers

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

3.1 Ancillary State Registers (ASRs)

3.1.1 TICK Register

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

The TICK register contains two fields: npt and counter. On an UltraSPARC T1 processor, the npt field is replicated per strand, while the counter field is shared by all four strands on a physical processor core. The counter increments each physical processor core clock but, on an UltraSPARC T1 processor, the least significant 2 bits of the counter field always read as 0.

3.1.2 General Status Register (GSR)

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

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

3.1.3 Software Interrupt Register (SOFTINT)

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

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

UltraSPARC T1 Programming Note It is possible (but difficult) in UltraSPARC T1 for software to clear a SOFTINT bit between the setting of that bit and the generation of the interrupt from the bit being set because (there is a three-cycle window between the setting of the bit and the interrupt in UltraSPARC T1). If software were to do this, it would see an *interrupt_level_n* interrupt, but would find no bit set in the SOFTINT register. Note that normal software would only clear a bit in response to taking the *interrupt_level_n* exception, so this race condition should not occur in normal operation.

UltraSPARC T1 Programming Note It is possible, but even more difficult than the above case, for software to zero a SOFTINT bit as it is getting set to 1, while another core is accessing its SOFTINT register, with timing such that hardware decides to take a SOFTINT trap, but the SOFTINT register is clear by the time it decides the trap number. In this case, hardware will take a trap 40_{16}. Since software should only clear a bit that is known to be set, this should never happen in normal operation.

3.1.4 Tick Compare Register (TICK_CMPR)

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

3.1.5 System Tick Register (STICK)

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

3.1.6 System Tick Compare Register (STICK_CMPR)

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

3.1.7 PCR and PIC Registers

TABLE 3-1 UltraSPARC T1-Specific Performance Instrumentation Registers

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

Notes:

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

3.2 PR State Registers

3.2.1 Trap State (TSTATE)

Each virtual processor (strand) has *MAXPTL*(2) Trap State (TSTATE) registers, as described in the *UltraSPARC Architecture 2005*.

3.2.2 Processor State Register (PSTATE)

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

3.2.3 Trap Level Register (TL)

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

The maximum trap level (*MAXPTL*) for UltraSPARC T1 is 2.

3.2.4 Global Level Register (GL)

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

The maximum global level (*MAXPGL*) for UltraSPARC T1 is 2.

3.3 Floating-Point State Register (FSR)

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

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

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

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

CHAPTER 4

Instruction Set Overview

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

4.1 State Register Access

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

4.2 Floating-Point Operate (FPop) Instructions

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

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

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

4.3 Reserved Opcodes and Instruction Fields

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

- Attempting to execute a reserved FPop causes an *fp_exception_other* trap (with FSR.ftt = unimplemented_FPop).
- Attempting to execute any other reserved opcode causes an *illegal_instruction* trap.
- Attempting to execute a Tcc instruction with a nonzero value in the reserved field (bits 10:8 and 6:5 when i = 0 or bits 10:7 when i = 1) causes an *illegal_instruction* trap. See *Trap on Integer Condition Codes (Tcc)* on page 16.

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

4.4 Register Window Management

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

CHAPTER 5

Instruction Definitions

5.1 Instruction Set Summary

The UltraSPARC T1 CPU implements both the standard UltraSPARC Architecture 2005 instruction set and a number of implementation-dependent extended instructions. Standard UltraSPARC Architecture 2005 instructions are documented in the *UltraSPARC Architecture 2005*. UltraSPARC T1 extended instructions are documented in *VIS Instructions* on page 16.

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

TABLE 5-1 Instruction Superscripts

Superscript	Meaning
D	Deprecated instruction
P	Privileged instruction

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

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

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

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

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

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

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

5.2 Prefetch and Prefetch from Alternate Space

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

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

5.3 Trap on Integer Condition Codes (Tcc)

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

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

5.4 VIS Instructions

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

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

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

5.5 Partitioned Add/Subtract Instructions

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

UltraSPARC T1 Programming Note

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

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

5.6 Align Data

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

UltraSPARC T1 Programming Note

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

5.7 F Register Logical Operate Instructions

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

UltraSPARC T1 Programming Note

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

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

5.8 Block Load and Store Instructions

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

UltraSPARC T1 Implementation Note

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

Compatibility Note

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

The use of block loads and stores on UltraSPARC T1 is deprecated; they are provided primarily for compatibility with existing software. UltraSPARC T1 provides a separate set of ASIs for high performance BCOPY and BFILL, as described in TABLE 9-1 on page 41. The performance of parallel BCOPY using appropriate ASIs (from among 22_{16}, 23_{16}, $E2_{16}$, $E3_{16}$, EA_{16}, and EB_{16}) will be 2.5 to 3.5 times that of a BCOPY using block loads and stores. The performance of a single-threaded BCOPY using these ASIs will be 15% to 50% better than that of a BCOPY using block loads and stores.

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

Similarly on UltraSPARC T1, STBLOCKF source data registers are not interlocked against completion of previous load instructions (even if a second LDBLOCKF has been performed). The previous load data must be referenced by some other intervening instruction, or an intervening MEMBAR #Sync must be performed. If the programmer violates these rules, data from before or after the load may be used. UltraSPARC T1 continues execution before all of the store data has been transferred. If store data registers are overwritten before the next block store or MEMBAR #Sync instruction, then the following rule must be observed. The first register can be overwritten in the same instruction group as the STBLOCKF, the second register can be overwritten in the instruction group following the block store and so on. If this rule is violated, the store may store correct data or the overwritten data. Block stores always operate under the relaxed memory order (RMO) memory model, regardless of the PSTATE.mm setting, and require a subsequent MEMBAR #Sync to order them with respect to following loads.

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

On UltraSPARC T1, LDBLOCKF does not follow memory model ordering with respect to stores. In particular, read-after-write and write-after-read hazards to overlapping addresses are not detected. The side-effects bit associated with the access is ignored (see *Translation Table Entry (TTE)* on page 53). If ordering with respect to earlier stores is important (for example, a block load that overlaps previous stores), then there must be an intervening MEMBAR #StoreLoad (or stronger MEMBAR). If ordering with respect to later stores is important (for example, a block load that overlaps a subsequent store), then there must be an intervening MEMBAR #LoadStore or reference to the block load data. This restriction does not apply when a trap is taken, so the trap handler need not consider pending block loads. If the LDBLOCKF overlaps a previous or later store and there is no intervening MEMBAR, trap, or data reference, the LDBLOCKF may return data from before or after the store.

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

STBLOCKF does not follow memory model ordering with respect to loads, stores or flushes. In particular, read-after-write, write-after-write, flush-after-write, and write-after-read hazards to overlapping addresses are not detected. The side-effects bit associated with the access is ignored. If ordering with respect to earlier or later loads or stores is important, then there must be an intervening reference to the load data (for earlier loads), or appropriate MEMBAR instruction. This restriction does not apply when a trap is taken, so the trap handler does not have to worry about pending block stores. If the STBLOCKF overlaps a previous load and there is no intervening load data reference or MEMBAR #LoadStore instruction, the load may return data from before or after the store and the contents of the block are undefined. If the STBLOCKF overlaps a later load and there is no intervening trap or MEMBAR #StoreLoad instruction, the contents of the block are undefined. If the STBLOCKF overlaps a later store or flush and there is no intervening trap or MEMBAR #StoreStore instruction, the contents of the block are undefined.

Block load and store operations do not obey the ordering restrictions of the currently selected virtual processor memory model (always TSO in UltraSPARC T1); block operations always execute under an RMO memory ordering model. Explicit MEMBAR instructions are required to order block operations among themselves or with respect to normal loads and stores. In addition, block operations do not conform to dependence order on the issuing strand; that is, no read-after-write or writer-after-read checking occurs between block loads and stores. Explicit MEMBARs must be used to enforce dependence ordering between block operations that reference the same address.

Typically, LDBLOCKF and STBLOCKF are used in loops where software can ensure that there is no overlap between the data being loaded and the data being stored. The loop must be preceded and followed by the appropriate MEMBARs to ensure that there are no hazards with loads and stores outside the loops. CODE EXAMPLE 5-1 illustrates the inner loop of a byte-aligned block copy operation.

Note that the loop must be unrolled twice to achieve maximum performance. All FP register references in this code example are to 64-bit registers. Eight versions of this loop are needed to handle all the cases of double word misalignment between the source and destination.

CODE EXAMPLE 5-1 Byte-Aligned Block Copy Inner Loop

```
loop:
   faligndata   %f0, %f2, %f34
   faligndata   %f2, %f4, %f36
   faligndata   %f4, %f6, %f38
   faligndata   %f6, %f8, %f40
   faligndata   %f8, %f10, %f42
   faligndata   %f10, %f12, %f44
   faligndata   %f12, %f14, %f46
   addcc        %l0, -1, %l0
   bg,pt        l1
   fmovd        %f14, %f48
     end of loop handling
l1: ldda        [regaddr] #ASI_BLK_P, %f0
   stda         %f32, [regaddr] #ASI_BLK_P
   faligndata   %f48, %f16, %f32
   faligndata   %f16, %f18, %f34
   faligndata   %f18, %f20, %f36
   faligndata   %f20, %f22, %f38
   faligndata   %f22, %f24, %f40
   faligndata   %f24, %f26, %f42
   faligndata   %f26, %f28, %f44
   faligndata   %f28, %f30, %f46
   addcc        %l0, -1, %l0
   be,pnt       done
   fmovd        %f30, %f48
   ldda         [regaddr] #ASI_BLK_P, %f16
   stda         %f32, [regaddr] #ASI_BLK_P
   ba           loop
   faligndata   %f48, %f0, %f32
done:   end of loop processing
```

5.9 Block Initializing Store ASIs

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

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

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

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

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

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

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

UltraSPARC T1 Implementation Note | On UltraSPARC T1, a noncacheable address is identified by .

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

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

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

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

Programming Note

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

```
%l0 ← [A]; %l1 ← [B]
prefetch [%l0]
for (i = 0;  i < N-4;  i++) {
   if (!(i % 4)) { prefetch [%l0+64] }
   ldda [%l0] #ASI_BLK_INIT_ST_P, %l2
   add %l0, 16, %l0
   stxa %l2, [%l1] #ASI_BLK_INIT_ST_P
   add %11, 8, %11
   stxa %l3, [%l1+8] #ASI_BLK_INIT_ST_P
   add %l1, 8, %l1
}
for (i = 0;  i < 4;  i++) {
   ldda [%l0] #ASI_BLK_INIT_ST_P, %l2
   add %l0, 16, %l0
   stx %l2, [%l1]
   stx %l3, [%l1+8]
   add %l1, 16, %l1
}
membar #Sync
```

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

UltraSPARC T1 Programming Notes

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

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

Exceptions

VA_watchpoint
mem_address_not_aligned
data_access_exception

5.10 Load Twin Extended Word Instructions (nonprivileged)

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

Description

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

See the *UltraSPARC Architecture 2005* for details.

Programming Note

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

UltraSPARC T1 Implementation Note

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

See the description of Block Initializing Stores for an example of how Load Twin Doubleword can be used in combination with those instructions.

UltraSPARC T1 Programming Notes

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

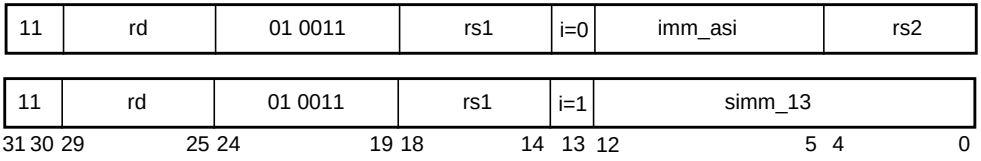
(2) These ASIs provide a higher performance `bcopy()` or `bfill()` than the block loads and stores described in Section 5.8, due to their ability to overlap multiple loads and stores between strands and to avoid the unnecessary fetch from memory of the data that is overwritten by the store. The performance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

See Also Block Initializing Store ASIs on page 21.

5.11 Load Twin Extended Word Instructions (privileged)

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

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



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

Description

These instructions atomically read a 128-bit data item into two 64-bit integer registers. They are intended to be used to access TSB entries without requiring locks. The data is placed in an even/odd pair of 64-bit integer registers. The lowest address 64 bits is placed in the even-numbered register; the highest address 64-bits is placed in the odd-numbered register.

`ASI_LDTX_REAL{_L}` bypasses the virtual-to-real portion of the translation, setting RA{63:0} = VA{63:0}.

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

Exceptions

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

CHAPTER 6

Traps

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

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

6.1 Trap Levels

Each UltraSPARC T1 virtual processor supports two trap levels (*MAXPTL* = 2) .

CHAPTER 7

Interrupt Handling

7.0.1 Interrupt Queue Registers

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

The tail registers are read-only. An attempted write to a tail register by privileged software generate a *data_access_exception* trap. The head registers are read/write.

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

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

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

Programming Note

These registers are intended to be used as head and tail pointers into a queue in memory storing the mondo or error interrupt data. When an interrupt is taken, the interrupt data are stored into the end of the appropriate queue. Then the corresponding tail register is updated to point beyond the new data, which causes a trap to be generated to privileged software (the operating system). Privileged software then processes the interrupt data from the head of the queue, updating the head register when the interrupt processing is completed.

While the first interrupt is being serviced, more interrupts may be placed on the queue. The operating system can read the tail pointer to service multiple interrupts at a time, or it can simply update the head pointer after each interrupt has been serviced and take a trap for each interrupt.

When all pending interrupts of the appropriate type have been serviced, the head and tail pointers will be equal again, and no further traps will be generated until new interrupt data is placed on the queue.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 8

Memory Models

8.1 Overview

SPARC V9 defines the semantics of memory operations for three memory models. From strongest to weakest, they are Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory Order (RMO). The differences in these models lie in the freedom an implementation is allowed in order to obtain higher performance during program execution. The purpose of the memory models is to specify any constraints placed on the ordering of memory operations in uniprocessor and shared-memory multiprocessor environments.

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

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

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

The current memory model is indicated in the PSTATE.mm field. Its value is always 0 on UltraSPARC T1. An UltraSPARC T1 virtual processor always operates under the TSO memory model.

Memory is logically divided into real memory (cached) and I/O memory (noncached, with and without side effects) spaces (impl. dep. #118-V9). Real memory spaces may be cached and can be accessed without side effects. For example, a read (load) from real memory space returns the information most recently written. In

addition, an access to real memory space does not result in program-visible side effects. In contrast, a read from I/O space may not return the most recently written information and may result in program-visible side effects.

8.2 Supported Memory Models

The following sections contain brief descriptions of the two memory models supported by UltraSPARC T1. These definitions are for general illustration. Detailed definitions of these models can be found in *UltraSPARC Architecture 2005*. The definitions in the following sections apply to system behavior as seen by the programmer. A description of MEMBAR can be found in Section 8.3.2, "Memory Synchronization: MEMBAR and FLUSH" on page 72.

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

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

8.2.1 Total Store Order

UltraSPARC T1 implements the following programmer-visible properties in Total Store Order (TSO) mode:

- Loads are processed in program order; that is, there is an implicit MEMBAR `#LoadLoad` between them.
- Loads may bypass earlier stores. Any such load that bypasses such earlier stores must check (snoop) the store buffer for the most recent store to that address. A MEMBAR `#Lookaside` is not needed between a store and a subsequent load at the same noncacheable address.
- A MEMBAR `#StoreLoad` must be used to prevent a load from bypassing a prior store, if Strong Sequential Order is desired.
- Stores are processed in program order.
- Stores cannot bypass earlier loads.
- An L2 cache update is delayed on a store hit until all outstanding stores reach global visibility. For example, a cacheable store following a noncacheable store is not globally visible until the noncacheable store has reached global visibility; there is an implicit MEMBAR `#MemIssue` between them.

8.2.2 Relaxed Memory Order

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

- There is no implicit order between any two memory references, either cacheable or noncacheable, except that noncacheable accesses 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 non-cacheable accesses. A MEMBAR `#StoreLoad` 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 T1 supports a 48-bit virtual address space.

9.1.1 Access to Nonexistent Memory or I/O

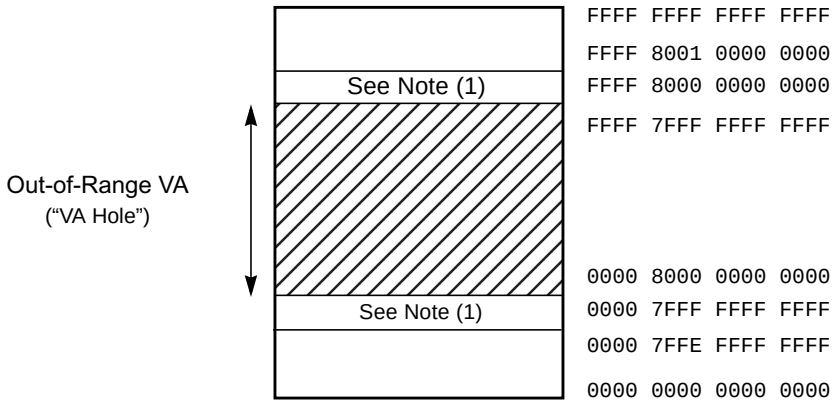
Accesses to nonexistent memory or I/O locations are treated as follows:

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

9.1.2 48-bit Virtual Address Space

UltraSPARC T1 supports a 48-bit subset of the full 64-bit virtual address space (see FIGURE 9-1). Although the full 64 bits are generated and stored in integer registers, legal addresses are restricted to two equal halves at the extreme lower and upper portions of the full virtual address space. Virtual addresses between $0000\ 8000\ 0000\ 0000_{16}$ and $FFFF\ 7FFF\ FFFF\ FFFF_{16}$, inclusive, lie within a “VA Hole”, are termed “out of range,” and are illegal.

Prior UltraSPARC implementations introduced the additional restriction on software to not use pages within 4 Gbytes of the VA hole as instruction pages, to avoid problems with prefetching into the VA hole. UltraSPARC T1 assumes that this convention is followed, for similar reasons. Note that there are no trap mechanisms to detect a violation of this convention.



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

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

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

A number of state registers are affected by the reduced virtual address space. TBA, TPC, and TNPC registers are 48 bits wide, sign-extended to 64 bits on read accesses. VA watchpointing is 48 bits, zero-extended to 64-bits on read accesses. No checks are done when these registers are written by software. It is the responsibility of privileged software to properly update these registers.

An out of range address during an instruction access causes an *instruction_access_exception* trap if PSTATE.am = 0.

If the target address of a JMPL or RETURN instruction is an out-of-range address and PSTATE.am is not set, a trap is generated with TPC[TL] set to the address of the JMPL or RETURN instruction. This *instruction_access_exception* trap is lower priority than other traps on the JMPL or RETURN (*illegal_instruction* due to nonzero reserved fields in the JMPL or RETURN, *mem_address_not_aligned* trap, or *window_fill* trap), because it really applies to the target. The trap handler can determine the out-of-range address by decoding the JMPL instruction from the code.

When any other control transfer instruction traps, it sets TPC[TL] to the address of the target instruction. Because the PC is sign-extended to 64 bits, the trap handler must adjust the PC value to compute the faulting address by **xor**ing ones into the most significant 16 bits.

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

To guarantee that all out of range instruction accesses cause traps, software should not map addresses within 2^{31} bytes of either side of the VA hole as executable.

An out-of-range address during a data access results in a *data_access_exception* trap if PSTATE.am is not set.

9.2 Alternate Address Spaces

TABLE 9-1 summarizes the ASI usage in UltraSPARC T1. The Section column lists where the operation of the ASI is explained. For several internal ASIs, a range of legal VAs is listed. An access outside the legal VA range will be aliased to a legal VA by ignoring the upper address bits.

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

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

TABLE 9-1 UltraSPARC T1 ASI Usage *(1 of 5)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
00_{16}–03_{16}			Any	—	*data_access_exception*	
04_{16}	ASI_NUCLEUS	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
05_{16}–$0B_{16}$			Any	—	*data_access_exception*	
$0C_{16}$	ASI NUCLEUS_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	

TABLE 9-1 UltraSPARC T1 ASI Usage *(2 of 5)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$0D_{16}$–$0F_{16}$			Any	—	*data_access_exception*	
10_{16}	ASI_AS_IF_USER_PRIMARY	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
11_{16}	ASI_AS_IF_USER_SECONDARY	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
12_{16}–13_{16}			Any	—	*data_access_exception*	
14_{16}	ASI_REAL	RW	Any	—	(See *UltraSPARC Architecture 2005*)	9.2.1
15_{16}	ASI_REAL_IO	RW	Any	—	(See *UltraSPARC Architecture 2005*)	9.2.2
16_{16}	ASI_BLOCK_AS_IF_USER_PRIMARY	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.8
17_{16}	ASI_BLOCK_AS_IF_USER_SECONDARY	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.8
18_{16}	ASI_AS_IF_USER_PRIMARY_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
19_{16}	ASI_AS_IF_USER_SECONDARY_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
$1A_{16}$–$1B_{16}$			Any	—	*data_access_exception*	
$1C_{16}$	ASI_REAL_LITTLE	RW	Any	—	Nonallocating in L1 cache, same as ASI_REAL_IO_LITTLE for I/O addresses	9.2.1
$1D_{16}$	ASI_REAL_IO_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	9.2.2
$1E_{16}$	ASI_BLOCK_AS_IF_USER_PRIMARY_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.8
$1F_{16}$	ASI_BLOCK_AS_IF_USER_SECONDARY_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.8
20_{16}	ASI_SCRATCHPAD	RW	0_{16}–18_{16}	Y	Scratchpad registers 0–3	9.2.3
		RW	20_{16}–28_{16}	—	*data_access_exception*	9.2.3
		RW	30_{16}–38_{16}	Y	Scratchpad registers 6–7	9.2.3
21_{16}	ASI_MMU_CONTEXTID	RW	0_{16}–$F8_{16}$	—	(See *UltraSPARC Architecture 2005*)	

TABLE 9-1 UltraSPARC T1 ASI Usage *(3 of 5)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
22_{16}	ASI_LDTX_AIUP,	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.10
	ASI_STBI_AIUP				ASI_STBI_AIUP is used for Block-Initializing stores, As If User, Primary Context	
23_{16}	ASI_LDTX_AIUS,	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.10
	ASI_STBI_AIUS				ASI_STBI_AIUS is used for Block-Initializing stores, As If User, Secondary Context	
24_{16}	ASI_TWINX (ASI_LDTX), ASI_QUAD_LDD^{D}†, ASI_NUCLEUS_QUAD_LDD^{D}†	R	Any	—	128-bit atomic Load Twin Doubleword (deprecated; superseded by ASI 27_{16})	
25_{16}	ASI_QUEUE	RW	0_{16}–$3B8_{16}$	—	Load/store does NOP	
		RW	$3C0_{16}$–$3F8_{16}$	Y	(See *UltraSPARC Architecture 2005*)	
26_{16}	ASI_LDTX_REAL	R	Any	—	128-bit atomic LDTX, real address (see *UltraSPARC Architecture 2005*)	5.11
27_{16}	ASI_LDTX_N,	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.10
	ASI_STBI_N				ASI_STBI_N is used for Block-Initializing stores, Nucleus Context	
28_{16}–29_{16}			Any	—	*data_access_exception*	
$2A_{16}$	ASI_LDTX_AIUP_L,	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.10
	ASI_STBI_AIUP_L				ASI_STBI_AIUP_L is used for Block-Initializing stores, As If User, Primary Context, Little Endian	
$2B_{16}$	ASI_LDTX_AIUS_L,	RW	Any	—	(See *UltraSPARC Architecture 2005*)	5.10
	ASI_STBI_AIUS_L				ASI_STBI_AIUS_L is used for Block-Initializing stores, As If User, Seondary Context, Little Endian	
$2C_{16}$	ASI_TWINX_LITTLE (ASI_LDTX_L), ASI_QUAD_LDD_LITTLE^{D}†, ASI_NUCLEUS_QUAD_LDD_LITTLE^{D}†	R	Any	—	128-bit atomic Load Twin Doubleword, little endian (deprecated; superseded by ASI $2F_{16}$)	

TABLE 9-1 UltraSPARC T1 ASI Usage *(4 of 5)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
$2D_{16}$			Any	—	*data_access_exception*	
$2E_{16}$	ASI_LDTX_REAL_L	R	Any	—	128-bit atomic LDTX, real address, little endian (see *UltraSPARC Architecture 2005*)	5.11
$2F_{16}$	ASI_LDTX_NL, ASI_STBI_NL	RW	Any	—	(See *UltraSPARC Architecture 2005*) ASI_STBI_NL is used for Block-Initializing stores, Nucleus context, Little-Endian	5.10
80_{16}	ASI_PRIMARY	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
81_{16}	ASI_SECONDARY	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
82_{16}	ASI_PRIMARY_NO_FAULT	R	Any	—	(See *UltraSPARC Architecture 2005*)	
83_{16}	ASI_SECONDARY_NO_FAULT	R	Any	—	(See *UltraSPARC Architecture 2005*)	
84_{16}–87_{16}			Any	—	*data_access_exception*	
88_{16}	ASI_PRIMARY_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
89_{16}	ASI_SECONDARY_LITTLE	RW	Any	—	(See *UltraSPARC Architecture 2005*)	
$8A_{16}$	ASI_PRIMARY_NO_FAULT_LITTLE	R	Any	—	(See *UltraSPARC Architecture 2005*)	
$8B_{16}$	ASI_SECONDARY_NO_FAULT_LITTLE	R	Any	—	(See *UltraSPARC Architecture 2005*)	
$8C_{16}$–BF_{16}			Any	—	*data_access_exception*	
$C0_{16}$	ASI_PST8_P		Any	—	*data_access_exception*[1]	
$C1_{16}$	ASI_PST8_S		Any	—	*data_access_exception*[1]	
$C2_{16}$	ASI_PST16_P		Any	—	*data_access_exception*[1]	
$C3_{16}$	ASI_PST16_S		Any	—	*data_access_exception*[1]	
$C4_{16}$	ASI_PST32_P		Any	—	*data_access_exception*[1]	
$C5_{16}$	ASI_PST32_S		Any	—	*data_access_exception*[1]	
$C6_{16}$–$C7_{16}$			Any	—	*data_access_exception*	
$C8_{16}$	ASI_PST8_PL		Any	—	*data_access_exception*[1]	
$C9_{16}$	ASI_PST8_SL		Any	—	*data_access_exception*[1]	
CA_{16}	ASI_PST16_PL		Any	—	*data_access_exception*[1]	
CB_{16}	ASI_PST16_SL		Any	—	*data_access_exception*[1]	

TABLE 9-1 UltraSPARC T1 ASI Usage *(5 of 5)*

ASI	ASI NAME	R/W	VA	Copy per strand	Description	Section
CC_{16}	ASI_PST32_PL		Any	—	*data_access_exception*[1]	
CD_{16}	ASI_PST32_SL		Any	—	*data_access_exception*[1]	
CE_{16}–CF_{16}			Any	—	*data_access_exception*	
$D0_{16}$	ASI_FL8_P		Any	—	*data_access_exception*[2]	
$D1_{16}$	ASI_FL8_S		Any	—	*data_access_exception*[2]	
$D2_{16}$	ASI_FL16_P		Any	—	*data_access_exception*[2]	
$D3_{16}$	ASI_FL16_S		Any	—	*data_access_exception*[2]	
$D4_{16}$–$D7_{16}$			Any	—	*data_access_exception*	
$D8_{16}$	ASI_FL8_PL		Any	—	*data_access_exception*[2]	
$D9_{16}$	ASI_FL8_SL		Any	—	*data_access_exception*[2]	
DA_{16}	ASI_FL16_PL		Any	—	*data_access_exception*[2]	
DB_{16}	ASI_FL16_SL		Any	—	*data_access_exception*[2]	
DC_{16}–DF_{16}			Any	—	*data_access_exception*	
$E0_{16}$	ASI_BLK_COMMIT_P	RW	Any	—	*data_access_exception*[3]	
$E1_{16}$	ASI_BLK_COMMIT_S	RW	Any	—	*data_access_exception*[3]	
$E4_{16}$–$E9_{16}$			Any	—	*data_access_exception*	
EC_{16}–EF_{16}			Any	—	*data_access_exception*	
$F0_{16}$	ASI_BLK_P	RW	Any	—	64-byte block load/store, primary address	5.8
$F1_{16}$	ASI_BLK_S	RW	Any	—	64-byte block load/store, secondary address	5.8
$F2_{16}$–$F7_{16}$			Any	—	*data_access_exception*	
$F8_{16}$	ASI_BLK_PL	RW	Any	—	64-byte block load/store, primary address, little endian	5.8
$F9_{16}$	ASI_BLK_SL	RW	Any	—	64-byte block load/store, secondary address, little endian	5.8
FA_{16}–FF_{16}			Any	—	*data_access_exception*	

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

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

9.2.1 ASI_REAL and ASI_REAL_LITTLE

This ASI is used to bypass the data MMU for memory addresses. Since the cp page attribute bit is clear, load accesses using this ASI will always fetch their data from the L2 cache. Using this ASI for an I/O address is permitted, and will follow the same page attributes (w = 1, all other attributes 0).

Programming Note Although it is permitted to use ASI_REAL (or ASI_REAL_LITTLE) for an I/O access, it is not recommended to do so because the e bit is not set for the access. ASI_REAL_IO and ASI_REAL_IO_LITTLE should be used instead.

9.2.2 ASI_REAL_IO and ASI_REAL_IO_LITTLE

This ASI is used to bypass the data MMU for I/O addresses. The physical page attributes e and w are set to 1 and all other attribute bits are set to 0 for accesses to this ASI. Using this ASI for a memory address is permitted, and will follow the same page attributes (e = 1, w = 1, all other attributes 0).

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

9.2.3 ASI_SCRATCHPAD

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

TABLE 9-2 defines the format of these registers.

TABLE 9-2 Scratchpad – `ASI_SCRATCHPAD` (ASI 20_{16}; VA 0_{16}, 8_{16}, 10_{16}, 18_{16}, 30_{16}, or 38_{16})

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

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

CHAPTER **10**

Performance Instrumentation

10.1 Performance Control Register

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

- ovfh and ovfl are state bits associated with the PIC.h and PIC.l overflow traps and are provided in this register to allow swapping out of a process that is in the state between the counter overflowing and the overflow trap being generated.
- sl controls which events are counted in PIC.l.
- ut controls whether user-level (nonprivileged) events are counted.
- st controls whether supervisor-level (privileged) events are counted.
- priv controls whether the PIC register can be read or written by nonprivileged software.

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

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

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

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

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

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

TABLE 10-2 sl Field Settings

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

1. Tcc instructions that are cancelled due to encountering a higher-priority trap are still counted.
2. SB_full increments every cycle a strand (virtual processor) is stalled due to a full store buffer, regardless of whether other strands are able to keep the processor busy. The overflow trap for SB_full is not precise to the instruction following the event that occurs when ovfl is set (the trap may occur on either the instruction following the event that occurs when ovfl is set, or on either of the next two instructions).
3. Only floating point instructions which execute in the shared FPU are counted. The following instructions are executed in the shared FPU: FADDS, FADDD, FSUBS, FSUBD, FMULS, FMULD, FDIVS, FDIVD, FSMULD, FSTOX, FDTOX, FXTOS, FXTOD, FITOS, FDTOS, FITOD, FSTOD, FSTOI, FDTOI, FCMPS, FCMPD, FCMPES, FCMPED.
4. L2 misses due to stores cannot be counted by the performance instrumentation logic.

10.2 SPARC Performance Instrumentation Counter

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

The ut and st fields for PCR control whether events from user (nonprivileged) mode, supervisor (privileged) mode, both, or neither are counted. Whenever PCR.ovfh is set (which normally occurs when the PIC.h counter overflows, but may also be set via a write to the PCR), a disrupting trap is generated on the next event that increments the counter.. This trap will appear to be precise to the instruction following the one that caused the event

Programming Note

A WRasr to PCR that modifies the ovfh or ovfl bit behaves as if the ovfh or ovfl bit was modified before the WRasr is executed.

This implies that if all of the following conditions are true, a performance counter overflow trap will be taken (to hyperprivileged software) on the instruction following the WRasr:

- a WRasr is executed in privileged mode that sets ovfh or ovfl to 1
- PCR.st = 1
- the WRasr generates the event being counted

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

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

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

CHAPTER 11

Memory Management

11.1 Translation Table Entry (TTE)

The Translation Table Entry (TTE) holds information for a single page mapping. The TTE is broken into two 64-bit words, representing the tag and data of the translation. Just as in a hardware cache, the tag is used to determine whether there is a hit in the TSB. If there is a hit, the data is fetched by software.

11.1.1 TTE Tag Format

UltraSPARC T1 supports both the UltraSPARC Architecture 2005 TTE tag format (as described in the *UltraSPARC Architecture 2005* specification; also known as the "sun4v" TTE format) and the older sun4u TTE tag format.

Note that UltraSPARC T1 only supports 13-bit context IDs; therefore, the most significant 3 bits of the (16-bit) context field are always zero.

UltraSPARC T1 supports 48-bit virtual addresses in hardware. When hardware writes a 48-bit virtual address into a 64-bit register, it sign-extends (copies) the most significant address bit (bit 47) into bits 63:48 of the register.

11.1.2 TTE Data Format

For the data portion of the TTE, both the sun4v and sun4u formats are supported by UltraSPARC T1. The sun4v TTE data format is described in the *UltraSPARC Architecture 2005* specification.

UltraSPARC Architecture 2005 specifies a 4-bit size field for TTE entries. Since UltraSPARC T1 only supports a 3-bit size field, the most significant bit of TTE (bit 3) is ignored when written.

In the sun4u TTE virtual address tag, bits 63:22 are used. Bits 21 through 13 are not maintained in the tag, since these bits are used to index the smallest direct-mapped TSB of 512 entries.

The sun4u TTE data format is shown in TABLE 11-1.

TABLE 11-1 Format 16 Sun4u TTE Data Format

Bit	*Field*	*Description*
63	v	Valid
62:61	szl	size{1:0}
60	nfo	No-fault-only
59	ie	Invert endianness
58:49	soft2	Soft2
48	szh	size{2}
47:40	diag	Diagnostic
39:13	pa	PA{39:13}
12:8	soft	Soft
7	—	Reserved
6	l	Locked
5	cp	Cacheable in physically indexed cache
4	cv	Cacheable in virtually indexed cache
3	e	Side effect
2	p	Privileged
1	w	Writable
0	—	Reserved

TABLE 11-2 provides UltraSPARC T1-specific information regarding sun4u TTE data fields.

TABLE 11-2 *TTE Field Description*

Field	*Description*
nfo	No-Fault-Only..
ie	Invert Endianness.
soft, soft2	Software-defined fields, provided for use by the operating system. Software fields are not implemented in UltraSPARC T1 hardware.
diag	Used by diagnostics
pa	The physical page number.
l	Lock. If this bit is set, the TTE entry will be “locked down”.
w	Writable.

11.2 Translation Storage Buffer

A Translation Storage Buffer (TSB) is an array of TTEs managed entirely by software. It serves as a cache of the Software Translation Table. The discussion in this section assumes the use of hardware support for TSB access, although the operating system is not required to make use of this support hardware.

Inclusion of the TLB entries in a TSB is not required; that is, translation information may exist in the TLB that is not present in the TSB.

A TSB is arranged as a direct-mapped cache of TTEs.The *n* least significant bits of a virtual page number is used as the offset from the respective TSB base address, where *n* equals $\log_2$ of the number of TTEs in the TSB.

A bit in the TSB register allows the PS0 and PS1 pointers to be computed for the case of separate or split PS0/PS1 TSB(s).

No hardware TSB indexing support is provided for TTEs of pages other than PS0 and PS1. Since the TSB is entirely software managed, however, the operating system may choose to place these different page TTEs in the TSB by forming the appropriate pointers. In addition, simple modifications to the PS0 and PS1 index pointers provided by the hardware allow formation of an M-way set-associative TSB, multiple TSBs per page size, and multiple TSBs per process.

The TSB exists as a normal data structure in memory, and therefore may be cached. Indeed, the speed of the TLB miss handler relies on the TSB accesses hitting the level-2 cache at a substantial rate. This policy may result in some conflicts with normal instruction and data accesses, but the dynamic sharing of the level-2 cache resource should provide a better overall solution than that provided by a fixed partitioning.

FIGURE 13-1 shows both the common and shared TSB organization. The constant *n* is determined by the size field in the TSB register; it may range from 512 entries to 16 M entries.

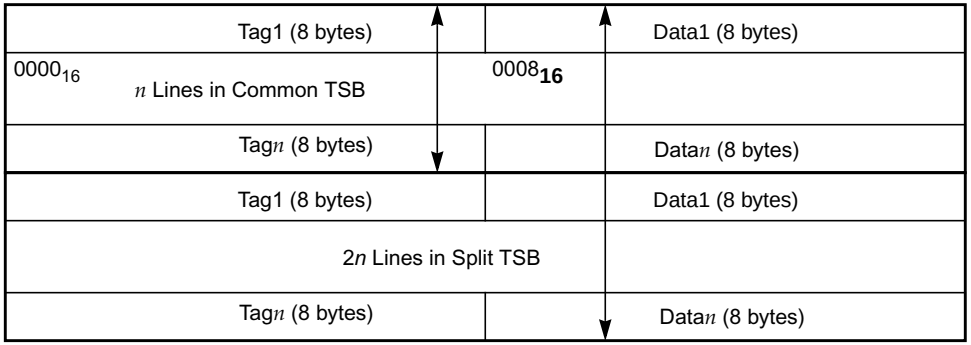


FIGURE 11-1 TSB Organization

11.3 MMU-Related Faults and Traps

MMU traps are described in TABLE 11-3.

TABLE 11-3 MMU Trap Description

Trap	Description
data_access_exception	Occurs when one of the following events (the D-MMU does not prioritize these and may set multiple bits) occurs: • The D-MMU detects a privilege violation for a data access; that is, an attempted access to a privileged page when PSTATE.priv = 0. • A speculative (nonfaulting) load instruction issued to a page marked with the side-effect (e) bit = 1. • An atomic instruction issued to an I/O address (that is, VA{39} = 1). • An invalid LDA/STA ASI value, invalid virtual address, read to write-only register, or write to read-only register, but not for an attempted user access to a restricted ASI (see the *privileged_action* trap described below) • An access with an ASI other than ASI_<PRIMARY,SECONDARY>_NO_FAULT[_LITTLE] to a page marked with the nfo (no-fault-only) bit. • Virtual address out of range and PSTATE.am is not set. See *48-bit Virtual Address Space* on page 39 for details.
instruction_access_exception	Occurs when the I-MMU is enabled and one of the following happens: • The I-MMU detects a privilege violation for an instruction fetch; that is, an attempted access to a privileged page when PSTATE.priv = 0. • Virtual address out of range and PSTATE.am is not set. See *48-bit Virtual Address Space* on page 39. Note that the case of JMPL/RETURN and branch-CALL-sequential are handled differently.
mem_address_not_aligned	Occurs when a load, store, atomic, or JMPL/RETURN instruction with a misaligned address is executed.
privileged_action	Occurs when an access is attempted using a *restricted* ASI while in nonprivileged mode (PSTATE.priv = 0).
VA_watchpoint	Occurs when virtual watchpoints are enabled and the D-MMU detects a load or store to the virtual address specified by the VA Data Watchpoint register.

CHAPTER 12

Implementation Dependencies

12.1 SPARC V9 General Information

12.1.1 Level-2 Compliance (Impl. Dep. #1)

UltraSPARC T1 is designed to meet Level-2 SPARC V9 compliance. It does the following:

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

Note | System emulation routines (for example, for quad-precision floating-point operations) shipped with UltraSPARC T1 also must be Level-2 compliant.

12.1.2 Unimplemented Opcodes, ASIs, and ILLTRAP

SPARC V9 unimplemented instructions, *reserved* instructions, ILLTRAP opcodes, and instructions with invalid values in *reserved* fields (other than *reserved* FPops and the *reserved* field in the Tcc instruction) encountered during execution cause an *illegal_instruction* trap. Reserved FPops cause an *fp_exception_other* (with FSR.ftt = unimplemented_FPop) trap. Unimplemented and *reserved* ASI values cause a *data_access_exception* trap.

12.1.3 Trap Levels (Imp. Dep. #37, 38, 39, 40, 101, 114, 115)

UltraSPARC T1 supports two trap levels; that is, *MAXPTL* = 2. Normal execution is at TL = 0.

A strand normally executes at trap level 0 (`execute_state`, TL = 0). In SPARC V9, a trap makes the CPU enter the next higher trap level, which is a fast and efficient process because there is one set of trap state registers for each trap level. After saving the most important machine states (PC, NPC, PSTATE) on the trap stack at this level, the trap (or error) condition is processed.

12.1.4 Trap Handling (Imp. Dep. #16, 32, 33, 35, 36, 44)

UltraSPARC T1 supports precise trap handling for all operations except for disrupting traps from hardware failures and interrupts. UltraSPARC T1 implements precise traps, interrupts, and exceptions for all instructions, including long latency floating-point operations.

UltraSPARC T1 can efficiently execute kernel code even in the event of multiple nested traps, promoting processor efficiency while dramatically reducing the system overhead needed for trap handling. Three sets of global registers are provided(*MAXPGL* = 2), for use at TL = 0, TL = 1, and TL = 2.

This further increases OS performance, providing fast trap execution by avoiding the need to save and restore registers while processing exceptions.

All traps supported in UltraSPARC T1 are listed in the "Traps" chapter of this document.

12.1.5 Population Count Instruction (POPC)

The population count instruction, POPC, generates an *illegal_instruction* exception and is emulated in software rather that being executed in hardware.

12.1.6 Secure Software

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

12.1.7 Address Masking (Impl. Dep. #125)

When PSTATE.am = 1, the CALL, JMPL, and RDPC instructions and all traps transmit zero in the high-order 32-bits of the PC to their specified destination registers. Traps also transmit zero in the high-order 32-bits of the NPC to the TNPC. Branch target addresses sent to the NPC and the updating of NPC with NPC+4 for a

non-control-transferring instruction do not zero the high-order 32-bits. Restoration of PC and NPC from TPC and TNPC on a DONE or RETRY instruction do not mask the high-order 32-bits.

Note | When PSTATE.am = 1, address masking applies to all VAs, even those that immediately do a VA-to-RA bypass. This implies that with PSTATE.am = 1, RA{63:32} will be zeros after a VA-to-RA bypass.

12.2 SPARC V9 Integer Operations

12.2.1 Integer Register File and Window Control Registers (Impl. Dep. #2)

UltraSPARC T1 implements an eight-window 64-bit integer register file; that is, *N_REG_WINDOWS* = 8. UltraSPARC T1 truncates values stored in the CWP, CANSAVE, CANRESTORE, CLEANWIN, and OTHERWIN registers to three bits. This includes implicit updates to these registers by SAVE(D) and RESTORE(D) instructions. The most-significant two bits of these registers read as zero.

12.2.2 SAVE Instruction

Upon a SAVE instruction, UltraSPARC T1 initializes the values of the local registers in the new window to the same values as the local registers in the old window and initializes the values of the *out* registers in the new window to the same values as the *in* registers in the old window (that is, the new window matches the old window with the *ins* and *outs* swapped). Since this implies that they contain values from the executing process, V9 compliance is maintained. In this sense, the behavior of the SAVE instruction on UltraSPARC T1 differs from most other SPARC V9 implementations.[1]

1. Most SPARC V9 processors do not initialize the *local* and *out* registers on a save instruction; instead, the values in the *local* and *out* registers are those left there from the last time the window was used.

12.2.3 Clean Window Handling (Impl. Dep. #102)

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

When a SAVE instruction requests a window and there are no more clean windows, a *clean_window* trap is generated. Note that the behavior on a *clean_window* trap for UltraSPARC T1 is the same as for a SAVE instruction, namely, the *local* registers for the new window remain the same as the *local* registers from the old window, while the *out* registers in the new window contain the contents of the *in* registers from the old window. Thus, while UltraSPARC T1 generates a *clean_window* trap, the new window is automatically cleaned by hardware. System software only needs to increment CLEANWIN before returning to the requesting context.

12.2.4 Integer Multiply and Divide

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

12.2.5 MULScc

SPARC V9 does not define the value of xcc and R[rd]{63:32} for MULScc. UltraSPARC T1 sets xcc and rd based on the results of adding either (32 copies of R[rs1]{63}**::** CCR.icc.n **xor** CCR.icc.v, R[rs1]{31:1}) or 0 (depending on Y{0}) to either R[rs2]{63:0} or the immediate operand.

12.3 SPARC V9 Floating-Point Operations

12.3.1 Subnormal Operands and Results: Nonstandard Operation

UltraSPARC T1 handles all cases of subnormal operands or results directly in hardware.

Because there is no trapping on subnormal operands, UltraSPARC T1 does not support the nonstandard result option of the SPARC V9 architecture, and the FSR.ns bit ignores any value written to it and always returns zero on a read.

12.3.2 Overflow, Underflow, and Inexact Traps (Impl. Dep. #3, 55)

UltraSPARC T1 implements precise floating-point exception handling. Underflow is detected before rounding.

Note | Major performance degradation may be observed while running with the inexact exception enabled.

12.3.3 Quad-Precision Floating-Point Operations (Impl. Dep. #3)

All quad-precision floating-point instructions, listed in TABLE 12-1, cause an *fp_exception_other* (with FSR.ftt = 3, unimplemented_FPop) trap. These operations are emulated in system software.

TABLE 12-1 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

TABLE 12-1 Unimplemented Quad-Precision Floating-Point Instructions *(Continued)*

Instruction	Description
FNEGq	Quad-precision floating-point negation
FSQRTq	Quad-precision floating-point square root
FSUBq	Quad-precision floating-point subtraction

12.3.4 Floating-Point Square Root

The three floating-point square root instructions: FSQRTS, FSQRTD, FSQRTQ are unimplemented. Execution of any of these instructions results in an *fp_exception_other* exception, with FSR.ftt= unimplemented_FPop.

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

The FPRS_dirty_upper (du) and FPRS_dirty_lower (dl) bits in the Floating-Point Registers State (FPRS) register are set when an instruction that modifies the corresponding upper and lower half of the floating-point register file is issued. Floating-point register file modifying instructions include floating-point operate, graphics, floating-point loads, and block load instructions.

The FPRS.du and FPRS.dl may be set pessimistically, even though the instruction that modified the floating-point register file is nullified due to a trap. This includes the case where the floating-point instruction itself takes a *fp_disabled* trap.

12.3.6 Floating-Point State Register (FSR) (Impl. Dep. #13, 19, 22, 23, 24)

UltraSPARC T1 supports precise-traps and implements all three exception fields (tem, cexc, and aexc) conforming to IEEE Standard 754-1985. TABLE 12-2 defines the register bits.

TABLE 12-2 Floating-Point Status Register Format

Bits	Field	RW	Description
63:38	—	R	*Reserved*
37:36	fcc3	RW	Floating-point condition code (set 3). One of four sets of 2-bit floating-point condition codes, which are modified by the FCMP{E} (and LD{X}FSR) instructions. The FBfcc, FMOVcc, and MOVcc instructions use one of these condition code sets to determine conditional control transfers and conditional register moves.
35:34	fcc2	RW	Floating-point condition code (set 2). See fcc3.
33:32	fcc1	RW	Floating-point condition code (set 1). See fcc3.
31:30	rd	RW	IEEE Std. 754-1985 rounding direction. Rounding modes are shown below.
29:28	—	R	*Reserved*
27:23	tem	RW	5-bit trap enable mask for the IEEE-754 floating-point exceptions. If a floating-point operate instruction produces one or more exceptions, the corresponding cexc/aexc bits are set and an *fp_exception_ieee_754* (with FSR.ftt = 1, IEEE_754_exception) exception is generated.
22	ns	R	Nonstandard floating-point results. Always 0: UltraSPARC T1 produces IEEE-754 compatible results.
21:20	—	R	*Reserved*
19:17	ver	R	FPU version number. Identifies a particular implementation of the UltraSPARC T1 FPU architecture.
16:14	ftt	R	Floating-point trap type. The 3-bit floating point trap type field is set whenever an floating-point instruction causes the *fp_exception_ieee_754* or *fp_exception_other* traps. Trap types are listed in TABLE 12-4, below.
13:	qne	R	Floating-point deferred-trap queue (FQ) not empty. Not used, because UltraSPARC T1 implements precise floating-point exceptions.
12	—	R	*Reserved*

rd	Round Toward
0	Nearest (even if tie)
1	0
2	+∞
3	−∞

TABLE 12-2 Floating-Point Status Register Format *(Continued)*

Bits	Field	RW	Description
11:10	fcc0	RW	Floating-point condition code (set 0). See fcc3. Note: fcc0 is the same as fcc in SPARC V8.
9:5	aexc	RW	5-bit accrued exception field. Accumulates IEEE 754 exceptions while floating-point exception traps are disabled (that is, FSR.tem = 0).
4:0	cexc	RW	5-bit current exception field indicates the most recently generated IEEE 754 exceptions.

Note | fcc0 is the same as the fcc in SPARC V8.

TABLE 12-3 Floating-Point Rounding Modes

rd	Round Toward
0	Nearest (even if tie)
1	0
2	$+\infty$
3	$-\infty$

TABLE 12-4 Floating-Point Trap Type Values

ftt	Floating-Point Trap Type	Trap Signaled
0	None	—
1	IEEE_754_exception	*fp_exception_ieee_754*
2	unfinished_FPop	—
3	unimplemented_FPop	*fp_exception_other*
4	sequence_error	—
5	hardware_error	—
6	invalid_fp_register	—
7	reserved	—

Notes | (1) UltraSPARC T1 neither detects nor generates the unfinished_FPop, sequence_error, hardware_error or invalid_fp_register trap types directly in hardware.

(2) UltraSPARC T1 does not contain an FQ. An attempt to read the FQ with a RDPR instruction causes an *illegal_instruction* trap.

12.4 SPARC V9 Memory-Related Operations

12.4.1 Load/Store Alternate Address Space (Impl. Dep. #5, 29, 30)

Supported ASI accesses are listed in *Alternate Address Spaces* on page 41.

12.4.2 Read/Write ASR (Impl. Dep. #6, 7, 8, 9, 47, 48)

Supported ASRs are discussed in *Ancillary State Registers (ASRs)* on page 7.

12.4.3 FLUSH and Self-Modifying Code (Impl. Dep. #122)

FLUSH is needed to synchronize code and data spaces after code space is modified during program execution. FLUSH is described in *Supported Memory Models* on page 36. On UltraSPARC T1, the FLUSH effective address is ignored, and as a result, FLUSH can not cause a *data_access_exception* trap.

Note | SPARC V9 specifies that the FLUSH instruction has no latency on the issuing strand. In other words, a store to instruction space prior to the FLUSH instruction is visible immediately after the completion of FLUSH. MEMBAR #`StoreStore` is required to ensure proper ordering in multiprocessing system when the memory model is not TSO. When a MEMBAR #`StoreStore`, FLUSH sequence is performed, UltraSPARC T1 guarantees that earlier code modifications will be visible across the whole system.

12.4.4 PREFETCH{A} (Impl. Dep. #103, 117)

For UltraSPARC T1, PREFETCH{A} instructions follow TABLE 12-5 based on the fcn value. All prefetches in UltraSPARC T1 are weak (on an MMU miss or when the MMU is bypassed the prefetch is dropped). The only trap that a prefetch can generate on UltraSPARC T1 is *illegal_instruction* (for fcn = 5_{16}–F_{16}).

TABLE 12-5 PREFETCH{A} Variants

fcn	Prefetch Function	Action
0_{16}	Weak prefetch for several reads	Weak prefetch into Level 2 cache
1_{16}	Weak prefetch for one read	
2_{16}	Weak prefetch for several writes	
3_{16}	Weak prefetch for one write	
4_{16}	Prefetch Page	No operation
5_{16}–F_{16}	-	*Illegal_instruction* trap.
10_{16}	Invalidate read-once prefetch	Weak prefetch into Level 2 cache
11_{16}	Prefetch for read to nearest unified cache	Weak prefetch into Level 2 cache
12_{16}–13_{16}	Strong prefetches	Weak prefetch into Level 2 cache
14_{16}	Strong prefetch for several reads	Weak prefetch into Level 2 cache
15_{16}	Strong prefetch for one read	
16_{16}	Strong prefetch for several writes	
17_{16}	Strong prefetch for one write	
18_{16}-$1F_{16}$	—	No operation

PREFETCHA is legal for all implemented ASIs in UltraSPARC T1 and will prefetch into the Level 2 cache from memory using the context listed in TABLE 12-6. Prefetching is done regardless of privilege level (for example, user mode can use ASI 10_{16} to prefetch into the L2 cache).

TABLE 12-6 PREFETCH{A} ASIs

Context	ASIs (hexadecimal)
Primary	10, 16, 18, 1E, 22, 2A, 80, 82, 88, 8A, C0, C2, C4, C8, CA, CC, D0, D2, D8, DA, E0, E2, EA, F0, F8
Secondary	11, 17, 19, 1F, 23, 2B, 81, 83, 89, 8B, C1, C3, C5, C9, CB, CD, D1, D3, D9, DB, E1, E3, EB, F1, F9
Nucleus	04, 0C, 14, 15, 1C, 1D, 20, 21, 24, 25, 26, 27, 2C, 2E, 2F, 31, 32, 33, 35, 36, 37, 39, 3A, 3B, 3D, 3E, 3F, 40, 42, 43, 44, 45, 46, 47, 4B, 4C, 4D, 4F, 50, 51, 52, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 5E, 5F, 60, 66, 67, 72, 73, 74

Implementation Note | Although it would have been desirable to treat PREFETCHA to restricted ASIs by underprivileged code as NOPs, PREFETCH only moves data between main memory and the L2 cache, so UltraSPARC T1's implementation causes no security issues.

12.4.5 Instruction Prefetch

UltraSPARC T1 does not implement an instruction prefetch. No prefetching is performed from the effective address of the BPN instruction.

12.4.6 LDTW/STTW Handling (Impl. Dep. #107, 108)

LDTW and STTW instructions are directly executed in hardware.

Note | LDTW/STTW were deprecated in SPARC V9. In UltraSPARC T1, it is more efficient to use LDX/STX for accessing 64-bit data. LDTW/STTW take longer to execute than two 32-/64-bit loads/stores.

12.4.7 Floating-Point *mem_address_not_aligned* (Impl. Dep. #109, 110, 111, 112)

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

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

12.4.8 Supported Memory Models (Impl. Dep. #113, 121)

UltraSPARC T1 supports only the TSO memory model, although certain specific operations such as block loads and stores operate under the RMO memory model. See *Supported Memory Models* on page 36.

12.4.9 Implicit ASI when TL > 0 (Impl. Dep. #124)

UltraSPARC T1 matches all UltraSPARC Architecture implementations, and makes the implicit ASI for instruction fetching `ASI_NUCLEUS` when TL > 0, while the implicit ASI for loads and stores when TL > 0 is `ASI_NUCLEUS` if PSTATE.cle = 0 or `ASI_NUCLEUS_LITTLE` if PSTATE.cle = 1.

Compatibility Note With an implicit ASI for instruction fetching of ASI_NUCLEUS, if software was to set the strand in a state where PSTATE.priv = 0 but TL > 0, an instruction fetch will generate an *instruction_access_exception*, because user-level code is accessing ASI_NUCLEUS. UltraSPARC I/II overrides this *instruction_access_exception* and allows instruction fetching when PSTATE.priv = 0 and TL > 0. UltraSPARC T1 is compatible with UltraSPARC I/II and does the same override of *instruction_access_exception* when PSTATE.priv = 0 and TL > 0.

12.5 Non-SPARC V9 Extensions

12.5.1 Cache Subsystem

UltraSPARC T1 contains one or more levels of cache. The cache subsystem architecture is described in Appendix F, *Caches and Cache Coherency*.

12.5.2 Block Memory Operations

UltraSPARC T1 supports 64-byte block memory operations utilizing a block of eight double-precision floating point registers as a temporary buffer. See *Block Load and Store Instructions* on page 18.

12.5.3 Partial Stores

UltraSPARC T1 does not support 8-/16-/32-bit partial stores to memory.

12.5.4 Short Floating-Point Loads and Stores

UltraSPARC T1 does not supports 8-/16-bit loads and stores to the floating-point registers.

12.5.5 Interrupt Vector Handling

CPUs and I/O devices can interrupt a selected CPU by assembling and sending an interrupt packet. This allows hardware interrupts and cross calls to have the same hardware mechanism and to share a common software interface for processing. Interrupt vectors are described in Chapter 7, *Interrupt Handling*.

12.5.6 Power-Down Support

UltraSPARC T1 supports the ability to power down virtual processors and I/O devices to reduce power requirements during idle periods.

12.5.7 UltraSPARC T1 Instruction Set Extensions (Impl. Dep. #106)

The UltraSPARC T1 CPU supports a subset of the VIS 1.0 and 2.0 instructions; see *UltraSPARC Architecture 2005 Instructions Not Directly Implemented by UltraSPARC T1 Hardware* on page 13.

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

12.5.8 Performance Instrumentation

UltraSPARC T1 performance instrumentation is described in *Performance Control Register* on page 49 and *SPARC Performance Instrumentation Counter* on page 51.

APPENDIX **A**

Assembly Language Syntax

The assembly language syntax used in this document follows that described in the "Assembly Language Syntax" appendix of the *UltraSPARC Architecture 2005* specification.

APPENDIX **B**

Programming Guidelines

B.1 Multithreading

In UltraSPARC T1, execution is switched in round-robin fashion every cycle among the strands that are ready to issue another instruction. Context switching is built into the UltraSPARC T1 pipeline and takes place during the SWITCH stage, thus contexts are switched each cycle with no pipeline stall penalty.

The following instructions change a strand from a ready-to-issue state to a not-ready-to-issue state, until hardware determines that their input/execution requirements can be satisfied:

- All branches (including CALL, JMPL, etc.)
- All VIS instructions
- All floating point (FPops)
- All WRPR, WR
- All RDPR, RD
- SAVE(D), RESTORE(D), RETURN, FLUSHW (all register management)
- All MUL and DIV
- MULSCC
- MEMBAR `#Sync`, MEMBAR `#StoreLoad`, MEMBAR `#MemIssue`
- FLUSH
- All loads
- All floating-point memory operations
- All memory operations to alternate space
- All atomics load-store operations
- Prefetch

B.2 Pipeline Strand Flush

The front end of the UltraSPARC T1 pipeline prevents instructions from being issued to the rest of the pipeline unless there is a high probability (for most instructions, a probability of 1.0) of the instruction having all its input dependencies satisfied. For certain instructions, the input dependencies cannot be determined by the front end, and the instruction (and any subsequent instructions issued from that strand) need to be flushed from the pipeline and replayed. TABLE B-1 lists instructions that may end up causing a strand flush.

TABLE B-1 Pipeline Strand Flush Events

Event	Strand Flush Description
Loads	The strand will be flushed if the load encounters a cache miss while executing with STRAND_STS_REG.spec_en = 1.
multiply/divide/ floating-point operate	Resource contention can cause a strand flush.
store buffer full	The strand will be flushed until space is available in the store buffer.
trap	Instruction and any subsequent instructions in the pipeline from that strand are flushed, and fetching restarts at the trap vector.

B.3 Instruction Latencies

TABLE B-2 lists the single-strand instruction latencies for UltraSPARC T1. When multiple strands are executing, much of the additional latency for multicycle instructions will be overlapped with execution of the additional strands.

In this table, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 5-1 on page 13.

TABLE B-2 Instruction Latencies *(1 of 14)*

Instruction	Latency	Comments
ADD	1	
ADDc	1	
ADDCC	1	
ADDNcc	1	

TABLE B-2 Instruction Latencies *(2 of 14)*

Instruction	Latency	Comments
AND	1	
ANDcc	1	
ANDN	1	
ANDNcc	1	
BA	3?	
BA_A	4	
BA_A_PN	4?	
BA_PN	3	
BA_XCC	3	
BA_XCC_A	4?	
BA_XCC_A_PN	4	
BA_XCC_PN	3?	
BCC	3	
BCC_A	3	
BCC_A_PN	4	
BCC_PN	3	
BCC_XCC	3	
BCC_XCC_A	3	
BCC_XCC_A_PN	4	
BCC_XCC_PN	3?	
BCS	3	
BCS_A	4	
BCS_A_PN	4?	
BCS_PN	3	
BCS_XCC	3	
BCS_XCC_A	4	
BCS_XCC_A_PN	4	
BCS_XCC_PN	4?	
BE	3	
BE_A	4?	

TABLE B-2 Instruction Latencies *(3 of 14)*

Instruction	Latency	Comments
BE_A_PN	4	
BE_PN	3	
BE_XCC	3	
BE_XCC_A	4	
BE_XCC_A_PN	4	
BE_XCC_PN	3	
BG	3	
BG_A	3	
BG_A_PN	4	
BG_PN	3	
BG_XCC	3	
BG_XCC_A	3	
BG_XCC_A_PN	3?	
BG_XCC_PN	3	
BGE	3	
BGE_A	3	
BGE_A_PN	4?	
BGE_PN	3	
BGE_XCC	3	
BGE_XCC_A	4	
BGE_XCC_A_PN	4?	
BGE_XCC_PN	3	
BGU	3	
BGU_A	3	
BGU_A_PN	3?	
BGU_PN	3	
BGU_XCC	3	
BGU_XCC_A	4	
BGU_XCC_A_PN	4	
BGU_XCC_PN	3	

TABLE B-2 Instruction Latencies *(4 of 14)*

Instruction	Latency	Comments
BL	3	
BL_A	3	
BL_A_PN	3?	
BL_PN	3	
BL_XCC	3	
BL_XCC_A	4?	
BL_XCC_A_PN	4	
BL_XCC_PN	3	
BLE	3	
BLE_A	3	
BLE_A_PN	3?	
BLE_PN	3	
BLE_XCC	3	
BLE_XCC_A	4	
BLE_XCC_A_PN	4?	
B LE_XCC_PN	3	
BLEU	3	
BLEU_A	4?	
BLEU_A_PN	4	
BLEU_PN	3?	
BLEU_XCC	3	
BLEU_XCC_A	4	
BLEU_XCC_A_PN	4	
BLEU_XCC_PN	3	
BN	3	
BN_A	4	
BN_A_PN	4	
BN_PN	3	
BN_XCC	3	
BN_XCC_A	4?	

TABLE B-2 Instruction Latencies *(5 of 14)*

Instruction	Latency	Comments
BN_XCC_A_PN	4	
BN_XCC_PN	3	
BNE	3	
BNE_A	3	
BNE_A_PN	3?	
BNE_PN	3	
BNE_XCC	3?	
BNE_XCC_A	4	
BNE_XCC_A_PN	4	
BNE_XCC_PN	3?	
BNEG	3	
BNEG_A	4	
BNEG_A_PN	4	
BNEG_PN	3	
BNEG_XCC	3	
BNEG_XCC_A	4	
BNEG_XCC_A_PN	4	
BNEG_XCC_PN	3	
BPOS	3	
BPOS_A	4?	
BPOS_A_PN	4	
BPOS_PN	3	
BPOS_XCC	3	
BPOS_XCC_A	4?	
BPOS_XCC_A_PN	4	
BPOS_XCC_PN	3	
BRGEZ	3	
BRGEZ_A	4?	
BRGEZ_A_PN	4	
BRGEZ_PN	3	

TABLE B-2 Instruction Latencies *(6 of 14)*

Instruction	Latency	Comments
BRGZ	3	
BRGZ_A	4?	
BRGZ_A_PN	4	
BRGZ_PN	3	
BRLEZ	3	
BRLEZ_A	4	
BRLEZ_A_PN	4	
BRLEZ_PN	3	
BRLZ	3	
BRLZ_A	3	
BRLZ_A_PN	3?	
BRLZ_PN	3	
BRNZ	3	
BRNZ_A	4	
BRNZ_A_PN	4	
BRNZ_PN	3	
BRZ	3	
BRZ_A	4	
BRZ_A_PN	4	
BRZ_PN	3	
BVC	3	
BVC_A	4?	
BVC_A_PN	4	
BVC_PN	3	
BVC_XCC	3	
BVC_XCC_A	4	
BVC_XCC_A_PN	4?	
BVC_XCC_PN	3	
BVS	3	
BVS_A	4	

TABLE B-2 Instruction Latencies *(7 of 14)*

Instruction	Latency	Comments
BVS_A_PN	4	
BVS_PN	3	
BVS_XCC	3	
BVS_XCC_A	4	
BVS_XCC_A_PN	4	
BVS_XCC_PN	3	
CASA$^{P_{ASI}}$	39	performed in L2
CASXA$^{P_{ASI}}$	39	performed in L2
FABSd	8	
FABSs	21	
FADDd	26	
FADDs	26	
FBA	3	
FBA_A	4?	
FBA_A_PN	4	
FBA_PN	3	
FBE	3	
FBE_A	4	
FBE_A_PN	4	
FBE_PN	3	
FBG	3	
FBG_A	4	
FBG_A_PN	4?	
FBG_PN	3	
FBGE	3	
FBGE_A	4	
FBGE_A_PN	4	
FBGE_PN	3	
FBL	3	
FBL_A	4?	

TABLE B-2 Instruction Latencies *(8 of 14)*

Instruction	Latency	Comments
FBL_A_PN	4	
FBL_PN	3	
FBLE	3	
FBLE_A	4	
FBLE_A_PN	4	
FBLE_PN	3	
FBLG	3	
FBLG_A	4	
FBLG_A_PN	4	
FBLG_PN	3	
FBN	3?	
FBN_A	4	
FBN_A_PN	4	
FBN_PN	3	
FBNE	3	
FBNE_A	4	
FBNE_A_PN	4?	
FBNE_PN	3	
FBUE	3	
FBUE_A	4	
FBUE_A_PN	4	
FBUE_PN	3	
FBUG	3	
FBUG_A	4	
FBUG_A_PN	4	
FBUG_PN	3	
FBUGE	3	
FBUGE_A	4	
FBUGE_A_PN	4	
FBUGE_PN	3	

TABLE B-2 Instruction Latencies *(9 of 14)*

Instruction	Latency	Comments
FBUL	3	
FBUL_A	4	
FBUL_A_PN	4	
FBUL_PN	3	
FBULE	3	
FBULE_A	4	
FBULE_A_PN	4	
FBULE_PN	3	
FDIVd	83	
FDIVs	54	
FdTOi	25	
FdTOs	25	
FdTOx	25	
FiTOd	25	
FiTOs	26	
FMOVd	8	
FMOVDA		
FMOVDE	8	
FMOVDG	8	
FMOVDGE	8	
FMOVDL	8	
FMOVDLE	8	
FMOVDLG	8	
FMOVDN	8	
FMOVDNE	8	
FMOVDO	8	
FMOVDU	8	
FMOVDUE	8	
FMOVDUG	8	
FMOVDUGE	8	

TABLE B-2 Instruction Latencies *(10 of 14)*

Instruction	Latency	Comments
FMOVDUL	8	
FMOVDULE	8	
FMOVs	8	
FMOVSA	8	
FMOVSE	8?	
FMOVSG	8?	
FMOVSGE	8?	
FMOVSL	8?	
FMOVSLE	8?	
FMOVSLG	8?	
FMOVSN	8	
FMOVSNE	8	
FMOVSO	8?	
FMOVSU	8	
FMOVSUE	8	
FMOVSUG	8	
FMOVSUGE	8	
FMOVSUL	8	
FMOVSULE	8	
FMULd	29	
FMULs	29	
FNEGd	8	
FNEGs	8	
FsMULd	29	
FsTOd	25	
FsTOi	25	
FsTOx	25	
FSUBd	26	
FSUBs	26	
FxTOd	26	

TABLE B-2 Instruction Latencies *(11 of 14)*

Instruction	Latency	Comments
FxTOs	26	
LD_FP	9	
LDFSR	9	
LDD_FP	9?	
LDD_FPD	9	
LDDFA$^{P_{ASI}}$	9	
LDSB	22	performed in L2
LDSBA	21	performed in L2
LDSH	3	
LDSHA$^{P_{ASI}}$	3	
LDSTUB	37	performed in L2
LDSTUBA$^{P_{ASI}}$	37	performed in L2
LDSW	3	
LDSWA	3	
LDUB	3	
LDUBA	3	
LDUH	3	
LDUHA$^{P_{ASI}}$	3	
LDUW	3	
LDUWA$^{P_{ASI}}$	3	
LDX	3	
LDX_FSR	27	
LDXA$^{P_{ASI}}$	3	
MOVA	1	
MOVA_FCC	1	
MOVCC	1	
MOVCS	1	
MOVE	1	
MOVE_FCC	1	
MOVG	1?	

TABLE B-2 Instruction Latencies *(12 of 14)*

Instruction	Latency	Comments
MOVG_FCC	1	
MOVGE	1	
MOVGE_FCC	1	
MOVGU	1?	
MOVL	1	
MOVL_FCC	1	
MOVLE	1	
MOVLE_FCC	1	
MOVLEU	1	
MOVLG_FCC	1	
MOVN	1	
MOVN_FCC	1	
MOVNE	1	
MOVNE_FCC	1	
MOVNEG	1?	
MOVO_FCC	1	
MOVPOS	1	
MOVRE	1	
MOVRGEZ	1	
MOVRGZ	1	
MOVRLEZ	1	
MOVRLZ	1	
MOVRNE	1	
MOVU_FCC	1	
MOVUE_FCC	1	
MOVUG_FCC	1	
MOVUGE_FCC	1	
MOVUL_FCC	1	
MOVULE_FCC	1	
MOVVC	1	

TABLE B-2 Instruction Latencies *(13 of 14)*

Instruction	Latency	Comments
MOVVS	1	
MULSCC	7	
MULX	11	
OR	1	
ORcc	1	
ORN	1	
ORNcc	1	
RD_CCR	4	
RDASI	4	
RD_FPRS	4	
RD_Y	4	
SDIV[D]	72	
SDIVcc[D]	72	
SDIVX	72	
SETHI	1	
SLL	1	
SLLX	1	
SMUL[D]	11	
SMULcc[D]	11	
SRA	1?	
SRAX	1	
SRL	1	
SRLX	1	
STFA[PASI]	8	
ST_FSR	8	
STFA[PASI]	8	
STB	1	
STBA	4	
STDF	8	
STDA_FP	8?	

TABLE B-2 Instruction Latencies *(14 of 14)*

Instruction	Latency	Comments
STDA_FP_ASI	8	
STH	1	
STHA	4	
STW	1?	
STWA	1?	
STX	1	
STX_FSR	8	
STXA	1-? (4-?)?	varies, depending on ASI
SUB	1	
SUBC	1	
SUBcc	1	
SUBCcc	1	
SWAP^{D}	49	performed in L2
SWAPA$^{D,\ P_{ASI}}$	37	performed in L2
TADDcc	1?	
TADDccTV^{D}	1	
TSUBcc	1	
TSUBccTV^{D}	1	
UDIV^{D}	72	
UDIVcc^{D}	72?	
UDIVX	72	
UMUL^{D}	11	
UMULcc^{D}	11	
WR_CCR	9	
WRASI	9	
WR_FPRS	9	
XNOR	1	
XNORcc	1	
XOR	1	
XORcc	1	

B.4 Grouping Rules

Each physical cores in UltraSPARC T1 are single-issue, so there are no grouping rules for UltraSPARC T1.

B.5 Floating-Point Operations

UltraSPARC T1 supports hardware floating-point operations, but since one floating-point unit (FPU) is shared among 8 physical cores (32 strands), there are limitations on dispatch of floating-point instructions to the FPU. Each physical processor core (four strands) can have a single floating-point instruction outstanding at any given time. For the purpose of this restriction, floating-point instructions include floating-point operations, VIS floating-point operations, floating-point loads and stores, and block loads and stores.

B.6 Synchronization

UltraSPARC T1 has two varieties of instructions for synchronization: memory barriers and flush. The following memory barrier instructions ensure that any load, store, or atomic memory operation issued after it take effect after all memory operations issued before it:

- MEMBAR with mmask{1} = 1 (MEMBAR #StoreLoad)
- MEMBAR with cmask{1} = 1 (MEMBAR #MemIssue)
- MEMBAR with cmask{2} = 1 MEMBAR #Sync)

All other types of membar instructions are treated as NOPs, since they are implied by the TSO memory ordering protocol followed by UltraSPARC T1.

However, the memory barriers do not guarantee that the instruction caches on UltraSPARC T1 have become consistent with the preceding memory operations. A FLUSH instruction guarantees that in addition to the preceding memory operations taking effect in the global memory system, all the instruction caches on UltraSPARC T1 are consistent with these operations. It also ensures that the instruction fetch buffer for the strand issuing the flush has become consistent with the preceding memory operations.

Thus, when one strand is modifying the instructions of another, the “producer” strand should

1. Complete all necessary modifications
2. Issue a FLUSH
3. Signal completion to the “consumer” strand

   Completion may be signalled by a store/atomic instruction which modifies a predetermined location, or by issuing an interrupt to the consumer strand.

The consumer strand at this point should make sure that its instruction fetch buffer (of size 2 entries) becomes consistent with the global memory system. This can be done by either

1. Issuing a branch to the modified location; or
2. Issuing a flush instruction; or
3. Waiting for a two-instruction gap, to allow the two instructions in the fetch buffer to drain.

In the case of a branch, the delay slot is not guaranteed to be consistent with global memory. The branch is a better option than flush for high performance.

Note that the HALT instruction is not meant to be a synchronization instruction and should *not* be used as such. For example, the following code, which uses halt to make sure func_A executes before func_B, may cause T0 to hang:

T0	T1
`st X`	`ld X`
`halt`	`do_func_A`
`do_func_B`	`intr T0`

APPENDIX **C**

Opcode Maps

This appendix contains the UltraSPARC T1 instruction opcode maps.

Opcodes marked with a dash (—) are reserved; an attempt to execute a reserved opcode causes a trap unless the opcode is an implementation-specific extension to the instruction set.

In this appendix, certain opcodes are marked with mnemonic superscripts. These superscripts and their meanings are defined in TABLE 5-1 on page 13.

In the tables in this appendix, *reserved* (—) and shaded entries indicate opcodes that are not implemented in the UltraSPARC T1 processor.

Shading	Meaning
	An attempt to execute opcode will cause an *illegal_instruction* exception.
	An attempt to execute opcode will cause an *fp_exception_other* exception with FSR.ftt = 3 (unimplemented_FPop).

TABLE C-1 op{1:0}

***op* {1:0}**			
0	**1**	**2**	**3**
Branches and SETHI *See* TABLE C-2.	CALL	Arithmetic and Miscellaneous *See* TABLE C-3	Loads/Stores *See* TABLE C-4

TABLE C-2 op2{2:0} (op = 0)

***op2* {2:0}**							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
ILLTRAP	BPcc – *See* TABLE C-7	Bicc[D] – *See* TABLE C-7	BPr – *See* TABLE C-8	SETHI NOP[†]	FBPfcc – *See* TABLE C-7	FBfcc[D] – *See* TABLE C-7	—

[†] rd = 0, imm22 = 0

The ILLTRAP and *reserved* (—) encodings generate an *illegal_instruction* trap.

TABLE C-3 op3{5:0} (op = 2) *(1 of 3)*

		op3 {5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**0**	ADD	ADDcc	TADDcc	WRY[D] (rd = 0) — (rd=1) WRCCR (rd = 2) WRASI (rd = 3) — (rd = 4, 5) — (rd = 5, rd = 0, rd = 1) WRFPRS (rd = 6) — (rd = 7–14) — (rd = 15 and (rs1 > 0 or i = 0)) WRPCR[P] (rd = 16, rd = 0) — (rd =16, rd = 1) WRPIC (rd = 17, rd = 0) — (rd = 17, rd = 1) — (rd = 18) WRGSR (rd = 19) WRSOFTINT_SET[P] (rd = 0) WRSOFTINT_CLR[P] (rd = 21) WRSOFTINT[P] (rd = 22) WRTICK_CMPR[P] (rd = 23) WRSTICK_CMPR (rd = 25) WR %asr26 (rd = 26, rd = 1) — (rd = 26, rd = 0) — (rd=27–31))
	1	AND	ANDcc	TSUBcc	SAVED[P] (fcn = 0), RESTORED[P] (fcn = 1) — (fcn > 1)
	2	OR	ORcc	TADDccTV[D]	WRPR[P] — (rd = 15, 17–31)
	3	XOR	XORcc	TSUBccTV[D]	— (rs1 = 2, 4, 7–30)
	4	SUB	SUBcc	MULScc[D]	FPop1 – *See* TABLE C-5
	5	ANDN	ANDNcc	SLL (*x* = 0), SLLX (*x* = 1)	FPop2 – *See* TABLE C-6
	6	ORN	ORNcc	SRL (*x* = 0), SRLX (*x* = 1)	IMPDEP1 (VIS) – *See* TABLE C-12
	7	XNOR	XNORcc	SRA (*x* = 0), SRAX (*x* = 1)	IMPDEP2

TABLE C-3 op3{5:0} (op = 2) *(2 of 3)*

		op3 {5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**8**	ADDC	ADDCcc	RDY^{D} (rs1 = 0, i = 0)) — (rs1 = 0, i = 1) — (rs1 = 1) RDCCR (rs1 = 2, i = 0)) — (rs1 = 2, i = 1) RDASI (rs1 = 3, i = 0) — (rs1 = 3, i = 1) RDTICK$^{P_{npt}}$ (rs1 = 4, i = 0) — (rs1 = 4, i = 1) RDPC (rs1 = 5, i = 0) — (rs1 = 5, i = 1) RDFPRS (rs1 = 6, i = 0) — (rs1 = 6, i = 1) — (rs1 = 7–14) MEMBAR (rs1 = 15, rd = 0, i = 1) STBAR^{D} (rs1 = 15, rd = 0, i = 0) — (rs1 = 15, rd > 0) RDPCR^{P} (rs1 = 16) RDPIC (rs1 = 17) — (rs1 = 18) RDGSR (rs1 = 19, i = 0) — (rs1 = 19, i = 1) — (rs1 = 20, 21) RDSOFTINT^{P} (rs1 = 22, i = 0) — (rs1 = 22, i = 1) RDTICK_CMPR^{P} (rs1 = 23, i = 0) — (rs1 = 23, i = 1) RDSTICK^{P} (rs1 = 24, i = 0) — (rs1 = 24, i = 1) RDSTICK_CMPR^{P} (rs1 = 25, i = 0) — (rs1 = 25, i = 1) rd %asr26 (rs1 = 26) — (rs1 = 27 - 31)	JMPL

TABLE C-3 op3{5:0} (op = 2) *(3 of 3)*

		op3 {5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**9**	MULX	—	— (rs1 = 2, 4, 7 - 30)	RETURN
	A	UMUL[D]	UMULcc[D]	RDPR[P] — (rs1 = 15, 17 - 30)	Tcc {(i = 0 and inst{10:5} = 0) or i = 1 and inst{10:8} = 0)}–*See* TABLE C-7 *and* TABLE C-11. — {((i = 0 and inst{10:5} > 0) or i = 1 and inst{10:8} > 0)}
	B	SMUL[D]	SMULcc[D]	FLUSHW	FLUSH
	C	SUBC	SUBCcc	MOVcc *See* TABLE C-9	SAVE
	D	UDIVX	—	SDIVX	RESTORE
	E	UDIV[D]	UDIVcc[D]	POPC (rs1 = 0) — (rs1 > 0)	DONE[P] (fcn = 0) RETRY[P] (fcn = 1) — (fcn > 1)
	F	SDIV[D]	SDIVcc[D]	MOVr *See* TABLE C-8	—

Shaded and the reserved (—) opcodes cause an illegal_instruction trap.

TABLE C-4 op3{5:0} (op = 3)

		op3{5:4}			
		0	**1**	**2**	**3**
op3 {3:0}	**0**	LDUW	LDUWA$^{P_{ASI}}$	LDF	LDFA$^{P_{ASI}}$
	1	LDUB	LDUBA$^{P_{ASI}}$	LDFSR^{D}, LDXFSR — (rd > 1)	—
	2	LDUH	LDUHA$^{P_{ASI}}$	LDQF	LDQFA$^{P_{ASI}}$
	3	LDD^{D} — (rd odd)	LDDA$^{D, P_{ASI}}$ — (rd odd)	LDDF	LDDFA$^{P_{ASI}}$ See 8.6.4 XREF
	4	STW	STWA$^{P_{ASI}}$	STF	STFA$^{P_{ASI}}$
	5	STB	STBA$^{P_{ASI}}$	STFSR^{D}, STXFSR — (rd > 1)	—
	6	STH	STHA$^{P_{ASI}}$	STQF	STQFA$^{P_{ASI}}$
	7	STD^{D} — (rd odd)	STDA$^{P_{ASI}}$ — (rd odd)	STDF	STDFA$^{P_{ASI}}$ See 8.6.4 XREF
	8	LDSW	LDSWA$^{P_{ASI}}$	—	—
	9	LDSB	LDSBA$^{P_{ASI}}$	—	—
	A	LDSH	LDSHA$^{P_{ASI}}$	—	—
	B	LDX	LDXA$^{P_{ASI}}$	—	—
	C	—	—	—	CASA$^{P_{ASI}}$
	D	LDSTUB	LDSTUBA$^{P_{ASI}}$	PREFETCH — (fcn = 5–15)	PREFETCHA$^{P_{ASI}}$ — (fcn = 5–15)
	E	STX	STXA$^{P_{ASI}}$	—	CASXA$^{P_{ASI}}$
	F	SWAP^{D}	SWAPA$^{D, P_{ASI}}$	—	—

LDQF, LDQFA, STQF, STQFA, and the *reserved* (—) opcodes cause an *illegal_instruction* trap.

TABLE C-5 opf{8:0} (op = 2, op3 = 34_{16} = FPop1)

	opf{2:0}							
opf{8:3}	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
00_{16}	—	FMOVs	FMOVd	FMOVq	—	FNEGs	FNEGd	FNEGq
01_{16}	—	FABSs	FABSd	FABSq	—	—	—	—
02_{16}	—	—	—	—	—	—	—	—
03_{16}	—	—	—	—	—	—	—	—
04_{16}	—	—	—	—	—	—	—	—
05_{16}	—	FSQRTs	FSQRTd	FSQRTq	—	—	—	—
06_{16}	—	—	—	—	—	—	—	—
07_{16}	—	—	—	—	—	—	—	—
08_{16}	—	FADDs	FADDd	FADDq	—	FSUBs	FSUBd	FSUBq
09_{16}	—	FMULs	FMULd	FMULq	—	FDIVs	FDIVd	FDIVq
$0A_{16}$	—	—	—	—	—	—	—	—
$0B_{16}$	—	—	—	—	—	—	—	—
$0C_{16}$	—	—	—	—	—	—	—	—
$0D_{16}$	—	FsMULd	—	—	—	—	FdMULq	—
$0E_{16}$	—	—	—	—	—	—	—	—
$0F_{16}$	—	—	—	—	—	—	—	—
10_{16}	—	FsTOx	FdTOx	FqTOx	FxTOs	—	—	—
11_{16}	FxTOd	—	—	—	FxTOq	—	—	—
12_{16}	—	—	—	—	—	—	—	—
13_{16}	—	—	—	—	—	—	—	—
14_{16}	—	—	—	—	—	—	—	—
15_{16}	—	—	—	—	—	—	—	—
16_{16}	—	—	—	—	—	—	—	—
17_{16}	—	—	—	—	—	—	—	—
18_{16}	—	—	—	—	FiTOs	—	FdTOs	FqTOs
19_{16}	FiTOd	FsTOd	—	FqTOd	FiTOq	FsTOq	FdTOq	—
$1A_{16}$	—	FsTOi	FdTOi	FqTOi	—	—	—	—
$1B_{16}$–$3F_{16}$	—	—	—	—	—	—	—	—

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with FSR.ftt = 3 (unimplemented_FPop).

TABLE C-6 opf{8:0} (op = 2, op3 = 35_{16} = FPop2)

	opf{3:0}								
opf{8:4}	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8–F**
00	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	†	†	†	—
01	—	—	—	—	—	—	—	—	—
02	—	—	—	—	—	FMOVsZ	FMOVdZ	FMOVqZ	—
03	—	—	—	—	—	—	—	—	—
04	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVsLEZ	FMOVdLEZ	FMOVqLEZ	—
05	—	FCMPs	FCMPd	FCMPq	—	FCMPEs	FCMPEd	FCMPEq	—
06	—	—	—	—	—	FMOVsLZ	FMOVdLZ	FMOVqLZ	—
07	—	—	—	—	—	—	—	—	—
08	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
09	—	—	—	—	—	—	—	—	—
0A	—	—	—	—	—	FMOVsNZ	FMOVdNZ	FMOVqNZ	—
0B	—	—	—	—	—	—	—	—	—
0C	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVsGZ	FMOVdGZ	FMOVqGZ	—
0D	—	—	—	—	—	—	—	—	—
0E	—	—	—	—	—	FMOVsGEZ	FMOVdGEZ	FMOVqGEZ	—
0F	—	—	—	—	—	—	—	—	—
10	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
11–17	—	—	—	—	—	—	—	—	—
18	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
19–1F	—	—	—	—	—	—	—	—	—

† Reserved variation of FMOVr

Shaded and *reserved* (—) opcodes cause an *fp_exception_other* trap with FSR.ftt = 3 (unimplemented_FPop).

TABLE C-7 cond{3:0}

		BPcc	Bicc[D]	FBPfcc	FBfcc[D]	Tcc
		op = 0 op2 = 1	op = 0 op2 = 2	op = 0 op2 = 5	op = 0 op2 = 6	op = 2 op3 = $3a_{16}$
cond {3:0}	0	BPN	BN[D]	FBPN	FBN[D]	TN
	1	BPE	BE[D]	FBPNE	FBNE[D]	TE
	2	BPLE	BLE[D]	FBPLG	FBLG[D]	TLE
	3	BPL	BL[D]	FBPUL	FBUL[D]	TL
	4	BPLEU	BLEU[D]	FBPL	FBL[D]	TLEU
	5	BPCS	BCS[D]	FBPUG	FBUG[D]	TCS
	6	BPNEG	BNEG[D]	FBPG	FBG[D]	TNEG
	7	BPVS	BVS[D]	FBPU	FBU[D]	TVS
	8	BPA	BA[D]	FBPA	FBA[D]	TA
	9	BPNE	BNE[D]	FBPE	FBE[D]	TNE
	A	BPG	BG[D]	FBPUE	FBUE[D]	TG
	B	BPGE	BGE[D]	FBPGE	FBGE[D]	TGE
	C	BPGU	BGU[D]	FBPUGE	FBUGE[D]	TGU
	D	BPCC	BCC[D]	FBPLE	FBLE[D]	TCC
	E	BPPOS	BPOS[D]	FBPULE	FBULE[D]	TPOS
	F	BPVC	BVC[D]	FBPO	FBO[D]	TVC

TABLE C-8 Encoding of rcond{2:0} Instruction Field

		BPr	MOVr	FMOVr
		op = 0 op2 = 3	op = 2 op3 = $2f_{16}$	op = 2 op3 = 35_{16}
rcond {2:0}	0	—	—	—
	1	BRZ	MOVRZ	FMOVRZ
	2	BRLEZ	MOVRLEZ	FMOVRLEZ
	3	BRLZ	MOVRLZ	FMOVRLZ
	4	—	—	—
	5	BRNZ	MOVRNZ	FMOVRNZ
	6	BRGZ	MOVRGZ	FMOVRGZ
	7	BRGEZ	MOVRGEZ	FMOVRGEZ

TABLE C-9 cc / opf_cc Fields (MOVcc and FMOVcc)

opf_cc			Condition Code Selected
cc2	cc1	cc0	
0	0	0	fcc0
0	0	1	fcc1
0	1	0	fcc2
0	1	1	fcc3
1	0	0	icc
1	0	1	—
1	1	0	xcc
1	1	1	—

TABLE C-10 cc Fields (FBPfcc, FCMP, and FCMPE)

cc1	cc0	Condition Code Selected
0	0	fcc0
0	1	fcc1
1	0	fcc2
1	1	fcc3

TABLE C-11 cc Fields (BPcc and Tcc)

cc1	cc0	Condition Code Selected
0	0	icc
0	1	—
1	0	xcc
1	1	—

TABLE C-12 VIS Opcodes op = 2, op3 = 36_{16} = IMPDEP1

		opf {3:0}							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
opf {8:4}	**00**	EDGE8	EDGE8N	EDGE8L	EDGE8LN	EDGE16	EDGE16N	EDGE16L	EDGE16LN
	01	ARRAY8		ARRAY16		ARRAY32			
	02	FCMPLE16		FCMPNE16		FCMPLE32		FCMPNE32	
	03		FMUL8X16		FMUL8X16AU		FMUL8X16AL	FMUL8SUX16	FMUL8ULX16
	04								
	05	FPADD16	FPADD16S	FPADD32	FPADD32S	FPSUB16	FPSUB16S	FPSUB32	FPSUB32S
	06	FZERO	FZEROS	FNOR	FNORS	FANDNOT2	FANDNOT2S	FNOT2	FNOT2S
	07	FAND	FANDS	FXNOR	FXNORS	FSRC1	FSRC1S	FORNOT2	FORNOT2S
	08	SHUTDOWN	SIAM						
	09.. 1F								

		opf {3:0}							
		8	**9**	**A**	**B**	**C**	**D**	**E**	**F**
opf {8:4}	**00**	EDGE32	EDGE32N	EDGE32L	EDGE32LN				
	01	ALIGN ADDRESS	BMASK	ALIGNADDR ESS_LITTLE					
	02	FCMPGT16		FCMPEQ16		FCMPGT32		FCMPEQ32	
	03	FMULD8SUX16	FMULD8ULX1 6	FPACK32	FPACK16		FPACKFIX	PDIST	
	04	FALIGNDATA			FPMERGE	BSHUFFLE	FEXPAND		
	05								
	06	FANDNOT1	FANDNOT1S	FNOT1	FNOT1S	FXOR	FXORS	FNAND	FNANDS
	07	FSRC2	FSRC2S	FORNOT1	FORNOT1S	FOR	FORS	FONE	FONES
	08								
	09.. 1F								

Note | An *illegal_instruction* exception is generated if the undefined or shaded opcodes in the IMPDEP1 space are used.

APPENDIX **D**

Instructions and Exceptions

The instructions supported by UltraSPARC T1 and the exceptions they generate are listed in the UltraSPARC Architecture 2005 specification.

APPENDIX **E**

IEEE 754 Floating Point Support

UltraSPARC T1 conforms to the SPARC V9 Appendix B (IEEE Std 754-1985 Requirements for SPARC-V9) recommendations.

Note | UltraSPARC T1 detects tininess before rounding.

E.1 Special Operand Handling

The UltraSPARC T1 FPU provides full hardware support for subnormal operands and results. Unlike UltraSPARC I/II and UltraSPARC III, UltraSPARC T1 will never generate an unfinished_FPop trap type. Also, unlike UltraSPARC I/II and UltraSPARC III, UltraSPARC T1 does not implement a nonstandard floating-point mode. The ns bit of the FSR is always read as 0, and writes to it are ignored.

The FPU generates +inf, –inf, +largest number, –largest number (depending on round mode) for overflow cases for multiply, divide, and add operations.

For higher-to-lower precision conversion instructions {FDTOS}:

- overflow, underflow, and inexact exceptions can be raised.
- overflow is treated the same way as an unrounded add result; depending on the round mode, we will either generate the properly signed infinity or largest number.
- underflow will produce a signed zero, smallest number, or subnormal result.

For conversion to integer instructions {F(s,d)TOi, F(s,d)TOx}: UltraSPARC T1 follows SPARC V9 appendix B.5, pg 246.

For NaN's: UltraSPARC T1 Follows SPARC V9 appendix B.2 (particularly Table 27) and B.5, pg 244-246.

- Please note that Appendix B applies to those instructions listed in IEEE 754 section 5: "All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the sqrt, find the remainder, round to integer in fp format, convert between different fp formats, convert

between fp and integer formats, convert binary<->decimal, and compare. Whether copying without change of format is considered an operation is an implementation option."

- The instructions involving copying/moving of fp data (FMOV, FABS, and FNEG) will follow earlier UltraSPARC implementations by doing the appropriate sign bit transformation but will not cause an invalid exception nor do a rs2 = SNaN to rd = QNaN transformation.
- Following UltraSPARC I/II implementations, all Fpops as defined in V9 will update cexc. All other instructions will leave cexc unchanged.
- Following SPARC V9 Manual 5.1.7.6, 5.1.7.8, 5.1.7.9, and figures in 5.1.7.10 Overflow Result is defined as:

  If the appropriate trap enable masks are not set (FSR.ofm = 0 *and* FSR.nxm = 0), then set aexc and cexc overflow and inexact flags: FSR.ofa = 1, FSR.nxa = 1, FSR.ofc = 1, FSR.nxc = 1. No trap is generated.

  If any or both of the appropriate trap enable masks are set (FSR.ofm = 1 *or* FSR.nxm = 1), then only an IEEE overflow trap is generated: FSR.ftt = 1. The particular cexc bit that is set diverges from UltraSPARC I/II to follow the SPARC V9 section 5.1.7.9 errata:

  If FSR.ofm = 0 *and* FSR.nxm = 1, then FSR.nxc = 1.

  If FSR.ofm = 1, independent of FSR.nxm, then FSR.ofc = 1 and FSR.nxc = 0.

- Following SPARC V9 Manual 5.1.7.6, 5.1.7.8, 5.1.7.9, and figures in 5.1.7.10 Underflow Result is defined as:

  If the appropriate trap enable masks are not set (FSR.ufm = 0 *and* FSR.nxm = 0), then set aexc and cexc underflow and inexact flags: FSR.ufa = 1, FSR.nxa = 1, FSR.ufc = 1, FSR.nxc = 1. No trap is generated.

  If any or both of the appropriate trap enable masks are set (FSR.ufm = 1 *or* FSR.nxm = 1), then only an IEEE underflow trap is generated: FSR.ftt = 1. The particular cexc bit that is set diverges from UltraSPARC I/II to follow the SPARC V9 section 5.1.7.9 errata:

  If FSR.ufm = 0 *and* FSR.nxm = 1, then FSR.nxc = 1.

  If FSR.ufm = 1, independent of FSR.nxm, then FSR.ufc = 1 and FSR.nxc = 0.

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

E.1.1 Infinity Arithmetic

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

E.1.1.1 One Infinity Operand Arithmetic

- Do not generate exceptions

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

Cases
+inf plus +num = +inf +inf plus -num = +inf -inf plus +num = -inf -inf plus -num = -inf
+inf minus +num = +inf +inf minus -num = +inf -inf minus +num = -inf -inf minus -num = -inf
+inf multiplied by +num = +inf +inf multiplied by -num = -inf -inf multiplied by +num = -inf -inf multiplied by -num = +inf
+inf divided by +num = +inf +inf divided by -num = -inf -inf divided by +num = -inf -inf divided by -num = +inf
+num divided by +inf = +0 +num divided by -inf = -0 -num divided by +inf = -0 -num divided by -inf = +0
fstod, fdtos (+inf) = +inf fstod, fdtos (-inf) = -inF
+inf divided by +0 = +inf +inf divided by -0 = -inf -inf divided by +0 = -inf -inf divided by -0 = +inf

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

Cases
Any arithmetic operation involving infinity as 1 operand and a QNaN as the other operand: SPARC V9 B.2.2 Table 27 (+/- inf) OPERATOR (QNaN2) = QNaN2 (QNaN1) OPERATOR (+/- inf) = QNaN1
Compares when other operand is not a NaN treat infinity just like a regular number: +inf = +inf, +inf > anything else; -inf = -inf, -inf < anything else. Affects following instructions: V9 fp compares (rs1 and/or rs2 could be +/- inf): * FCMPE * FCMP
Compares when other operand is a QNaN, SPARC V9 A.13, B.2.1; fcc value = unordered = 2'b11 fcmp(s/d) (+/- inf) with (QNaN2) - no invalid exception fcmp(s/d) (QNaN1) with (+/- inf) - no invalid exception

- Could generate exceptions

TABLE E-2 One Infinity Operations That Could Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) iF tem is cleared
SPARC V9 Appendix B.5[1]	IEEE_754 7.1	
F{s,d}TOi (+inf) = invalid	IEEE_754 invalid	2^{31}-1
F{s,d}TOx (+inf) = invalid	IEEE_754 invalid	2^{63}-1
F{s,d}TOi (-inf) = invalid	IEEE_754 invalid	-2^{31}
F{s,d}TOx (-inf) = invalid	IEEE_754 invalid	-2^{63}
SPARC V9 B.2.2	IEEE_754 7.1	(No NaN operand result)
+inf multiplied by +0 = invalid	IEEE_754 invalid	QNaN
+inf multiplied by -0 = invalid	IEEE_754 invalid	QNaN
-inf multiplied by +0 = invalid	IEEE_754 invalid	QNaN
-inf multiplied by -0 = invalid	IEEE_754 invalid	QNaN

TABLE E-2 One Infinity Operations That Could Generate Exceptions *(Continued)*

Cases	Possible Exception	Result (in addition to accrued exception) iF tem is cleared
SPARC V9 B.2.2 Table 27[2] Any arithmetic operation involving infinity as 1 operand and a SNaN as the other operand except copying/moving data	IEEE_754 7.1	(One operand, a SNaN)
(+/- inf) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (+/- inf)	IEEE_754 invalid	QSNaN1
SPARC V9 A.13, B.2.1[2] Any compare operation involving infinity as 1 operand and a SNaN as the other operand:	IEEE_754 7.1	
FCMP(s/d) (+/- inf) with (SNaN2)	IEEE_754 invalid	fcc value = unordered = 2′b11
FCMP(s/d) (SNaN1) with (+/- inf)	IEEE_754 invalid	fcc value = unordered = 2′b11
FCMPE(s/d) (+/- inf) with (SNaN2)	IEEE_754 invalid	fcc value = unordered = 2′b11
FCMPE(s/d) (SNaN1) with (+/- inf)	IEEE_754 invalid	fcc value = unordered = 2′b11
SPARC V9 A.13[2] Any compare & generate exception operation involving infinity as 1 operand and a QNaN as the other operand:	IEEE_754 7.1	
FCMPE(s/d) (+/- inf) with (QNaN2)	IEEE_754 invalid	fcc value = unordered = 2′b11
FCMPE(s/d) (QNaN1) with (+/- inf)	IEEE_754 invalid	fcc value = unordered = 2′b11

1. Similar invalid exceptions also included in SPARC V9 B.5 are generated when the source operand is a NaN(QNaN or SNaN) or a resulting number that cannot fit in 32b[64b] integer format: (large positive argument >= 2^{31}[2^{63}] or large negative argument <= -(2^{31} + 1)[-(2^{63}+1)]

2. Note that in the IEEE 754 standard, infinity is an exact number; so this exception could also applies to non-infinity operands as well. Also note that the invalid exception and SNaN to QNaN transformation does not apply to copying/moving fpops (fmov,fabs,fneg).

E.1.1.2 Two Infinity Operand Arithmetic

- Do not generate exceptions

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

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

- Could generate exceptions

TABLE E-4 Two Infinity Operations That Generate Exceptions

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

E.1.2 Zero Arithmetic

TABLE E-5 Zero Arithmetic Operations That Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 B.2.2 & 5.1.7.10.4	IEEE_754 7.1	(No NaN operand
+0 divided by +0 = invalid	IEEE_754 invalid	result)
+0 divided by -0 = invalid	IEEE_754 invalid	QNaN
-0 divided by +0 = invalid	IEEE_754 invalid	QNaN
-0 divided by -0 = invalid	IEEE_754 invalid	QNaN
		QNaN
SPARC V9 5.1.7.10.4	IEEE_754 7.2	
+num divided by +0 = divide by zero	IEEE_754 div_by_zero	+inf
+num divided by -0 = divide by zero	IEEE_754 div_by_zero	-inf
-num divided by +0 = divide by zero	IEEE_754 div_by_zero	-inf
-num divided by -0 = divide by zero	IEEE_754 div_by_zero	+inf
SPARC V9 B.2.2 Table 27[1] Any arithmetic operation involving zero as 1 operand and a SNaN as the other operand except copying/moving data	IEEE_754 7.1	(One operand, a SNaN)
(+/- 0) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (+/- 0)	IEEE_754 invalid	QSNaN1

1.In this context, 0 is again another exact number; so this exception could also applies to non-zero operands as well. Also note that the invalid exception and SNaN to QNaN transformation does not apply to copying/moving data instructions (FMOV, FABS, FNEG)

TABLE E-6 Interesting Zero Arithmetic Sign Result Case

Cases
+0 plus -0 = +0 for all round modes except round to -infinity where the result is -0.

E.1.3 NaN Arithmetic

- Do not generate exceptions

TABLE E-7 NaN Arithmetic Operations that do not generate exceptions

Cases
SPARC V9 B.2.1: Fp convert to wider NaN transformation FsTOd (QNaN2) = QNaN2 widened FsTOd(0x7fd10000) = 0x7ffa2000 8'h0 FsTOd(0xffd10000) = 0xfffa2000 8'h0
SPARC V9 B.2.1: Fp convert to narrower NaN transformation FdTOs (QNaN2) = QNaN2 narrowed FdTOs(0x7ffa2000 8'h0) = 0x7fd1000 FdTOs(0xfffa2000 8'h0) = 0xffd1000
SPARC V9 B.2.2 Table 27 Any non-compare arithmetic operations --result takes sign of QNaN pass through operand.
(+/- num) OPERATOR (QNaN2) = QNaN2
(QNaN1) OPERATOR (+/- num) = QNaN1
(QNaN1) OPERATOR (QNaN2) = QNaN2

- Could generate exceptions

TABLE E-8 NaN Arithmetic Operations That Could Generate Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 B.2.1: Fp convert to wider NaN transformation FsTOd (SNaN2) = QSNaN2 widened	IEEE_754 7.1	
FsTOd(0x7f910000) = 0x7ffa2000 8'h0 FsTOd(0xff910000) = 0xfffa2000 8'h0	IEEE_754 invalid	QSNaN2 widened
SPARC V9 B.2.1: Fp convert to narrower NaN transformation FdTos (SNaN2) = QSNaN2 narrowed	IEEE_754 7.1	
FdTos(0x7ff22000 8'h0) = 0x7fd1000 FdTos(0xfff22000 8'h0) = 0xffd1000	IEEE_754 invalid	QSNaN2 narrowed
SPARC V9 B.2.2 Table 27 Any non-compare arithmetic operations except copying/ moving (fmov, fabs, fneg)	IEEE_754 7.1	
(+/- num) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (+/- num)	IEEE_754 invalid	QSNaN1
(SNaN1) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(QNaN1) OPERATOR (SNaN2)	IEEE_754 invalid	QSNaN2
(SNaN1) OPERATOR (QNaN2)	IEEE_754 invalid	QSNaN1
SPARC V9 Appendix B.5	IEEE_754 7.1	
F{s,d}TOi (+QNaN) = invalid	IEEE_754 invalid	2^{31}-1
F{s,d}TOi (+SNaN) = invalid	IEEE_754 invalid	2^{31}-1
F{s,d}TOx (+QNaN) = invalid	IEEE_754 invalid	2^{63}-1
F{s,d}TOx (+SNaN) = invalid	IEEE_754 invalid	2^{63}-1
F{s,d}TOi (-QNaN) = invalid	IEEE_754 invalid	-2^{31}
F{s,d}TOi (-SNaN) = invalid	IEEE_754 invalid	-2^{31}
F{s,d}TOx (-QNaN) = invalid	IEEE_754 invalid	-2^{63}
F{s,d}TOx (-SNaN) = invalid	IEEE_754 invalid	-2^{63}

E.1.4 Special Inexact Exceptions

UltraSPARC T1 Follows SPARC V9 5.1.7.10.5 (IEEE_754 Section 7.5) and sets FSR_inexact whenever the rounded result of an operation differs from the infinitely precise unrounded result.

Additionally, there are a few special cases to be aware of:

TABLE E-9 Fp <-> Int Conversions With Inexact Exceptions

Cases	Possible Exception	Result (in addition to accrued exception) if tem is cleared
SPARC V9 A.14: Fp convert to 32b integer when source operand lies between $-(2^{31}-1)$ and 2^{31}, but is not exactly an integer FsTOi, FdTOi	IEEE_754 7.5	
	IEEE_754 inexact	An integer number
SPARC V9 A.14: Fp convert to 64b integer when source operand lies between $-(2^{63}-1)$ and 2^{63}, but is not exactly an integer FsTOx, FdTOx	IEEE_754 7.5	
	IEEE_754 inexact	An integer number
SPARC V9 A.15: Convert integer to fp format when 32b integer source operand magnitude is not exactly representable in single precision (23b mantissa). Note, even if the operand is $> 2^{24}-1$, if enough of its trailing bits are zeros, it may still be exactly representable. FiTOs	IEEE_754 7.5	
	IEEE_754 inexact	A SP number
SPARC V9 A.15: Convert integer to fp format when 64b integer source operand magnitude is not exactly representable in single precision (23b mantissa). Note, even if the operand is $> 2^{24}-1$, if enough of its trailing bits are zeros, it may still be exactly representable. FxTOs	IEEE_754 7.5	
	IEEE_754 inexact	A SP number
SPARC V9 A.15: Convert integer to fp format when 64b integer source operand magnitude is not exactly representable in double precision (52b mantissa). Note, even if the operand is $> 2^{53}-1$, if enough of its trailing bits are zeros, it may still be exactly representable. FxTOd	IEEE_754 7.5	
	IEEE_754 inexact	A DP number

E.2 Subnormal Handling

The UltraSPARC T1 FPU provides full hardware support for subnormal operands and results. Unlike UltraSPARC I/II and UltraSPARC III, UltraSPARC T1 will never generate an unfinished_FPop trap type.

APPENDIX **F**

Caches and Cache Coherency

Chapter Revision History

Date	By	Comment
28 Aug 03	Bill Bryg	Started outline of appendix.
22 Sep 03	Bill Croxton	Copied and reformatted Chapter 8, Cache and Memory Interactions, from USIIi User Manual.
25 Nov 03	J. Laudon	Initial changes for UltraSPARC T1
4 Jun 04	J. Laudon	More information on unit of coherence.
2 Aug 04	J. Laudon	Add cache index information.
20 Dec 04	J. Laudon	Better documentation of how to flush L2.
11 Feb 05	M. L. Nohr	Converted to UltraSPARC Architecture format

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

- Invalidation of one or more cache entries – when and how to do it
- Differences between cacheable and noncacheable accesses
- Ordering and synchronization of memory accesses
- Accesses to addresses that cause side effects (I/O accesses)
- Nonfaulting loads
- Cache sizes, associativity, replacement policy, etc.

F.1 Cache Flushing

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

Cache flushing is required in the following cases:

- **I-cache**: Flush is needed before executing code that is modified by a local store instruction other than block commit store, see Section 3.1.1.1, "Instruction Cache (I-cache)." This is done with the FLUSH instruction or by using ASI accesses. When ASI accesses are used, software must ensure that the flush is done on the same virtual core as the stores that modified the code space.
- **D-cache**: Flush is needed when a physical page is changed from (virtually) cacheable to (virtually) noncacheable. This is done with a displacement flush (*Displacement Flushing* on page 116).
- **L2 cache**: Flush is needed for stable storage. Examples of stable storage include battery-backed memory and transaction logs. This is done with a displacement flush (see *Displacement Flushing* on page 116). Flushing the L2 cache flushes the corresponding blocks from the I- and D-caches because UltraSPARC T1 maintains inclusion between the L2 and L1 caches.

F.1.1 Displacement Flushing

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

Note | Diagnostic ASI accesses to the L2 cache can be used to invalidate a line, but they are generally not an alternative to displacement flushing. Modified data in the L2 cache will not be written back to memory using these ASI accesses.

F.1.2 Memory Accesses and Cacheability

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

F.1.3 Coherence Domains

Two types of memory operations are supported in UltraSPARC T1: cacheable and noncacheable accesses, as indicated by the page translation. Cacheable accesses are inside the coherence domain; noncacheable accesses are outside the coherence domain.

SPARC V9 does not specify memory ordering between cacheable and noncacheable accesses. In TSO mode, UltraSPARC T1 maintains TSO ordering, regardless of the cacheability of the accesses. For SPARC V9 compatibility while in PSO or RMO mode, a MEMBAR `#Lookaside` should be used between a store and a subsequent load to the same noncacheable address. See the *SPARC Architecture Manual, Version 9* for more information about the SPARC V9 memory models.

On UltraSPARC T1, a MEMBAR `#Lookaside` executes more efficiently than a MEMBAR `#StoreLoad`.

F.1.3.1 Cacheable Accesses

Accesses that fall within the coherence domain are called cacheable accesses. They are implemented in UltraSPARC T1 with the following properties:

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

F.1.3.2 Noncacheable and Side-Effect Accesses

Accesses that are outside the coherence domain are called noncacheable accesses. Accesses of some of these memory (or memory mapped) locations may result in side effects. Noncacheable accesses are implemented in UltraSPARC T1 with the following properties:

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

Noncacheable accesses with the e bit set (that is, those having side-effects) are all strongly ordered with respect to other noncacheable accesses with the e bit set. Speculative loads with the e bit set cause a *data_access_exception* trap.

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

F.1.3.3 Global Visibility and Memory Ordering

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

CODE EXAMPLE F-1 Memory Ordering and MEMBAR Examples

```
Assume that all accesses go to non-side-effect memory locations.
Process A:
While (1)
{


    Store D1:data produced
1   MEMBAR #StoreStore (needed in PSO, RMO)
    Store F1:set flag
    While F1 is set (spin on flag)
    Load F1
2 MEMBAR #LoadLoad | #LoadStore (needed in RMO)

  Load D2
}

Process B:
While (1)
{

  While F1 is cleared (spin on flag)

       Load F1
2 MEMBAR #LoadLoad | #LoadStore (needed in RMO)

  Load D1

  Store D2
1 MEMBAR #StoreStore (needed in PSO, RMO)

Store F1:clear flag
}
```

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

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

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

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

F.1.4 Memory Synchronization: MEMBAR and FLUSH

The MEMBAR (STBAR in SPARC V8) and FLUSH instructions are provide for explicit control of memory ordering in program execution. MEMBAR has several variations; their implementations in UltraSPARC T1 are described below.

- MEMBAR **#LoadLoad** — Forces all loads after the MEMBAR to wait until all loads before the MEMBAR have reached global visibility.
- MEMBAR **#StoreLoad** — Forces all loads after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility.
- MEMBAR **#LoadStore** — Forces all stores after the MEMBAR to wait until all loads before the MEMBAR have reached global visibility.
- MEMBAR **#StoreStore** and STBAR — Forces all stores after the MEMBAR to wait until all stores before the MEMBAR have reached global visibility.

  Notes | (1) STBAR has the same semantics as MEMBAR #StoreStore; it is included for SPARC V8 compatibility.

  (2) The above four MEMBARs do not guarantee ordering between cacheable accesses after noncacheable accesses.

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

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

- MEMBAR **#MemIssue** — Forces all outstanding memory accesses to be *completed* before any memory access instruction after the MEMBAR is issued. It must be used to guarantee ordering of cacheable accesses following noncacheable

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

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

- MEMBAR **#Sync** (Issue Barrier) — Forces all outstanding instructions and all deferred errors to be completed before any instructions after the MEMBAR are issued.

> **Note** | MEMBAR #Sync is a costly instruction; unnecessary usage may result in substantial performance degradation.

See the references to "Memory Barrier," "The MEMBAR Instruction," and "Programming With the Memory Models," in *The SPARC Architecture Manual, Version 9* for more information.

F.1.4.1 Self-Modifying Code (FLUSH)

The SPARC V9 instruction set architecture does not guarantee consistency between code and data spaces. A problem arises when code space is dynamically modified by a program writing to memory locations containing instructions. LISP programs and dynamic linking require this behavior. SPARC V9 provides the FLUSH instruction to synchronize instruction and data memory after code space has been modified.

In UltraSPARC T1, a FLUSH behaves like a store instruction for the purpose of memory ordering. In addition, all instruction fetch (or prefetch) buffers are invalidated. The issue of the FLUSH instruction is delayed until previous (cacheable) stores are completed. Instruction fetch (or prefetch) resumes at the instruction immediately after the FLUSH.

F.1.5 Atomic Operations

SPARC V9 provides three atomic instructions to support mutual exclusion. These instructions behave like both a load and a store but the operations are carried out indivisibly. Atomic instructions may be used only in the cacheable domain.

An atomic access with a restricted ASI in nonprivileged mode (PSTATE.priv = 0) causes a *privileged_action* trap. An atomic access with a noncacheable address causes a *data_access_exception* trap. An atomic access with an unsupported ASI causes a *data_access_exception* trap. TABLE F-1 lists the ASIs that support atomic accesses.

TABLE F-1 ASIs that Support SWAP, LDSTUB, and CAS

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

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

F.1.5.1 SWAP Instruction

SWAP atomically exchanges the lower 32 bits in an integer register with a word in memory. This instruction is issued only after store buffers are empty. Subsequent loads interlock on earlier SWAPs. A cache miss allocates the corresponding line.

Note | If a page is marked as virtually noncacheable but physically cacheable, allocation is done to the L2 cache only.

F.1.5.2 LDSTUB Instruction

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

F.1.5.3 Compare and Swap (CASX) Instruction

Compare-and-swap combines a load, compare, and store into a single atomic instruction. It compares the value in an integer register to a value in memory; if they are equal, the value in memory is swapped with the contents of a second integer register. All of these operations are carried out atomically; in other words, no other memory operation may be applied to the addressed memory location until the entire compare-and-swap sequence is completed.

F.1.6 Nonfaulting Load

A nonfaulting load behaves like a normal load, except as follows:

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

APPENDIX **G**

Glossary

This chapter defines concepts and terminology common to all implementations of SPARC V9. It also includes terms that are unique to the UltraSPARC T1 implementation.

ALU Arithmetic Logical Unit

address space identifier (ASI) An 8-bit value that identifies an address space. For each instruction or data access, the integer unit appends an ASI to the address. *See also* **implicit ASI**.

application program A program executed with the processor in nonprivileged *mode*. **Note:** Statements made in this specification regarding application programs may not be applicable to programs (for example, debuggers) that have access to *privileged* processor state (for example, as stored in a memory-image dump).

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

ARF Architectural Register File.

ASI Address Space Identifier.

ASR Ancillary State Register.

big-endian An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the most significant; a byte's significance decreases as its address increases.

BLD (Obsolete) abbreviation for Block Load instruction; replaced by LDBLOCKF.

blocking ASI An ASI that will access its ASI register/array location once all older instructions in that strand have retired, there are no instructions in the other strand which can issue, and the store queue, TSW, and LMB are all empty. Additionally, the snoop pipeline is stalled before accessing the ASI register/array location. *See* **nonblocking ASI**.

branch outcome Refers to whether or not a branch instruction will alter the flow of execution from the sequential path. A taken branch outcome results in execution proceeding with the instruction at the branch target; a not-taken branch outcome results in execution proceeding with the instruction along the sequential path after the branch.

branch resolution A branch is said to be resolved when the result (that is, the branch outcome and branch target address) has been computed and is known for certain. Branch resolution can take place late in the pipeline.

branch target address The address of the instruction to be executed if the branch is taken.

BST (Obsolete) abbreviation for Block Store instruction; replaced by STBLOCKF.

bypass ASI An ASI that refers to memory and for which the MMU does not perform virtual-to-real address translation (that is, memory is accessed using a direct real address).

byte Eight consecutive bits of data.

CCR Condition Codes register

clean window A register window in which all of the registers contain 0, a valid address from the current address space, or valid data from the current address space.

CMP Chip multiprocessor. A single chip processor that contains more than one virtual processor. *See also* **processor** and **virtual processor**.

coherence A set of protocols guaranteeing that all memory accesses are globally visible to all caches in a shared-memory system.

commit An instruction commits when it modifies architectural state.

completed A memory transaction is said to be completed when an idealized memory has executed the transaction with respect to all processors. A load is considered completed when no subsequent memory transaction can affect the value returned by the load. A store is considered completed when no subsequent load can return the value that was overwritten by the store.

complex instruction An instruction that requires the creation of secondary "helper" instructions for normal operation, excluding trap conditions such as spill/fill traps (which use helpers).

consistency *See* **coherence**.

context A set of translations that supports a particular address space. *See also* **Memory Management Unit (MMU)**.

CWP Current Window Pointer

CPI Cycles per instruction. The number of clock cycles it takes to execute an instruction.

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

cross-call An interprocessor call in a multiprocessor system.

CSR Control Status Register.

CTI Control transfer instruction

current window The block of 24 R registers that is currently in use. The Current Window Pointer (CWP) register points to the current window.

DCTI Delayed control transfer instruction.

DDR Double Data Rate.

deprecated The term applied to an architectural feature (such as an instruction or register) for which a SPARC V9 implementation provides support only for compatibility with previous versions of the architecture. Use of a deprecated feature must generate correct results but may compromise software performance. Deprecated features should not be used in new SPARC V9 software and may not be supported in future versions of the architecture.

DFT Designed for test.

doublet Two bytes (16 bits) of data.

doubleword An aligned octlet. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

DTLB Data Cache Translation lookaside buffer.

even parity The mode of parity checking in which each combination of data bits plus a parity bit contains an even number of set bits.

exception A condition that makes it impossible for the processor to continue executing the current instruction stream without software intervention. *See also* **trap**.

extended word An aligned octlet, nominally containing integer data. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

EXU Execution Unit

F register A floating-point register. SPARC V9 includes single-, double-, and quad-precision F registers.

fcc*n* One of the floating-point condition code fields fcc0, fcc1, fcc2, or fcc3.

FGU Floating-point and Graphics Unit.

floating-point exception An exception that occurs during the execution of an FPop instruction as defined by the Fpop1, Fpop2, IMPDEP1, and IMPDEP2 opcodes. The exceptions are unfinished_FPop, unimplemented_FPop, sequence_error, hardware_error, invalid_fp_register, or IEEE_754_exception.

floating-point IEEE-754 exception A floating-point exception, as specified by IEEE Std 754-1985. Listed within this specification as IEEE_754_exception.

floating-point operate (FPop) instructions Instructions that perform floating-point and graphics calculations, as defined by the FPop1, FPop2,and IMPDEP1 opcodes. FPop instructions do not include FBfcc instructions, loads and stores between memory and the floating-point unit, or instructions defined by the IMPDEP2 opcodes.

floating-point trap type The specific type of a floating-point exception, encoded in the FSR.ftt field.

floating-point unit A processing unit that contains the floating-point registers and performs floating-point operations, as defined by this specification.

FP Floating Point

FPRS Floating Point Register State (register).

FRF Floating-point register file.

FSR Floating-Point Status register.

GL Global-Level register.

GSR General Status register.

halfword An aligned doublet. **Note:** The definition of this term is architecture dependent and may differ from that used in other processor architectures.

helper An instruction generated by the IRU in response to a complex instruction. Helper instructions are not visible to software. Refer to *Instruction Latencies* on page 76 for a complete list of all complex instructions and their helper sequences.

hexlet Sixteen bytes (128 bits) of data.

hyperprivileged An adjective that describes the state of the processor when it is executing in *hyperprivileged mode*

IFU Instruction Fetch Unit.

implementation Hardware or software that conforms to all of the specifications of an instruction set architecture (ISA).

implementation dependent An aspect of the architecture that can legitimately vary among implementations. In many cases, the permitted range of variation is specified in the SPARC V9 standard. When a range is specified, compliant implementations must not deviate from that range.

implicit ASI The address space identifier that is supplied by the hardware on all instruction accesses and on data accesses that do not contain an explicit ASI or a reference to the contents of the ASI register.

informative appendix An appendix containing information that is useful but not required to create an implementation that conforms to the SPARC V9 specification. *See also* **normative appendix**.

initiated *Synonym*: **issued**.

instruction field A bit field within an instruction word.

instruction set architecture A set that defines instructions, registers, instruction and data memory, the effect of executed instructions on the registers and memory, and an algorithm for controlling instruction execution. Does not define clock cycle times, cycles per instruction, data paths, etc. The bulk of the ISA implemented by UltraSPARC T1 is defined in the *UltraSPARC Architecture 2005*; a few extensions are described in this document.

integer unit (IU) A processing unit that performs integer and control-flow operations and contains general-purpose integer registers and processor state registers, as defined by this specification.

interrupt request A request for service presented to the processor by an external device.

IRF Integer register file.

IRU

ISA Instruction set architecture.

issue Used to describe the act of conveying an instruction from the instruction fetch unit for execution on the pipeline.

L2C Level 2 Cache.

leaf procedure A procedure that is a leaf in the program's call graph; that is, one that does not call (by using CALL or JMPL) any other procedures.

little-endian An addressing convention. Within a multiple-byte integer, the byte with the smallest address is the least significant; a byte's significance increases as its address increases.

load An instruction that reads (but does not write) memory or reads (but does not write) location(s) in an alternate address space. *Load* includes loads into integer or floating-point registers, block loads, Load Quadword Atomic, and alternate address space variants of those instructions. *See also* **load-store** and **store**, the definitions of which are mutually exclusive with *load*.

load-store An instruction that explicitly both reads and writes memory or explicitly reads and writes location(s) in an alternate address space. *Load-store* includes instructions such as CASA, CASXA, LDSTUB, LDSTUBA and the deprecated SWAP and SWAPA instructions. *See also* **load** and **store**, the definitions of which are mutually exclusive with *load-store*.

may A keyword indicating flexibility of choice with no implied preference. **Note:** "May" indicates that an action or operation is allowed; "can" indicates that it is possible.

Memory Management Unit (MMU) The address translation hardware that translates 64-bit virtual address into real addresses. *See also* **context**, **physical address**, and **virtual address**.

must *Synonym*: **shall**.

next program counter (NPC) A register that contains the address of the instruction to be executed next if a trap does not occur.

NFO Nonfault access only.

nonblocking ASI An ASI that will access its ASI register/array location once all older instructions in that strand have retired and there are no instructions in the other strand which can issue. *See* **blocking ASI**.

nonfaulting load A load operation that, in the absence of faults or in the presence of a recoverable fault, completes correctly, and in the presence of a nonrecoverable fault returns (with the assistance of system software) a known data value (nominally zero). *See* **speculative load**.

nonprivileged An adjective that describes:
(1) the state of the processor when PSTATE.priv = 0, that is, nonprivileged mode;
(2) processor state information that is accessible to software while the processor is in either privileged mode or nonprivileged mode; for example, nonprivileged registers, nonprivileged ASRs, or, in general, nonprivileged state;
(3) an instruction that can be executed when the processor is in either privileged mode or nonprivileged mode.

nonprivileged mode The mode in which a processor is operating when PSTATE.priv = 0. *See also* **privileged**.

normative appendix An appendix containing specifications that must be met by an implementation conforming to the SPARC V9 specification. *See also* **informative appendix**.

nontranslating ASI An ASI that does not refer to memory (for example, refers to control/status register(s)) and for which the MMU does not perform address translation.

NPC Next program counter.

npt Nonprivileged trap.

N_REG_WINDOWS The number of register windows present in a particular implementation.

octlet Eight bytes (64 bits) of data. Not to be confused with "octet," which has been commonly used to describe eight bits of data. In this document, the term *byte*, rather than octet, is used to describe eight bits of data.

odd parity The mode of parity checking in which each combination of data bits plus a parity bit contains an odd number of set bits.

older instruction Refers to the relative fetch order of instructions. Instruction *i* is older than instruction *j* if instruction *i* was fetched before instruction *j*. Data dependencies flow from older instructions to younger instructions, and an instruction can only be dependent upon older instructions. *See* **younger instruction**.

one-hot An *n*-bit binary signal is one-hot if, and only if, $n - 1$ of the bits are each 0 and a single bit is 1.

opcode A bit pattern that identifies a particular instruction.

optional A feature not required for compliance to an architecture specification (such as UltraSPARC Architecture 2005 or SPARC V9).

PC Program counter.

PCR Performance Control Register.

PIC Performance Instrumentation Counter.

PIL Processor Interrupt Level.

prefetchable (1) An attribute of a memory location that indicates to an MMU that PREFETCH operations to that location may be applied.
(2) A memory location condition for which the system designer has determined that no undesirable effects will occur if a PREFETCH operation to that location is allowed to succeed. Typically, normal memory is prefetchable.

Nonprefetchable locations include those that, when read, change state or cause external events to occur. For example, some I/O devices are designed with registers that clear on read; others have registers that initiate operations when read. *See* **side effect**.

privileged An adjective that describes
(1) the state of the processor when it is executing in *privileged mode* (PSTATE.priv = 1);
(2) processor state that is only accessible to software while the processor is in *privileged mode*; for example, privileged registers, privileged ASRs, privileged ASIs, or in general, privileged state;
(3) an instruction that can be executed only when the processor is in *privileged mode*.

privileged mode The mode in which a processor is operating when PSTATE.priv = 1. *See also* **nonprivileged**.

processor The unit on which a shared interface is provided to control the configuration and execution of a collection of strands. A processor contains one or more physical cores, each of which contains one or more strands. On a more physical side, a processor is a physical module that plugs into a system. A processor is expected to appear logically as a single agent on the system interconnect fabric.

Therefore, a simple processor, like an UltraSPARC I processor, that can only execute one thread at a time would be a processor with a single physical core that is single-stranded. A processor that follows the academic model of simultaneous multithreading (SMT) would be a processor with a single physical core, where that physical core supports multiple strands in order to execute multiple threads at the same time (multi-stranded physical core). A processor that follows the academic model of a CMP would be a processor with multiple physical cores, each only supporting a single strand.

One can also have multiple physical cores where each physical core is multi-stranded. UltraSPARC T1 is an example of the latter, where each UltraSPARC T1 processor contains eight physical cores, each of which contains four strands.

program counter (PC) A register that contains the address of the instruction currently being executed by the IU.

PSO Partial store order.

quadlet Four bytes (32 bits) of data.

quadword Aligned hexlet. **Note:** The definition of this term is architecture dependent and may be different from that used in other processor architectures.

RA Real address.

RAS Return Address Stack;
also Reliability, Availability and Serviceability.

RAW Read After Write

R register An integer register. Also called a general-purpose register or working register.

rd Rounding direction.

RDPR Read Privileged Register instruction

real address An address used by privileged mode code to describe the underlying physical memory. Real address are usually translated by a combination of hyperprivileged hardware and software to physical addresses which can be used to access real physical memory or I/O device space.

reserved Describing an instruction field, certain bit combinations within an instruction field, or a register field that is reserved for definition by future versions of the architecture.

Reserved instruction fields shall read as 0, unless the implementation supports extended instructions within the field. The behavior of SPARC V9 processors when they encounter nonzero values in reserved instruction fields is undefined.

Reserved bit combinations within instruction fields are defined in Chapter 5, *Instruction Definitions*. In all cases, SPARC V9 processors shall decode and trap on these reserved combinations.

Reserved register fields should always be written by software with values of those fields previously read from that register or with zeroes; they should read as zero in hardware. Software intended to run on future versions of SPARC V9 should not assume that these fields will read as 0 or any other particular value. Throughout this specification, figures and tables illustrating registers and instruction encodings indicate reserved fields and combinations with an em dash (—).

restricted Describing an address space identifier (ASI) that may be accessed only while the processor is operating in privileged mode.

rs1, rs2, rd The integer or floating-point register operands of an instruction. rs1 and rs2 are the source registers; rd is the destination register.

RMO Relaxed memory order.

RTO Read to own (cache line state).

RTS Read to share (cache line state).

shall A keyword indicating a mandatory requirement. Designers shall implement all such mandatory requirements to ensure interoperability with other SPARC V9-compliant products. *Synonym:* **must**.

should A keyword indicating flexibility of choice with a strongly preferred implementation. *Synonym:* **it is recommended**.

SIAM Set interval arithmetic mode instruction.

side effect The result of a memory location having additional actions beyond the reading or writing of data. A side effect can occur when a memory operation on that location is allowed to succeed. Locations with side effects include those that, when accessed, change state or cause external events to occur. For example, some I/O devices contain registers that clear on read; others have registers that initiate operations when read. *See also* **prefetchable**.

SIMD Single instruction stream, multiple data stream.

store An instruction that writes (but does not explicitly read) memory or writes (but does not explicitly read) location(s) in an alternate address space. *Store* includes stores from either integer or floating-point registers, block stores, partial store, and alternate address space variants of those instructions. *See also* **load** and **load-store**, the definitions of which are mutually exclusive with *store*.

strand A term for thread-specific hardware support that identifies the hardware state used to hold a software thread in order to execute it. Strand is specifically the software visible architected state (PC, NPC, general-purpose registers, floating-point registers, condition codes, status registers, ASRs, etc.) of a thread and any microarchitecture state required by hardware for its execution. "Strand" replaces the ambiguous term "hardware thread". The number of strands in a processor defines the number of threads that the operating system can schedule on that processor at any given time. *See also* **thread**, *and* **virtual processor**.

strand identifier (SID) An n-bit value, in a processor implementing 2^n strands, that uniquely identifies each strand. The strand identifier in UltraSPARC T1 is five bits wide.

superscalar An implementation that allows several instructions to be issued, executed, and committed in one clock cycle.

supervisor software Software that executes when the processor is in privileged mode.

thread An executing process or lightweight process (LWP). Historically, the term thread is overused and ambiguous. Software and hardware have historically used it differently. From software's (operating system) perspective, the term thread refers to an entity that can be run on hardware, it is something that is scheduled and may or may not be actively running on hardware at any given time, and may migrate around the hardware of a system. From hardware's perspective, the term multithreaded processor refers to a processor that run multiple software threads simultaneously. To avoid confusion the term thread is used exclusively in the manner in which it is used by software and, specifically, the operating system. A thread can be viewed in a practical sense as a Solaris™ process or lightweight process (LWP). *See also* **strand**, *and* **virtual processor**.

TICK Hardware clock—TICK counter register.

TL Trap Level

TPC Trap-saved PC.

trap The action taken by the processor when it changes the instruction flow in response to the presence of an exception, a Tcc instruction, or an interrupt. The action is a vectored transfer of control to privileged or hyperprivileged software through a table. *See also* **exception**.

TSB Translation storage buffer. A table of the address translations that is maintained by software in system memory and that serves as a cache of the address translations.

TSO Total store order.

TTE Translation table entry. Describes the virtual-to-physical translation and page attributes for a specific page in the Page Table. In some cases, the term is explicitly used for the entries in the TSB.

unassigned A value (for example, an ASI number), the semantics of which are not architecturally mandated and which may be determined independently by each implementation within any guidelines given.

undefined An aspect of the architecture that has deliberately been left unspecified. Software should have no expectation of, or make any assumptions about, an undefined feature or behavior. Use of such a feature can deliver unexpected results, may or may not cause a trap, can vary among implementations, and can vary with time on a given implementation.

Notwithstanding any of the above, undefined aspects of the architecture shall not cause security holes (such as allowing user software to access privileged state), put the processor into supervisor mode, or put the processor into an unrecoverable state.

unimplemented An architectural feature that is not directly executed in hardware because it is optional or is emulated in software.

unpredictable *Synonym:* **undefined**.

unrestricted Describing an address space identifier (ASI) that can be used regardless of the processor mode; that is, regardless of the value of PSTATE.priv.

user application program *Synonym:* **application program**.

VA *Virtual address*.

virtual address An address produced by a processor that maps all systemwide, program-visible memory. Virtual addresses usually are translated by a combination of hardware and software to real addresses.

virtual processor The term **virtual processor**, is used to identify each strand in a processor. Each virtual processor corresponds to a specific strand on a specific physical core where there may be multiple physical cores each with multiple strands. In most respects, a virtual processor appears to the system, and to the operating system software, as a processing unit equivalent to a traditional single-stranded microprocessor (as in UltraSPARC I). Each virtual processor has its own interrupt ID and the operating system can schedule independent threads on each virtual processor. How multiple virtual processors are achieved within a processor is an implementation issue, and as much as possible the software interface is independent of how multiple virtual processors are implemented. The term virtual processor is used in place of strand because of the common association of the term strand with multi-stranded physical cores. *See also* **strand,** *and* **thread**.

VIS™ Visual instruction set.

word An aligned quadlet. Note: The definition of this term is architecture dependent and may differ from that used in other processor architectures.

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

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

WRPR Write Privileged Register.

Index

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W