OpenSPARC™ T2 Core Microarchitecture Specification

Sun Microsystems, Inc.
www.sun.com

Part No. 820-2545-11
December 2007, Revision A

Submit comments about this document at: `http://www.sun.com/hwdocs/feedback`

Contents

Figures

Tables

Preface

This *OpenSPARC T2 Core Microarchitecture Specification* includes detailed functional descriptions of the OpenSPARC T2 SPARC core processor components. This manual also provides I/O signal list for each component. This processor expands Sun's throughput computing initiative by doubling the number of threads from the OpenSPARC T1 processor.

How This Document Is Organized

Chapter 1 describes OpenSPARC T2 Basics

Chapter 2 describes the Introduction to T2 Core Specification

Chapter 3 describes the Instruction Fetch Unit

Chapter 4 describes the Execution Unit

Chapter 5 describes the Load Store Unit

Chapter 6 describes the Cache Crossbar

Chapter 7 describes the Floating-Point and Graphics Unit

Chapter 8 describes the Trap Logic Unit

Chapter 9 describes the Memory Management Unit.

Chapter 10 describes Reliability and Serviceability

Chapter 11 describes ASI/ASR/HPR/PR Access

Chapter 12 describes Reset

Chapter 13 describes Debug

Chapter 14 describes the Power Management

Chapter 15 describes the Performance Monitors

Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:

  `http://docs.sun.com`

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Typographic Conventions

Typeface*	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

* The settings on your browser might differ from these settings.

Related Documentation

The documents listed as online are available at:

http://www.opensparc.net/

Application	Title	Part Number	Format	Location
Documentation	*OpenSPARC T2 Core Microarchitecture Specification*	820-2545	PDF	Online
Documentation	*OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification*	820-2620	PDF	Online

Documentation, Support, and Training

Sun Function	URL
OpenSPARC T2	http://www.opensparc.net/
Documentation	http://www.sun.com/documentation/
Support	http://www.sun.com/support/
Training	http://www.sun.com/training/

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

http://www.sun.com/hwdocs/feedback

Please include the title and part number of your document with your feedback:

OpenSPARC T2 Core Microarchitecture Specification, part number 820-2545-11

CHAPTER 1

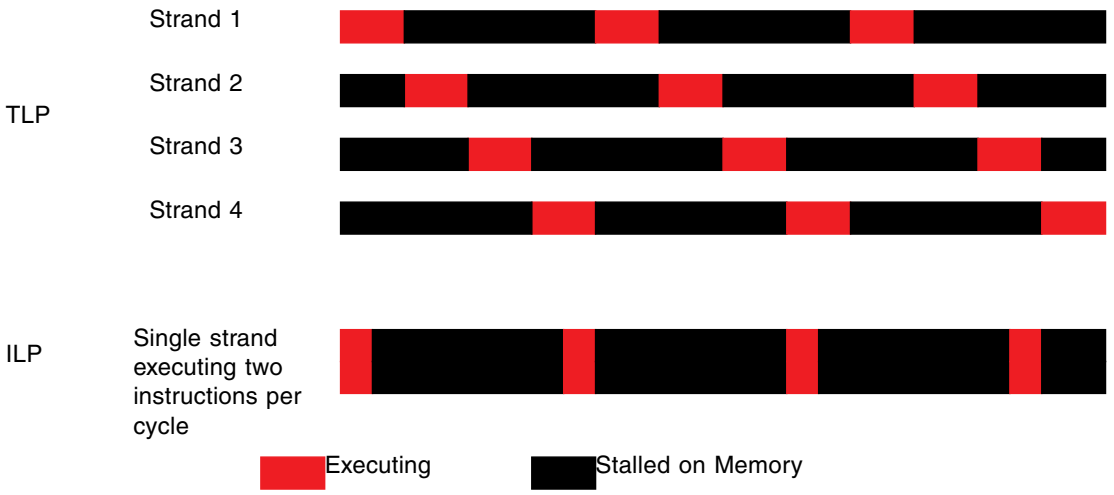
OpenSPARC T2 Basics

1.1 Background

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

Historically, microprocessors have been designed to target desktop workloads, and as a result have focused on running a single thread as quickly as possible. Single thread performance is achieved in these processors by a combination of extremely deep pipelines (over 20 stages in Pentium 4) and by executing multiple instructions in parallel (referred to as instruction-level parallelism or ILP). The basic tenet behind Throughput Computing is that exploiting ILP and deep pipelining has reached the point of diminishing returns, and as a result current microprocessors do not utilize their underlying hardware very efficiently. For many commercial workloads, the processor will be idle most of the time waiting on memory, and even when it is executing it will often be able to only utilize a small fraction of its wide execution width. So rather than building a large and complex ILP processor that sits idle most of the time, a number of small, single-issue processors that employ multithreading are built in the same chip area. Combining multiple processors on a single chip with multiple strands per processor, allows very high performance for highly threaded commercial applications. This approach is called thread-level parallelism (TLP), and the difference between TLP and ILP is shown in the FIGURE 1-1.

FIGURE 1-1 Differences Between TLP and ILP



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

Given this ability to overlap execution with memory latency, why don't more processors utilize TLP? The answer is that designing processors is a mostly evolutionary process, and the ubiquitous deeply pipelined, wide ILP processors of today are the evolutionary outgrowth from a time when the processor was the bottleneck in delivering good performance. With processors capable of multiple GHz clocking, the performance bottleneck has shifted to the memory and I/O subsystems, and TLP has an obvious advantage over ILP for tolerating the large I/O and memory latency prevalent in commercial applications. Of course, every architectural technique has its advantages and disadvantages. The one disadvantage to employing TLP over ILP is that execution of a single thread will be slower on the TLP processor than an ILP processor. With processors running well over a GHz, a strand capable of executing only a single instruction per cycle is fully capable of completing tasks in the time required by the application, making this disadvantage a nonissue for nearly all commercial applications.

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

1.2 OpenSPARC T2 Overview

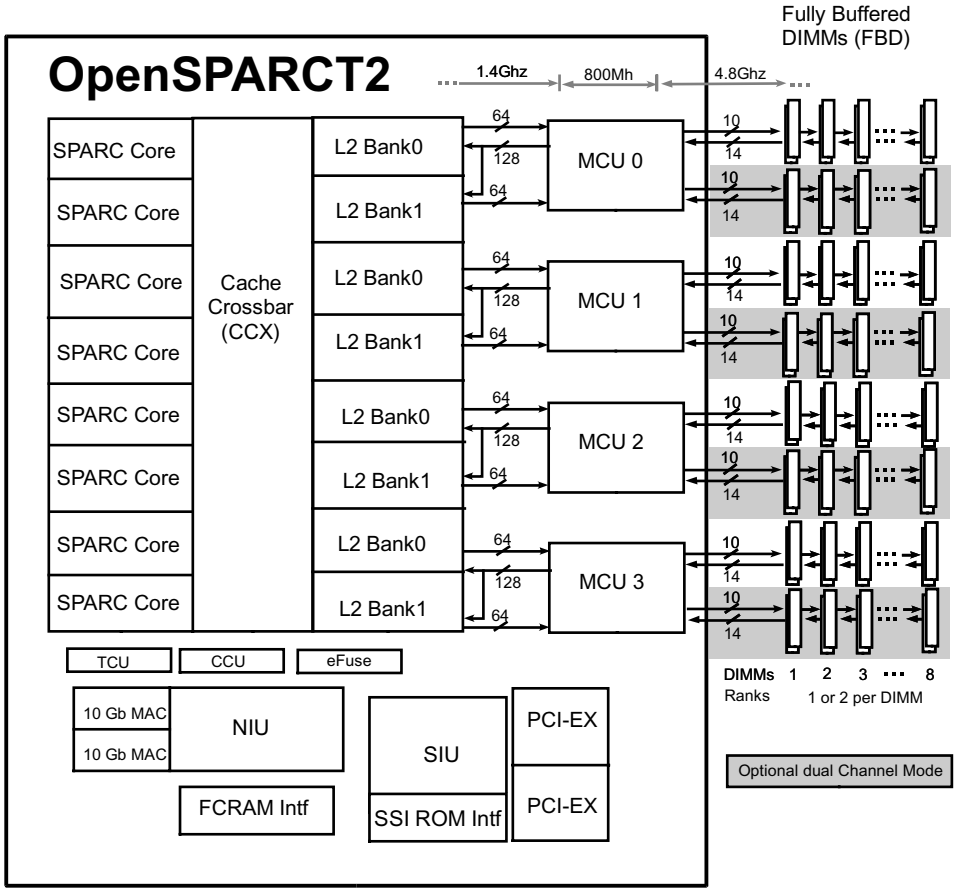
OpenSPARC T2 is a single chip multi-threaded (CMT) processor. OpenSPARC T2 contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for eight strands, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The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

Each SPARC physical core has a 16 KB, 8-way associative instruction cache (32-byte lines), 8 Kbytes, 4-way associative data cache (16-byte lines), 64-entry fully-associative instruction TLB, and 128-entry fully associative data TLB that are shared by the eight strands. The eight SPARC physical cores are connected through a crossbar to an on-chip unified 4 Mbyte, 16-way associative L2 cache (64-byte lines). The L2 cache is banked eight ways to provide sufficient bandwidth for the eight SPARC physical cores. The L2 cache connects to four on-chip DRAM controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels. In addition, an on-chip PCI-EX controller, two 1-Gbit/10-Gbit Ethernet MACs, and several on-chip I/O-mapped control registers are accessible to the SPARC physical cores. Traffic from the PCI-EX port coherently interacts with the L2 cache.

Note – OpenSPARC T2 currently does not include PCI-Express and 10Gigabit Ethernet design implementation due to current legal restrictions. Equivalent models may be available in the subsequent releases of OpenSPARC T2.

A block diagram of the OpenSPARC T2 chip is shown in FIGURE 1-2

FIGURE 1-2 OpenSPARC T2 Chip Block Diagram



1.3 OpenSPARC T2 Components

This section describes each component in OpenSPARC T2.

1.3.1 SPARC Physical Core

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

There is a single floating-point unit within each SPARC physical core for a total of 8 on a T2 chip. Each floating-point unit is shared by all eight strands and fully pipelined. The theoretical floating-point bandwidth is 11 Giga Floating Point Ops (GFlops) per second making the T2 an excellent floating-point processor.

Detailed information on the core processor is provided in *OpenSPARC T2 Core Microarchitecture Specification* (this manual).

1.3.2 SPARC System-On Chip (SoC)

Each SPARC physical core is supported by system on chip hardware components.

Information on each of the functioning units of the system on chip of OpenSPARC T2 are provided in the following chapters of *OpenSPARC T2 System-On Chip (SoC) Microarchitecture Specification*.

1. Chapter 1 - OpenSPARC T2 Basics

This chapter is an overall to the OpenSPARC T2 documents.

2. Chapter 2 - Introduction to T2 Core Specification

This chapter is an introduction to the microarchitecture specification for the OpenSPARC T2 core.

3. Chapter 3 - Instruction Fetch Unit

The IFU provides instructions to the rest of the core. The IFU generates the Program Counter (PC) and maintains the instruction cache (icache).

4. Chapter 4 - Execution Unit

The Execution Unit (EXU) executes all integer arithmetic and logical operations except for integer multiplies and divides. The EXU calculates memory and branch addresses. The EXU handles all integer source operand bypassing.

5. Chapter 5 - Load Store Unit

The OpenSPARC T2 Load Store Unit (LSU) handles memory references between the SPARC core, the L1 data cache, and the L2 cache. All communication with the L2 cache is through the crossbars (processor to cache and cache to processor, a.k.a. PCX and CPX) via the gasket. All SPARC V9 and VIS 2.0 memory instructions are supported with the exception of quad precision floating-point loads and stores.

6. Chapter 6 - Cache Crossbar

The cache crossbar (CCX) connects the 8 SPARC cores to the 8 banks of the L2 cache. An additional port connects the SPARC cores to the IO bridge. A maximum of 8 load/store requests from the cores and 8 data returns/acks/invalidations from the L2 can be processed simultaneously.

7. Chapter 7 - Floating-Point and Graphics Unit

The OpenSPARC T2 floating-point and graphics unit (FGU) implements the SPARC V9 floating-point instruction set, the SPARC V9 integer multiply, divide, and population count (POPC) instructions, and the VIS 2.0 instruction set.

8. Chapter 8 - Trap Logic Unit

The Trap Logic Unit (TLU) manages exceptions, trap requests, and traps for the SPARC core. Exceptions and trap requests are conditions that may cause a thread to take a trap. A trap is a vectored transfer of control to supervisor software through a trap table (from the SPARC Version 9 Architecture). The TLU maintains processor state related to traps as well as the Program Counter (PC) and Next Program Counter (NPC).

9. Chapter 9 - Memory Management Unit

The Memory Management Unit (MMU) reads Translation Storage Buffers (TSBs) for the Translation Lookaside Buffers (TLBs) for the instruction and data caches. The MMU receives reload requests for the TLBs and uses its hardware tablewalk state machine to find valid Translation Table Entries (TTEs) for the requested access. The TLBs use the TTEs to translate Virtual Addresses (VAs) and Real Addresses (RAs) into Physical Addresses (PAs). The TLBs also use the TTEs to validate that a request has the permission to access the requested address.

10. Chapter 10 - Reliability and Serviceability (RAS)

This chapter outlines the OpenSPARC T2 core RAS features. The expected FIT rates of OpenSPARC T2 microarchitectural structures drive the RAS features.

11. Chapter 11 - ASI/ASR/HPR/PR Access

OpenSPARC T2 conceptually has ASI “rings” to access registers defined in ASI space. These registers are accessed using Load and Store alternate instructions. Access to Ancillary State Registers (ASR), Privileged Registers (PR), and

Hyperprivileged Registers (HPR) via RDASR/WRASR, RDPR/WRPR, and RDHPR/WRHPR instructions also occur over the ASI rings. Briefly, there are three logical rings: fast, local, and global.

12. Chapter 12 - Reset

This chapter describes the OpenSPARC T2 reset philosophy and operation.

Similar to previous SPARC processors, OpenSPARC T2 provides several flavors of resets. Resets can be activated as:

- a side-effect of an internal processor or system error, related either to instruction execution or an external event such as failure of a system component
- a result of explicit instruction execution (e.g., SIR)
- a result of a processor write to an ASI register which generates a reset
- a command over an external bus, such as the system bus or the JTAG interface to the Test Control Unit (TCU)
- a result of activating a pin on the OpenSPARC T2 chip

Some resets are local to a given physical core, or affect only one thread (CMP core). Other resets affect all threads.

13. Chapter 13 - Debug

This chapter has been superseded by the PRM Debug chapter.

14. Chapter 14 - Power Management

OpenSPARC T2's power management support consists of two parts. Hardware power management uses clock gating within functional units to reduce power consumed by flops, latches, and static arrays. Since the OpenSPARC T2 core is static, there is no dynamic logic to be power-managed. Hardware power management can be enabled by software.

15. Chapter 15 - Performance Monitors

This chapter describes the OpenSPARC T2 performance monitors. Goals of the performance monitoring capability are:

- Enable data collection to develop accurate modeling for OpenSPARC T2 and future highly threaded processors
- Enable debug of performance issues
- Minimize hardware cost consistent with the above objectives

CHAPTER 2

Introduction to T2 Core Specification

2.1 Introduction

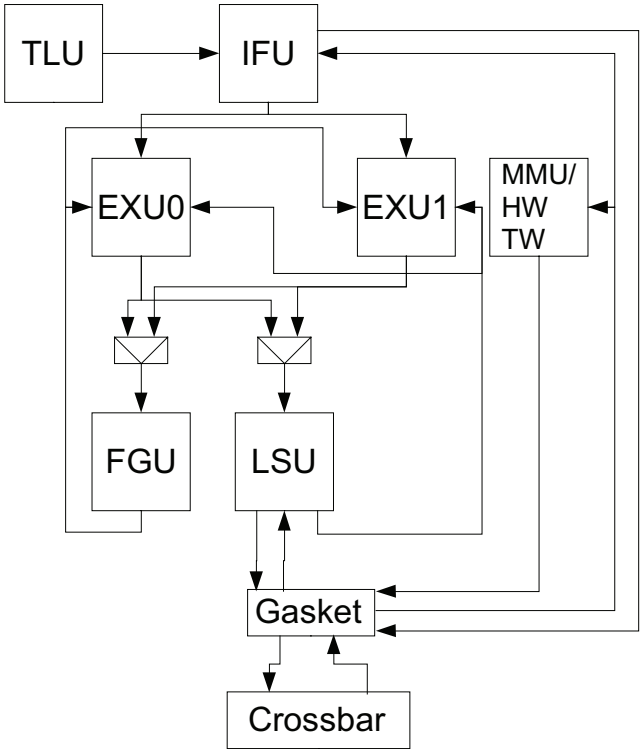
This document is the microarchitecture specification for the OpenSPARC T2 core. The primary audience for this document is:

- core designers
- performance modelers
- Key design aspects of OpenSPARC T2 are:
- full implementation of the SPARC V9 instruction set except for quad instructions including load/store
- full implementation of the VIS 2.0 specification
- support for 8 threads
- 2 integer execution units (EXUs)
- 1 shared load/store unit (LSU)
- 1 shared floating-point and graphics unit (FGU)
- 8 way, 16 KB instruction cache
- 4 way, 8 KB data cache
- 8 stage integer pipeline
- Fetch, Cache, Pick, Decode, Execute, Memory, Bypass, Writeback
- Extended pipeline for long latency operations
- 12 stage floating-point and graphics pipeline
- Fetch, Cache, Pick, Decode, Execute (integer pipe), FX1, FX2, FX3, FX4, FX5, FB, FW

- Capable of sustaining 1 FGU operation per thread every clock
- Fully pipelined between different threads (multiplies are pipelined every other clock)
- Extended pipeline for long latency operations
- Instruction fetching of up to 4 instructions per cycle
- Data TLB of 128 entries, fully associative
- Instruction TLB of 64 entries, fully associative
- Hardware tablewalk support

2.1.1 Overview

FIGURE 2-1 Core Block Diagram



2.1.2 Integer Pipeline

OpenSPARC T2 has an 8 stage integer pipeline as shown below. OpenSPARC T2 adds one additional pipe stage called the pick stage to the original OpenSPARC T1 pipeline. The pick stage enables up to 2 threads to be picked each cycle.

In the bypass stage, the load/store unit (LSU) forwards data to the integer register files (IRFs) with sufficient write timing margin. All integer operations pass through the bypass stage.

Some instructions, such as load misses, fall into a long latency pipe after the bypass stage. OpenSPARC T2 supports at most one long latency instruction per thread.

Integer multiplies are pipelined between different threads. Integer multiplies block within the same thread.

Integer divide is a long latency operation. Integer divides are not pipelined between different threads.

TABLE 2-1 OpenSPARC T2 Integer Pipeline

Fetch (F)	Any Integer Op							
Cache (C)		Any Integer Op						
Pick (P)			Any Integer Op					
Decode (D) Read IRF				Any Integer Op				
Execute (E) Read FRF EXU Data Forward					Any Integer Op			

TABLE 2-1 OpenSPARC T2 Integer Pipeline *(Continued)*

Memory (M) EXU Data Forward						Any Integer Op		
Bypass (B) EXU & Load Data Forward EXU & LSU Trap Status							Any Integer Op	
Writeback (W) EXU & Load Data Forward								Any Integer Op

2.1.3 Floating-point Pipeline

OpenSPARC T2 has a 12 stage floating-point pipeline (6 stage execution pipeline internal to the FGU). All floating-point instructions pass through the integer execute stage. Floating-point instructions that need integer sources obtain them during the execute stage. The floating-point register file (FRF) is accessed during the execute stage of the integer pipe. All floating-point operations except for divide and square root have a fixed latency of 6 cycles in the FGU pipe (FX1-FX5,FB). Floating-point data bypasses to dependent floating-point operations at execute during the float bypass (FB) and float writeback (FW) stages. Floating-point data writes into the FRF during the float writeback (FW) stage.

The FGU executes all integer and floating-point multiplies. Multiplies are fully pipelined.

The FGU executes all integer and floating-point divides. Up to two divides can be below pick at a time across all threads.

OpenSPARC T2 sustains a rate of one FGU instruction every 6 cycles for a given thread if speculation is disabled. If speculation is enabled, OpenSPARC T2 can sustain a rate of one FGU instruction every clock as long as the instructions are independent of one another.

The FGU is fully pipelined between threads except for instructions that have pipelining restrictions (e.g. divide, square root, PDIST).

TABLE 2-2 OpenSPARC T2 Floating-Point Pipeline

Fetch (F)	Any FGU Op											
Cache (C)		Any FGU Op										
Pick (P)			Any FGU Op									
Decode (D) Read IRF				Any FGU Op								
Execute (E) Read FRF					Any FGU Op							
Float1 (FX1 / M)						Any FGU Op						
Float2 (FX2 / B) Predict Exception Status sent to TLU							Any FGU Op					
Float3 (FX3 / W) FCC sent to Decode								Any FGU Op				
Float4 (FX4)									Any FGU Op			

TABLE 2-2 OpenSPARC T2 Floating-Point Pipeline *(Continued)*

Float5 (FX5)										Any FGU Op		
Float Bypass (FB) Actual FGU Trap Status											Any FGU Op	
Float Writeback (FW)												Any FGU Op

CHAPTER 3

Instruction Fetch Unit

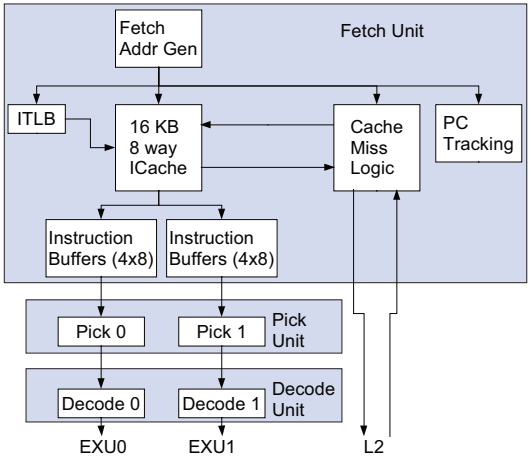
3.1 Instruction Fetch Unit

The IFU provides instructions to the rest of the core. The IFU generates the Program Counter (PC) and maintains the instruction cache (icache).

3.1.1 Overview

The IFU contains three subunits: the fetch unit, pick unit, and decode unit.

FIGURE 3-1 IFU Block Diagram



3.1.2 Fetch Unit

OpenSPARC T2 has an 8-way set associative, 16 KB instruction cache (icache) with a 32 byte line. Each cycle the fetch unit fetches up to 4 instructions for one thread. The fetch unit is shared by all 8 threads of OpenSPARC T2 and only one thread is fetched at a time. The fetched instructions are written into instruction buffers (IBs) which feed the pick logic. Each thread has a dedicated 8 entry IB.

The fetch unit maintains all PC addresses for all threads. The fetch unit redirects threads due to branch mispredicts, LSU synchronization, and traps. The fetch unit handles instruction cache misses and maintains the Miss Buffer (MB) for all threads. The MB ensures that the L2 does not receive duplicate icache misses.

3.1.2.1 Fetch Pipe Stages

Before Fetch Stage

The fetch unit picks the next thread to fetch during the BF stage. The next fetch address is calculated in the BF stage. The fetch thread selection mechanism is discussed in Section 3.1.2.2, “Fetch Thread Pick Mechanism” on page 3-3.

Fetch Stage

The icache data array, the tag array, and the instruction TLB (ITLB) are accessed in parallel during the fetch stage. ITLB hit or miss is determined during this cycle. The data read from all 8 ways of the icache is latched at the end of this cycle. Physical address information from the ITLB and from the tag array is latched at the end of the fetch stage.

Cache Stage

Hit or miss of the icache is determined during the cache stage. Way selects choose the correct instruction data in the cache stage. The cache data is aligned. This aligned data is written into the instruction buffers of the fetched thread.

3.1.2.2 Fetch Thread Pick Mechanism

The fetch unit can only fetch 1 thread at a time because the icache has one port. A Least Recently Fetched (LRF) mechanism ensures fairness in picking this thread out of the 8 possible threads.

Every cycle the fetch unit picks a LRF thread from the set of all READY threads. The picked thread ID (if there is one) is written to the current fetch thread ID register Curr_Fetch_ThreadID.

Thread WAIT conditions

All threads are either in READY state or WAIT state to facilitate the fetch picking process. A thread that is READY can be picked for fetch. The fetch unit can only pick one thread per cycle. A thread in WAIT is waiting for one or more conditions to resolve before it can be picked for fetch. The WAIT state is actually the presence of any of the different specific wait conditions. The READY state is the absence of all of the different specific wait conditions.

Real Miss Wait

A thread that misses the icache during the cache stage and does not hit in the miss buffer (MB) transitions to Real Miss Wait at the end of the cycle. The thread remains in this state until the instruction cache fill data is ready to be written into the icache. Once the fill data is ready to be written, the real miss thread transitions to the Fill Ready state and all other threads transition to the Fill Wait state. The Fill Ready state enables writing of all the cache line from the fill buffer into the icache. Once the data is written in the Fill Ready state, all threads leave the Fill Wait state.

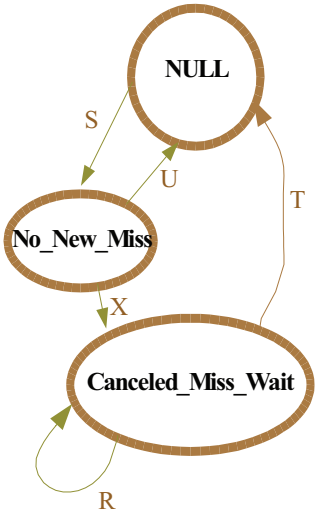
Cache writes and TLB writes can occur in parallel.

Duplicate Miss Wait

A thread that misses the icache during the cache stage and hits in the miss buffer (MB) transitions to Duplicate Miss Wait state at the end of the cycle. The thread remains in this state until the instruction fill data is ready to be written into the icache. Once the fill data is ready to be written, the thread exits the Duplicate Miss Wait state at the end of the cycle.

Canceled Miss Wait

FIGURE 3-2 Canceled Miss Wait State Machine



If a thread with a real miss outstanding is redirected, the thread initiates a fetch from this new redirect address and transitions to a state called NO_NEW_MISS. The thread continues to fetch normally as long as its fetch address hits in the icache. If the thread's fetch address does not hit in the icache, it transitions into the Canceled Miss Wait state. The thread remains in this state until the outstanding cache miss for the relevant thread finishes. This ensures that each thread can have only one outstanding cache miss.

No Room Wait

The fetch unit tracks the number of instructions in each instruction buffer and the number of instructions in flight (in the F and C stages) per thread. If a thread cannot accommodate 3 more instructions in flight, it transitions to the No Room Wait state. A thread exits the No Room Wait state once it has room for 4 or more instructions.

Fill Wait

Fetch can only read or write the icache in a given cycle because the icache has one port. If a thread writes fill data into the icache, no other thread can access the icache in the same cycle. The fetch unit has one fill buffer. Any thread writing into the icache is in Fill Ready state. The Fill Ready state is an extensions of the READY state, i.e. threads can be picked for fetch in the Fill Ready states. If any thread is in either Fill Ready state, all other threads are in the Fill Wait state. Threads exit the Fill Wait state once the writing thread exits the Fill Ready state.

ITLB Miss/Exception Wait

A thread that misses the ITLB during the fetch stage transitions to the ITLB Miss Wait state at the end of the cycle. Any redirection of fetch for this thread causes an exit of the ITLB Miss Wait state.

Fetch pipelines ITLB miss information as an ITLB miss nop instruction down the pipeline. This ITLB miss nop instruction is flushed identically to normal instructions. The nop does not issue to the execution units and does not update any architectural state. The PC for this ITLB miss nop instruction is piped in the normal fashion. If the ITLB miss nop instruction reaches the W stage and hardware tablewalk is enabled, the Trap Logic Unit (TLU) sends a TLB reload request to the MMU. If the hardware tablewalk is successful, the TLU sends the IFU a TLB reload packet. The ITLB writes have the highest priority in fetch arbitration. The cycle after the TLU signals an ITLB write, all threads go to ITLB_WRITE_WAIT for two cycles and the ITLB autodemaps and writes the TTE in the reload packet.

All threads exit the ITLB_WRITE_WAIT state the cycle after the ITLB is written.

If the hardware tablewalk is not successful, the TLU initiates the appropriate trap to the IFU.

ITLB Write Wait

When the TLU returns a TLB reload packet, all threads go to the ITLB Write Wait state for two cycles. During these cycles, the ITLB demaps any matching TTE entries and writes the TTE. The following cycle, all threads exit the ITLB Write Wait state.

Cache writes and TLB writes can occur in parallel.

3.1.2.3 Address Generation

The fetch unit maintains the fetch addresses for all threads. The address generation logic computes the fetch address (PC_BF) for the next fetch stage. Since the OpenSPARC T2 icache is single-ported, only 1 fetch address is needed. The PC_BF accesses the caches, tags, and ITLB. Fetch maintains a set of 8 registers to track the PC of each thread in the machine (T[0.7]_PC_BF).

In the BF stage, a thread is picked for fetch. The PC for this thread is in F the next cycle.

The picked thread's PC address comes from one of the following sources:

- T[0.7]_PC_BF which contain the PCs of the threads. A thread initially picked for fetch always starts from this address.
- PC_INC which contains the incremented PC_BF. If the thread picked for fetch last cycle is picked again, this address is selected. The amount that PC_BF is incremented by can be 4, 8, 12, or 16, depending on the number of instructions fetched (up to 4).
- Br_PC_E_0 or Br_PC_E_1 which is the target address of a branch. Br_PC_E_0 is for branches executed in Thread Group 0 and Br_PC_E_1 is for Thread Group 1.
- Trap_pc_0 or Trap_pc_1 which is the redirection address from TLU. Trap_pc_0 is for thread group 0 and Trap_pc_1 is for thread group 1.
- NPC_w which is used to redirect the IFU to the NPC of a load instruction if the load instruction misses and the instruction behind it is the pipeline.

3.1.2.4 Redirection Sources

Instructions executing in the machine can redirect the fetch unit. Redirection sources are described below.

Branch Misprediction

The target and direction of the branch is known at the end of the execute stage. The EXU supplies the target address to the fetch unit. OpenSPARC T2 predicts that conditional branches are not taken; branch misprediction occurs if a conditional branch is taken. If the branch mispredict thread is the same as the current thread, the target address is written into PC_BF and used for fetching the next cycle. If the branch mispredict thread is not the same as the current thread, the target address is written into the appropriate T[0.7]_PC_F address register. The fetch unit supports up to 2 branch mispredictions in the same cycle, one from each thread group.

LSU Synchronization

If speculation is enabled and a load is found to miss the dcache at the B stage, the LSU synchronizes the relevant thread. The redirection address to fetch in this case is the NPC of the load. The NPC address is always written into the appropriate T[0.7]_PC_BF address register. LSU synchronization is described in more detail in LSU synchronization occurs for a variety of conditions.

Taken Trap

The trap unit generates traps for a variety of conditions. The trap unit supplies a trap valid and the PC of the trap. The trap addresses are written into the appropriate T[0.7]_PC_BF address registers.

Cache Miss Bypass

When a cache miss occurs, the fetch unit writes the PC of the missed instruction into the appropriate T[0.7]_PC_BF register. When the data for the cache miss returns, the cache miss thread transitions to Fill Ready. During Fill Ready, the fetch unit bypasses up to 4 instructions sequentially to the instruction buffers and updates the appropriate T[0.7]_PC_BF with the cache miss address plus the bypass amount.

3.1.2.5 Instruction Data Fetching

In the fetch stage, the cache, tags, and ITLB are accessed using the PC_BF address picked in the previous cycle. Up to 4 instructions can be fetched per cycle. Fetches cannot cross cache lines. Data read from the icache is latched at the end of F stage. The ITLB detects misses and exceptions in the F stage.

In the cache stage, the icache instruction data from the ways is muxed. The muxed instruction data is aligned during the C stage. This alignment left justifies the addressed instruction. This aligned data is written to the instruction buffers of the appropriate thread at the end of the C stage. Icache hit or miss is determined by comparing the physical addresses from the icache tags and the TLB during the cache stage.

3.1.2.6 Cache Invalidate

The fetch unit handles invalidates from the L2 cache. The L2 invalidate indicates which way or ways must be invalidated. L2 invalidates are processed from the fill buffer.

The icache valid array is dual-ported with one port for fetch and the other for invalidates. Invalidates are processed in parallel with thread fetching.

3.1.2.7 PC and NPC Tracking

The fetch unit maintains a PC per thread in the BF stage.

The fetch unit tracks the PC of all instructions at the fetch and cache stages; the fetch unit maintains a single PC per stage for the fetch and cache stages.

The trap unit maintains the architectural PC and NPC at the writeback stage.

The trap unit calculates the PC of instructions at the decode stage for use in generating targets for relative branches.

3.1.2.8 Instruction Cache Miss Handling

The fetch unit handles instruction cache misses for OpenSPARC T2. Instruction cache misses are detected at the cache stage.

PC for Instruction Cache Misses

When a cache miss is detected at the cache stage, the missing thread is flushed from the BF and F stages. The missing thread stores the PC of the missed instruction to the thread register at the BF stage, which holds until the miss data returns.

If the instruction buffer is empty when the icache miss completes, the instruction data load into the appropriate instruction buffer. The instruction data bypasses the cache.

If the instruction buffer is not empty when the icache miss completes, the instruction data bypasses into the appropriate instruction buffer.

Instruction Cache Bypass

The fetch unit supports instruction cache bypassing of up to 4 instructions on cache write. Bypassing only occurs for real misses, not duplicate misses. The number of instructions to bypass depends on where the miss falls in the cache line. Only instructions within the cache line bypass; cache line wrapping is not supported.

The PC_BF of the missing thread is incremented stage the cycle that instructions bypass to the instruction buffers. The BF PC increments by the number of instructions bypassed.

Miss Buffer

The fetch unit divides icache misses into two classes: real misses and duplicate misses. Real misses are icache misses that are not identical to any outstanding icache misses. Duplicate misses are misses that match an outstanding cache miss. The miss buffer (MB) tracks all outstanding icache misses and identifies real and duplicate misses. The number of entries in the MB is 8 to match the number of threads.

Initially, the miss buffer is empty. The MB compares the physical address of a miss to the physical addresses of all outstanding misses. If the icache miss does not hit in the MB, it is a real miss. The fetch unit writes real misses into the MB; the MB indicates which thread is waiting on the miss. If the icache miss hits to the MB, it is a duplicate miss. A duplicate miss updates its corresponding real miss entry to reflect the additional thread that is waiting on the miss. Entries are removed from the MB once the icache miss is complete.

TABLE 3-1 Format of Miss Buffer Entry Format

Waiting_On_Miss[7:0]	Cacheable Bit	Replace Way [2:0]	Imiss_Physical_Addr[39:0]

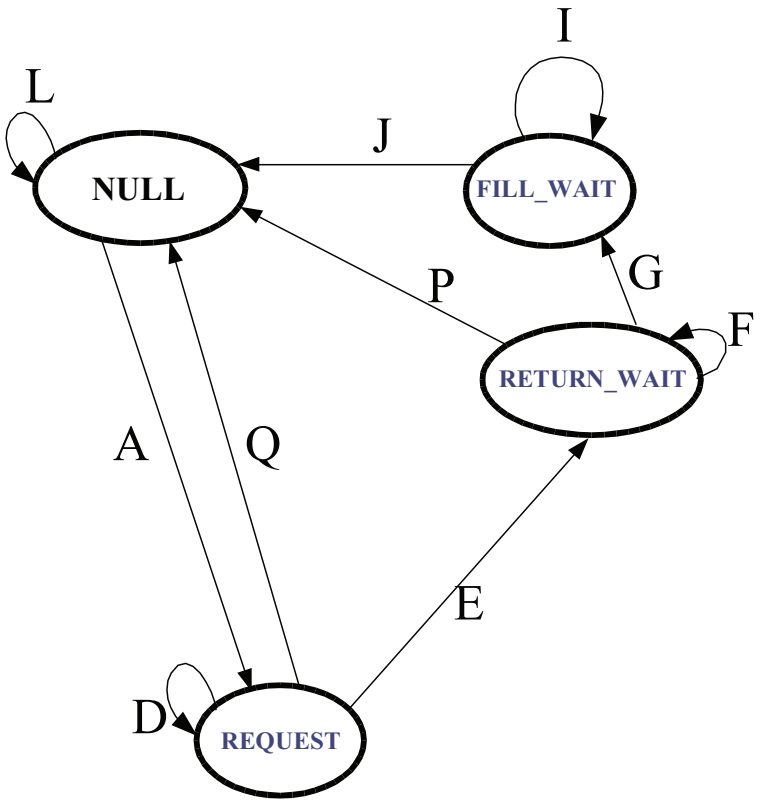
TABLE 3-1 shows the format of a miss buffer entry. Each MB entry has the following fields:

Waiting_On_Miss[7:0]: This field shows all the threads waiting on this miss. The bits in this field update dynamically to reflect threads with new duplicate misses. Similarly, the appropriate bit is reset for any thread that is redirected and no longer needs the icache line.

C: This one bit field shows whether the cache line is cacheable or not. If it is cacheable, the line writes into the icache. If it is not cacheable, the line can only bypass to the instruction buffers. If the C bit is off, this entry cannot have duplicate misses; any miss on the same line creates a new MB entry.

IMISS_PHYSICAL_ADDR[39:0]: This field holds the physical address of the missed line.

FIGURE 3-3 Cache Miss State Machine



Each thread has its own Cache Miss State Machine (CMSM) to interface with the gasket to request cache misses and wait for the cache miss data. Multiple threads can wait for different cache lines at the same time. Threads with duplicate misses remain in the NULL state. Threads with duplicate misses rely on the real miss thread to complete the icache miss.

The key CMSM transitions are defined below:

A: Thread misses the icache and it is a real miss.

Q: If the thread has been flushed and the duplicate misses are flushed before the request is accepted, the entry in the MB is invalidated and the state machine returns to the Null state.

E: When the gasket accepts the IMISS request, the state machine transitions to Return_Wait and waits for the miss data to return.

G: When data returns, the thread state machine transitions to Fill_Wait. During this state, the relevant thread writes the line into the icache.

J: When write completes, the real miss state machine transitions to Null.

Cache Miss Timing Diagrams

The following cycle diagrams show cache miss timing.

Legend:

R = READY

RMW = Real Miss Wait

FR = Fill Ready

DMW = Duplicate Miss Wait

FW = Fill Wait

TABLE 3-2 Real Instruction Cache Miss Timing

Fetch State Machine	R	R	R	RMW	RMW	RMW	FR	R	R
BF	Op 0							Op1	
F		O p0							Op1
C			Op0 Icache Miss Miss Buffer Miss Restore the PC to the Thread registers				Op0		

TABLE 3-2 Real Instruction Cache Miss Timing *(Continued)*

MISS Buffer				Op0 Req to Gasket Ack from Gasket	Op0 WAIT for the DATA Multiple cycles ...	Op0 Data RDY from Gasket	Op0 Bypass the Data. Write the Line to Icache		
P							Op0		
D / IRF								Op0	
E / FRF									Op0

TABLE 3-3 Duplicate Instruction Cache Miss Timing

Fetch State Machine	R	R	R	DMW	DMW	FW	R	R	R	R
BF	Op0						Op0			
F		Op0						Op0		
C			Op0 Icache Miss Miss Buffer Hit Restore the PC/NPC to the Thread registers						Op0	
MISS Buffer	Op0 (already in Miss buffer from an earlier miss)	Op0	Op0 Waiting for the DATA	Op0 Waiting for the DATA Multiple cycles ...	Op0 Data RDY from Gasket	Data written to the Icache by the Real Miss Thread				
P										Op0

3.1.3 Pick Unit

The pick unit attempts to find two instructions to execute among eight different threads. The threads are divided into two different thread groups of four threads each: TG0 (threads 0-3) and TG1 (threads 4-7). The Least Recently Picked (LRP) ready thread within each thread group is picked each cycle. The pick process within a thread group is independent of the pick process within the other thread group. This independence facilitates a high frequency implementation. In some cases, hazards arise because of this independence. For example, each thread group may pick an FGU instruction in the same cycle. Since OpenSPARC T2 has only one FGU, a hardware hazard results. The decode unit resolves hardware hazards that result from independent picking.

3.1.3.1 Pick Unit Overview

The eight instruction buffers (IBs) feed the pick unit. Each instruction buffer contains instructions for one of the eight different threads in the machine. The instructions are maintained in program order with IB entry 0 being the oldest. Each instruction buffer holds up to eight instructions.

The IBs are divided into two thread groups: thread group 0 (TG0) and thread group 1 (TG1). TG0 contains threads 0-3 and TG1 contains threads 4-7. Pick attempts to find one instruction to schedule for execution per thread group. Within a thread group, pick chooses the LRP ready thread, prioritizing between speculative and non-speculative threads. Non speculative threads have higher priority than speculative threads.

Pick maintains a state machine per thread to indicate whether the thread can be picked. A thread is either in READY state or in WAIT state. If a thread is READY and IB entry 0 is valid, it can be picked. If a thread is not READY, then it is in the WAIT state. A thread remains in the WAIT state until the condition or conditions that caused the transition to WAIT are resolved or the thread is flushed. A thread is in WAIT state if any wait conditions exist for the thread. A thread is in READY state if no wait conditions exist for the thread.

Pick is initiated before the type of instruction being picked can be determined. Once the instruction type is known, dependency and resource limitations may require the pick to be canceled for correct machine behavior. A cancel pick transitions the picked thread to WAIT the next cycle unless the condition or conditions giving rise to the hazard or hazards resolve this cycle. If the hazard or hazards resolve this cycle, the thread remains in the READY state.

Threads enter the WAIT state in one of two ways. A thread may enter WAIT after it has been picked to allow dependency and/or hardware hazards to resolve. Alternatively, a thread may enter WAIT before it is actually picked. WAIT conditions for OpenSPARC T2 are discussed in more detail below.

3.1.3.2 Wait Conditions

Post Synchronization Wait Conditions

OpenSPARC T2 transitions a thread to WAIT after certain instructions are picked. These instructions are considered to be post-synchronized (postsync). Instructions that cause this to happen are:

CALL, JMP

DONE, RETRY

Integer instructions executed by the FGU

Integer multiply

POPC

MULSCC

Pixel compare instructions

FLUSH, FLUSHW

MEMBAR

STBAR

Write privileged registers

Write State Registers

SAVE, RESTORE

SAVED, RESTORED

RETURN

Atomics

- CASA
- LDSTUB
- SWAP

LDFSR

PREFETCH

The selected thread transitions from WAIT to READY as soon as the condition it is waiting on resolves (i.e., the instruction completes or is flushed). The TABLE 3-4 below shows the reset WAIT conditions for postsync instructions.

TABLE 3-4 Reset WAIT Conditions for Postsync Instructions

Instruction s	Reset WAIT at end of B stage	Reset WAIT at end of FB stage	Reset WAIT due to Branch Flush	Reset WAIT due to Trap Flush
CALL, JMP				
DONE, RETRY				
Integer Multiply, POPC, MULSCC, Pixel Compares				
FLUSH, FLUSHW, MEMBAR, STBAR				
Write Privileged Registers, Write State Registers				
SAVE, RESTORE, SAVED, RESTORED, RETURN				

Timing diagrams for various postsync instructions are given below.

TABLE 3-5 Call or Return Timing Diagram (branch-taken case)

Thread State	Ready	Wait	Wait	Ready	Ready	Ready	Ready
Cancel Pick							
Branch Mispredict							
P	Call, Return						
D Read IRF		Call, Return					
E Read FRF			Call, Return				
M / FX1				Call, Return			
B / FX2					Call, Return		
W / FX3						Call, Return	
Branch Flush							

TABLE 3-6 Done or Retry Timing Diagram

Thread State	Ready	Wait	Wait	Wait	Wait	Wait	Ready
Cancel Pick							
P	Done, Retry						
D Read IRF		Done, Retry					
E Read FRF			Done, Retry				
M / FX1				Done, Retry			
B / FX2					Done, Retry		
W / FX3						Done, Retry	
Trap Flush							

TABLE 3-7 Integer Instructions Executed by FGU

Thread State	Ready	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Ready	Ready
Cancel Pick											
Completion Status											
P	FGU Integer Op									Next Inst	
D Read IRF		FGU Integer Op				Integer Hole					Next Inst
E Read FRF			FGU Integer Op				Integer Hole				
FX1				FGU Integer Op							
FX2					FGU Integer Op						
FX3						FGU Integer Op					
FX4							FGU Integer Op				
FX5								FGU Integer Op			
FB									FGU Integer Op		
FW											

TABLE 3-7 Integer Instructions Executed by FGU *(Continued)*

M								Integer Hole			
B									Integer Hole		
W										FGU Integer Op	

TABLE 3-8 Timing diagram for SAVE, RESTORE, SAVED, RESTORED

Thread State	Ready	Wait	Wait	Wait	Wait	Ready	Ready	Ready	Ready	Ready
P	Save, Restore ,Saved, Restored	Any Op0	Any Op0	Any Op0	Any Op0	Any Op0	Any Op1			
D Read IRF		Save, Restore ,Saved, Restored					Any Op0	Any Op1		
E Read FRF			Save, Restore ,Saved, Restored					Any Op0	Any Op1	
M / FX1				Save, Restore ,Saved, Restored					Any Op0	Any Op1
B / FX2					Save, Restore ,Saved, Restored					Any Op0
W / FX3						Save, Restore ,Saved, Restored				

TABLE 3-8 Timing diagram for SAVE, RESTORE, SAVED, RESTORED *(Continued)*

IRF internal decode					Save or Restore Ops					
IRF internal save						Save or Restore Ops				
IRF internal restore							Save or Restore Ops			

TABLE 3-9 RETURN Instruction Timing Diagram

Thread State	Ready	Wait	Wait	Wait	Wait	Ready	Ready	Ready	Ready	Ready
BF				Next Op						
F					Next Op					
C						Next Op				
P	Return	Delay Slot	Delay Slot	Delay Slot	Delay Slot	Delay Slot	Next Op			
D Read IRF		Return					Delay Slot	Next Op		
Branch Mispredict								Delay Slot	Next Op	
E Read FRF			Return						Delay Slot	Next Op
M / FX1										Delay Slot
B / FX2										
W / FX3										
IRF internal decode					Restore Op					
IRF internal save						Restore Op				
IRF internal restore							Restore Op			

Note – The branch mispredict for the RETURN instruction does not clear the pick WAIT state.

Speculation Not Enabled Wait Conditions

OpenSPARC T2 transitions the selected thread to WAIT state as soon as certain instructions are picked and speculation is not enabled. These instructions include:

All loads (including VIS 2.0)

All FGU instructions which produce FP results (including VIS 2.0)

Bicc, BPcc, BPr, FBfcc, FBPfcc

The selected thread transitions from WAIT to READY as soon as the thread is no longer speculative (i.e., the instruction completes or is flushed).

The timing diagrams below illustrate several cases of speculation enabled and disabled.

TABLE 3-10 Timing Diagram For Any Integer Branch (not-taken) Followed By Any Op With Speculation Enabled

Thread State	Ready	Ready	Ready	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick									
Completion Status									
P	Any Integer Branch	Any Op1	Any Op2	Any Op3					
D Read IRF		Any Integer Branch	Any Op1	Any Op2	Any Op3				
E Read FRF			Any Integer Branch	Any Op1	Any Op2	Any Op3			
M/FX1				Any Integer Branch	Any Op1	Any Op2	Any Op3		
B/FX2					Any Integer Branch	Any Op1	Any Op2	Any Op3	
W/FX3						Any Integer Branch	Any Op1	Any Op2	Any Op3

TABLE 3-11 Timing Diagram For Any Integer Branch (not-taken) Followed By Any Op With Speculation Disabled

Thread State	Ready	Wait	Wait	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick									
Completion Status									
P	Any Integer Branch	Any Op	Any Op	Any Op					
D Read IRF		Any Integer Branch			Any Op				
E Read FRF			Any Integer Branch			Any Op			
M/FX1				Any Integer Branch			Any Op		
B/FX2					Any Integer Branch			Any Op	
W/FX3						Any Integer Branch			Any Op

TABLE 3-12 Timing Diagram for Any Load Followed By Any Op With Speculation Disabled (dcache hit case)

Thread State	Ready	Ready	Wait	Wait	Wait	Ready	Ready	Ready	Ready
Cancel Pick									
Completion Status									
P	Any Load	Any Op	Any Op	Any Op	Any Op	Any Op			
D Read IRF		Any Load					Any Op		
E Read FRF			Any Load					Any Op	

TABLE 3-12 Timing Diagram for Any Load Followed By Any Op With Speculation Disabled (dcache hit case) *(Continued)*

M/FX1				Any Load					Any Op
B/FX2					Any Load				
W/FX3						Any Load			

TABLE 3-13 Timing Diagram For FGU Operation Followed by Any Op With Speculation Disabled

Thread State	Ready	Wait	Wait	Wait	Wait	Wait	Ready			
Cancel Pick										
Completion Status										
P	FGU op	Any Op	Any Op	Any Op	Any Op	Any Op	Any Op			
D Read IRF		FGU op						Any Op		
E Read FRF			FGU op						Any Op	
M/FX1				FGU op						
B/FX2					FGU op					
W/FX3						FGU op (FCC sent here)				
FX4							FGU op			

TABLE 3-13 Timing Diagram For FGU Operation Followed by Any Op With Speculation Disabled

FX5								FGU op		
FB									FGU op (FP data sent here)	
FW										FGU op

Dependency Wait Conditions

OpenSPARC T2 picks instructions after integer load or FGU instructions if the subsequent instruction does not have register dependencies on prior instructions in the pipe and speculation is enabled. If the subsequent instruction has register dependencies on prior instructions in the pipe, it is not picked until all dependency hazards resolve.

OpenSPARC T2 has no dependency logic for the FP condition codes (FCCs). All instructions that source the FCC bits (FBfcc, FBPfcc, MOVCC, FMOVCC) are considered to have a FCC dependency on any prior FGU op in the D, E, M, B and W stages which updates the FCC bits. No pick occurs for a FP branch with a FCC dependency. The IFU maintains a shadow copy of the FCCs per thread; the FGU maintains the master copy. The shadow copies of the FCCs reflect the master copies delayed one cycle in time.

OpenSPARC T2 usually transitions the relevant thread to WAIT state as soon as a dependency is detected between the current instruction and any previous instruction in the pipe. A thread does not transition to WAIT if all dependency hazards resolve the next cycle. Pick is canceled if a dependency hazard is detected.

The thread transitions from WAIT to READY the cycle prior to the resolution of all dependency hazards.

The timing diagrams below illustrate several dependency wait conditions on OpenSPARC T2.

TABLE 3-14 Timing Diagram For Any Load Followed By Independent Operations With Speculation Enabled (hit dcache case)

Thread State	Ready	Ready	Ready	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick									
Completion Status									
P	Any Load	Independent Op1	Independent Op2	Independent Op3	Independent Op4				
D Read IRF		Any Load	Independent Op1	Independent Op2	Independent Op3	Independent Op4			
E Read FRF			Any Load	Independent Op1	Independent Op2	Independent Op3	Independent Op4		
M/FX1				Any Load	Independent Op1	Independent Op2	Independent Op3	Independent Op4	
B/FX2					Any Load	Independent Op1	Independent Op2	Independent Op3	Independent Op4
W/FX3						Any Load	Independent Op1	Independent Op2	Independent Op3

TABLE 3-15 Timing Diagram For Integer Load Followed By Dependent Op With Speculation Enabled (hit dcache case)

Thread State	Ready	Ready	Wait	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick									
Completion Status (internal to pick)									
P	Integer Load	Dependent Op	Dependent Op	Dependent Op					
D Read IRF		Integer Load			Dependent Op				
E Read FRF			Integer Load			Dependent Op			
M/FX1				Integer Load			Dependent Op		
B/FX2					Integer Load			Dependent Op	
W/FX3						Integer Load			Dependent Op

TABLE 3-16 Timing Diagram For FGU Load Followed By Dependent Op With Speculation Enabled (dcache hit case)

Thread State	Ready	Ready	Ready	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick									
Completion Status (internal to pick)									
P	FGU Load	Dependent Op	Dependent Op						
D Read IRF		FGU Load		Dependent Op					

TABLE 3-16 Timing Diagram For FGU Load Followed By Dependent Op With Speculation Enabled (dcache hit case) *(Continued)*

E Read FRF			FGU Load		Dependent Op				
M/FX1				FGU Load		Dependent Op			
B/FX2					FGU Load		Dependent Op		
W/FX3						FGU Load		Dependent Op	

TABLE 3-16 assumes that FGU load data can be bypassed the cycle it is driven.

TABLE 3-17 Timing Diagram For FGU Operation Followed By Dependent Operation With Speculation Enabled

Thread State	Ready	Ready	Ready	Wait	Wait	Wait	Wait	Ready		
Cancel Pick										
Completion Status (internal to pick)										
P	FGU op0	FGU op1	Depends on FGU op1	Depends on FGU op1	Depends on FGU op1	Depends on FGU op1	Depends on FGU op1	Depends on FGU op1		
D Read IRF		FGU op0	FGU op1						Depends on FGU op1	
E Read FRF			FGU op0	FGU op1						Depends on FGU op1
M/FX1				FGU op0	FGU op1					
B/FX2					FGU op0	FGU op1				

TABLE 3-17 Timing Diagram For FGU Operation Followed By Dependent Operation With Speculation Enabled *(Continued)*

W/FX3 (FCC sent here)						FGU op0	FGU op1			
FX4							FGU op0	FGU op1		
FX5								FGU op0	FGU op1	
FB (FP data sent here)									FGU op0	FGU op1
FW										FGU op0

TABLE 3-18 Timing Diagram For Write After Write (WAW) Hazard For Any FGU Op Followed By Load Float Both Writing Same Register

Thread State	Ready	Ready	Wait	Wait	Wait	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick											
P	Any FGU op write F0	Any FP load write F0	Any FP load write F0	Any FP load write F0	Any FP load write F0	Any FP load write F0					
D Read IRF		Any FGU op write F0					Any FP load write F0				
E Read FRF			Any FGU op write F0					Any FP load write F0			
M / FX1				Any FGU op write F0					Any FP load write F0		

TABLE 3-18 Timing Diagram For Write After Write (WAW) Hazard For Any FGU Op Followed By Load Float Both Writing Same Register *(Continued)*

B / FX2					Any FGU op write F0					Any FP load write F0	
W / FX3						Any FGU op write F0					Any FP load write F0
FX4							Any FGU op write F0				
FX5								Any FGU op write F0			
FB									Any FGU op write F0		
FW										Any FGU op write F0	

TABLE 3-19 STFSR Timing Diagram

Thread State	Ready	Ready	Wait	Wait	Wait	Wait	Wait	Wait	Ready	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick															
Completion Status															
P	Any FGU op	STFSR	STFSR	STFSR	STFSR	STFSR	STFSR	STFSR	STFSR	Any FGU op	Any FGU op				
D Read IRF		Any FGU op								STFSR	Any FGU op	Any FGU op			
E Read FRF			Any FGU op								STFSR	Any FGU op	Any FGU op		
M / FX1				Any FGU op								STFSR	Any FGU op	Any FGU op	
B / FX2					Any FGU op								STFSR	Any FGU op	Any FGU op
W / FX3 (FCC sent here)						Any FGU op								STFSR	Any FGU op
FX4							Any FGU op								

TABLE 3-19 STFSR Timing Diagram *(Continued)*

FX5								Any FGU op							
FB (FP data sent here)									Any FGU op						
FW										Any FGU op					

TABLE 3-20 LDFSR Timing Diagram

Thread State	Ready	Ready	Wait	Wait	Wait	Wait	Wait	Wait	Ready	Wait	Wait	Wait	Ready	Ready	Ready
Cancel Pick															
Completion Status															
P	Any FGU Op	LD FSR	LD FSR	LD FSR	LD FSR	LD FSR	LD FSR	LD FSR	LD FSR	Any FGU Op	Any FGU Op	Any FGU Op	Any FGU Op	Any FGU Op	
D Read IRF		Any FGU Op								LD FSR				Any FGU Op	Any FGU Op
E Read FRF			Any FGU Op								LD FSR				Any FGU Op
M/FX1				Any FGU Op								LD FSR			
B/FX2					Any FGU Op								LD FSR		
W/FX3 (FCC sent here)						Any FGU Op								LD FSR	
FX4							Any FGU Op								

TABLE 3-20 LDFSR Timing Diagram *(Continued)*

FX5								Any FGU Op							
FB (FP data sent here)									Any FGU Op						
FW										Any FGU Op					

LDFSR has a presync just as STFSR does.

TABLE 3-21 Timing Diagram For Any FGU Op That Writes FCC Followed By FBfcc, MOVfcc, FMOVfcc

Thread State	Ready	Ready	Wait	Wait	Ready	Ready	Ready	Ready	Ready	Ready	Ready
Cancel Pick											
P	Any FGU Op	FBfcc	FBfcc	FBfcc	FBfcc						
D Read IRF		Any FGU Op				FBfcc					
E Read FRF			Any FGU Op				FBfcc				
FX1 / M				Any FGU Op							
FX2 / B					Any FGU Op						

TABLE 3-21 Timing Diagram For Any FGU Op That Writes FCC Followed By FBfcc, MOVfcc, FMOVfcc

FX3 / W						Any FGU Op (FCC status sent here)					
FX4							Any FGU Op				
FX5								Any FGU Op			
FB									Any FGU Op		
FW										Any FGU Op	

The FGU sends FCC nonspeculatively during FX3.

Divide Wait Conditions

OpenSPARC T2 transitions the selected thread to WAIT state when a divide is picked.

OpenSPARC T2 has only one divider; it resides in the floating-point and graphics unit (FGU).

After a divide is picked in either TG0 or TG1, no divides are allowed into the machine from any thread until all outstanding divides complete (up to two). Since pick is independent between thread groups, two divides can be picked the same cycle.

The selected thread transitions from WAIT to READY the cycle after all outstanding divides complete.

OpenSPARC T2 transitions a thread to WAIT state as soon as a divide is detected in IB entry 0 and a divide is already outstanding for any of the other threads. The thread cancels pick in this case.

The selected thread transitions from WAIT to READY as soon as all pending divides complete or are flushed.

A timing diagram for divides is given below. Divides have the highest priority for accessing the W2 port of the IRF and the FRF. FGU stall 1 is required because the FGU has a single 64 bit integer bus shared between integer divides and FGU integer ops. FGU stall 1 ensures that no FGU integer op is scheduled that conflicts for use of this shared bus. FGU stall 2 is required because the LSU drives load-float dcache hits into the W2 port of the IRF. FGU stall 2 ensures that no load-float (dcache hit case) operation is scheduled that conflicts for use of the W2 port with a floating-point divide operation. All integer and floating-point divides generate stalls.

TABLE 3-22 Divide Timing Diagram

Thread State	Ready	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Wait	Ready	Ready
Cancel Pick																	
Branch Flush																	
P	Div															Any Op	Any Op
D Read IRF		Div						NO Integer Op that executes on FGU Decodes				NO LSU Op Decodes					Any Op
E Read FRF			Div														
FX1 / M				Div													
FX2 / B					Div												
FX3 / W						Div											
FX4																	
FX5																	
FGU stall1																	

TABLE 3-22 Divide Timing Diagram *(Continued)*

FGU stall2																	
Divide Complete																	
Internal FGU Divide Pipeline							Div	Div	Div	Div	Div	Div	Div	Div			
FB															Div		
FW																Div	
FGU stall load miss to LSU																	
LSU no load miss write W2																	

Delayed CTI Wait Conditions

OpenSPARC T2 transitions a thread to WAIT state when a delayed CTI (DCTI) is detected in IB entry 0, IB entry 0 is valid, and IB entry 1 is not valid. Any pick of a thread in this state is canceled. DCTIs are not allowed into the machine unless the delay slot of the DCTI is known. This simplifies the problem of delay slot handling; when OpenSPARC T2 executes a DCTI, its delay slot is in either IB entry 0 or the decode stage.

Once a thread is in WAIT state due to a DCTI without a delay slot, it transitions to READY as soon as any write to the IB occurs.

For the boundary case in which a DCTI without a delay slot is detected and a write to the IB is occurring, the thread does not transition to the WAIT state. The thread cancels pick in this case.

OpenSPARC T2 transitions a thread to WAIT state as soon as a DCTI is detected in IB entry 0 and a DCTI is detected at decode. Pick of a thread under these conditions is canceled.

If a DCTI is detected in IB entry 0 and a DCTI is found at execute, the thread does not transition to the WAIT state from READY. The thread cancels pick in this case.

OpenSPARC T2 does not allow the second DCTI of a DCTI couple to advance past pick until the first DCTI has executed. This guarantees that the delay slot of the second DCTI is known.

LSU Synchronization Wait Conditions

A LSU synchronization occurs when the latency of a LSU operation is greater than the normal latency of the integer pipeline. One example of a LSU synchronization is when a speculative load misses the dcache (in the B stage). In the case of a LSU synchronization, the relevant thread is flushed, the appropriate WAIT state conditions are reset, a LSU synchronization WAIT state is set, and the instruction after the LSU instruction is refetched from the icache.

The thread transitions from LSU synchronization WAIT to READY once the LSU signals that the synchronization is complete or the TLU flushes the relevant thread.

Store Buffer Full Wait Conditions

The LSU has an eight entry store buffer for each thread. The LSU informs the IFU each cycle of the number of entries deallocated from the store buffer per thread. The pick logic maintains a four bit speculative store counter per thread. This counter increments every time a store is picked. It decrements every time a store buffer entry is deallocated. Pick also maintains a four bit actual store counter per thread. This counter increments every time a store reaches the W stage and is not flushed. The counter decrements every time a store buffer entry deallocates. Two cycles after any flush, the actual store counter overwrites the speculative store counter.

Pick transitions to the WAIT state as soon as a valid store is detected and a store buffer full condition exists (i.e., speculative store counter[3] == 1). The pick of any store under a store buffer full condition is canceled. Pick remains in the WAIT state as long as a store buffer full condition exists. Pick transitions from WAIT to READY when the store buffer full condition no longer exists (i.e., speculative store counter[3] == 0).

3.1.3.3 Trap Synchronization

The following instructions require trap synchronization (trapsync) by the TLU in order to execute correctly:

FLUSH

MEMBAR

STBAR

Write privileged registers

Write state registers

Stores to some ASI registers

The TLU treats instructions requiring trapsync as long latency operations. The LSU generates the completion signal for an instruction requiring trapsync. This completion signal is generated after the instruction:

updates architectural state if required

updates internal processor state if required (e.g., cache line valid bits)

satisfies any coherency requirements

The TLU redirects fetch to the NPC of the trapsync'ed instruction once it receives the completion signal from the LSU.

The pick unit ensures that no instruction from a thread enters the machine after an instruction requiring trapsync by generating a postsync wait condition; see Section , "Post Synchronization Wait Conditions" on page 3-14.

3.1.3.4 LSU Synchronization

The following instructions generate LSU synchronizations to the pick unit:

Loads that miss the dcache

Loads to ASI registers

Stores to some ASI registers

Read State Register instructions

Read Privileged Register instructions

CASA

SWAP

LDSTUB

PREFETCH, PREFETCHA

LSU synchronization is described further in Section 3.1.3.4, "LSU Synchronization" on page 3-38.

3.1.3.5 Speculation

OpenSPARC T2 supports three forms of thread speculation: load-hit speculation (same as N1), branch speculation, and FGU exception prediction.

If speculation is enabled, all integer loads (non-atomic) are assumed to hit in the L1 dcache. If a load that is speculated to hit turns out to miss the dcache, the thread is flushed (LSU synchronization), put into the WAIT state, and refetched from the icache. The thread transitions from WAIT to READY when the load miss completes.

If speculation is enabled, all conditional integer branches are assumed to be not-taken. In the event a conditional branch is found to be taken at execute, the thread is flushed (the delay slot is annulled or not as appropriate) and the target of the branch is fetched from the icache. The thread transitions to the READY state (it is not picked until IB entry 0 is valid).

If speculation is enabled, the FGU predicts the exception status of every FGU instruction. The predicted trap status is reported to the TLU during the B stage. FGU exception prediction is described further in Section 3.1.4.3, "FGU – FGU Hazard" on page 3-42.

In OpenSPARC T2, a thread is speculative if a conditional integer branch is in the decode or execute stage and speculation is enabled OR an integer load (non-atomic) is in the decode, execute, memory, bypass or writeback stage and speculation is enabled. Threads that do not meet these conditions are considered non-speculative. Non-speculative threads have higher priority than speculative threads during the thread picking process.

3.1.3.6 Thread Flushing

Threads flush on OpenSPARC T2 for the following reasons: branch mispredict, LSU synchronization, or any trap flush. In order to handle flushes correctly, each thread on OpenSPARC T2 independently tracks speculative instructions until resolution.

An integer conditional branch causes a flush if it is taken. In this case, the delay slot is annulled as appropriate, the thread is flushed, the appropriate WAIT state conditions are reset, and the target of the branch is fetched from the icache.

A load synchronization flush occurs when a load that is assumed to hit in the dcache is found to miss (in the B stage). In the case of a load synchronization flush, the relevant thread is flushed, the appropriate WAIT state conditions are reset, a load-miss WAIT state is set, and the instruction after the load that missed is refetched from the icache. The thread transitions from LSU synchronization WAIT to READY once the LSU synchronization completes. LSU synchronizations occur for a variety of reasons; see Section 3.1.3.4, "LSU Synchronization" on page 3-38.

A trap flush is signaled by the trap unit for a given thread. In this case, the thread is flushed, the appropriate WAIT state conditions are reset, and the appropriate trap address is fetched from the icache under trap unit control.

3.1.4 Decode Unit

The decode unit decodes one instruction from each thread group (TG0 and TG1) per cycle. Decode determines the outcome of all instructions that depend on the CC and FCC bits (conditional branches, conditional moves, etc.). The integer source operands rs1 and rs2 are read from the IRF during the decode stage. The integer source for integer stores is also read from the IRF during decode stage. The decode unit supplies pre-decodes to the execution units.

The decode unit resolves scheduling hazards not detected during the pick stage between the 2 thread groups. These scheduling hazards include:

- Both TG0 and TG1 instructions require the LSU AND the FGU unit (storeFGU-storeFGU hazard)
- Both TG0 and TG1 instructions require the LSU (load-load hazard, including all loads and integer stores)
- Both TG0 andTG1 instructions require the FGU (FGU-FGU hazard)
- Either TG0 or TG1 is a multiply and a multiply block stall is in effect (multiply block hazard)
- Either TG0 or TG1 require the FGU unit and a PDIST block is in effect (PDIST block hazard)

In OpenSPARC T2, the FGU executes all multiplies and divides (integer and floating-point); instruction scheduling identifies them as FGU operations. Instruction scheduling also identifies MULSCC, POPC and pixel compares as FGU operations. The LSU executes floating-point loads; instruction scheduling identifies them as load operations. The LSU and FGU both participate in executing floating-point stores; instruction scheduling identifies them as both FGU operations and load operations.

The load-load hazard case is illustrated below. This hazard exists when 2 instructions are at decode at the same time and both require the LSU. In this example, both thread groups pick instructions that require the LSU. The load-favor bit decides which load decodes this cycle and which load stalls. The load-favor bit toggles once a load decodes under a load-load hazard condition.

The decode unit assists in executing integer instructions that require 2 cycles to execute.

The decode unit assists in executing block store instructions; see Section 3.1.4.7, “Block Store Hazards” on page 3-43

The decode unit assists in writing the DTLB upon a miss and a hardware tablewalk hit; see Section 3.1.4.8, “DTLB Reloads” on page 3-43

TABLE 3-23 Load-Load Hazards

TG0 Pick	TG0_Load0	TG0_Load1		TG0_Load2					
TG1 Pick	TG1_Load0		TG1_Load1						
TG0 Decode Read IRF		TG0_Load0 decodes	TG0_Load1 stall	TG0_Load1 decodes	TG0_Load2 stall	TG0_Load2			
TG1 Decode Read IRF		TG1_Load0 stall	TG1_Load0 decodes	TG1_Load1 stall	TG1_Load1 decodes				
Load Favor Bit	0	0	1	0	1	0	0	0	0
		Load-Load decode hazard	Load-Load decode hazard	Load-Load decode hazard	Load-Load decode hazard				
TG0 Execute			TG0_Load0		TG0_Load1		TG0_Load2		
TG1 Execute				TG1_Load0		TG1_Load1			
Memory EXU Trap Status				TG0_Load0	TG1_Load0	TG0_Load1	TG1_Load1	TG0_Load2	
Bypass (assume hit to dcache)					TG0_Load0	TG1_Load0	TG0_Load1	TG1_Load1	TG0_Load2
Writeback						TG0_Load0	TG1_Load0	TG0_Load1	TG1_Load1

3.1.4.1 StoreFGU – StoreFGU Hazard

StoreFGU-storeFGU hazards exist when 2 store-float instructions are present at decode, since both require the FGU and the LSU. StoreFGU-storeFGU hazards always have priority over load-load or FGU-FGU hazards; this avoids deadlock if the load and FGU favor status are for opposite thread groups. A storeFGU favor bit decides which storeFGU decodes and which storeFGU stalls.

If the storeFGU favor bit is 0, then storeFGU0 decodes, storeFGU1 stalls, and the storeFGU favor bit is set to 1. If the storeFGU favor bit is 1, then storeFGU1 decodes, storeFGU0 stalls and the storeFGU favor bit is set to 0.

3.1.4.2 Load – Load Hazard

Load-load hazards exist when two instructions are present at decode and both require the LSU (e.g. loads, integer stores). A load favor bit decides which instruction decodes and which instruction stalls.

If the load favor bit is 0, then instruction0 decodes, instruction1 stalls and the load favor bit is set to 1. If the load favor bit is 1, then instruction1 decodes, instruction0 stalls and the load favor bit is set to 0.

3.1.4.3 FGU – FGU Hazard

FGU-FGU hazards exist when two FGU instructions are present at decode. A FGU favor bit decides which FGU decodes and which FGU stalls.

If the FGU favor bit is 0, then instruction0 decodes, instruction1 stalls and the FGU favor bit is set to 1. If the FGU favor bit is 1, then instruction1 decodes, instruction0 stalls and the FGU favor bit is set to 0.

3.1.4.4 Multiply Block Hazard

All multiplies except for FMULS (blocking multiplies) require the hardware multiplier for two back-to-back cycles. Decode establishes a Multiply Block the cycle after a blocking multiply decodes. The Multiply Block prevents any multiply from decoding the cycle after a blocking multiply decodes. This prevents a hardware hazard for the FGU multiplier.

3.1.4.5 PDIST Block Hazard

The PDIST instruction requires three FP sources. The FRF has two read ports. The PDIST instruction accesses the FRF in back-to-back cycles to read all three sources. Decode establishes a PDIST block the cycle a PDIST decodes. The PDIST block prevents any FGU instruction from decoding the cycle after a PDIST decodes. This prevents a hardware hazard on the read ports of the FRF.

3.1.4.6 Two Cycle Execution Hazard

Decode assists in the execution of integer instructions that take two cycles to execute in the EXU. Instructions that take two cycles to execute in the EXU are:

compare-and-swap (CASA)

store doubleword integer (STD)

A CASA instruction takes two cycles to execute within the EXU. The EXU sends rs2 from the IRF to the LSU on the first cycle. The EXU sends rd from the IRF to the LSU on the second cycle. No instruction is allowed to decode after a CASA instruction decodes within a given thread group. This hole permits the read of the rd source of the CASA.

A STD instruction has four integer source operands (rd, rd+1, rs1, and rs2). The IRF has three read ports. OpenSPARC T2 executes STD in two consecutive cycles. The EXU sends rd from the IRF to the LSU on the first cycle. The EXU sends rd+1 from the IRF to the LSU on the second cycle. No instruction is allowed to decode after a STD instruction decodes within a given thread group. This hole permits the read the extra rd+1 source for the STD.

3.1.4.7 Block Store Hazards

Decode creates a block store stall the cycle after the LSU signals a block store read request. This stall remains in effect for eight cycles. Decode provides the store data to the block store during the eight cycles of the stall condition. A block store stall prevents the decode of instructions from either TG0 or TG1. This eases the implementation of block stores on OpenSPARC T2. A complete description of block stores can be found in Section 3.1.4.7, “Block Store Hazards” on page 3-43.

3.1.4.8 DTLB Reloads

Decode creates a DTLB reload stall the cycle after the TLU signals a DTLB reload request. This stall remains in effect for two cycles and prevents the decode of any LSU instruction from either thread group. LSU uses these holes to reload the appropriate DTLB entry upon a DTLB miss and a hardware tablewalk hit.

3.1.4.9 Register Files Write Port Arbitration

The integer and floating-point register files on OpenSPARC T2 each have two write ports: W1 and W2.

All integer instructions that execute in the normal integer or floating-point pipeline use the W1 port of the Integer Register File (IRF). All integer operations that do not fit in the normal integer or floating-point pipe use the W2 port. Operations that use the W2 port include integer loads that miss the dcache and integer divides.

All floating-point instructions that execute in the normal floating-point pipeline use the W1 port of the Floating-point Register File (FRF). All floating-point operations that do not fit in the normal floating-point pipe use the W2 port. Operations that use the W2 port include floating-point loads and floating-point divides.

Integer multiplies, pixel compares, MULSCC, and POPC execute in the FGU and produce integer results. Four cycles after one of these instructions decodes a hole is created in the integer pipe of the originating thread group. A hole is created by blocking the decode of any instruction for one cycle. The integer result is written into the IRF using the W1 port during the FB stage. These FGU integer instructions are pipelined between threads. Completion signals for these operations are generated during the float bypass stage.

TABLE 3-24 Integer Multiply, POPC, MULSCC, and Pixel Compare Timing

Thread State	Ready	Ready	Wait	Wait	Wait	Wait	Wait	Wait	Ready	Ready	Ready
Cancel Pick											
P	Int Mult	Next Inst	Next Inst	Next Inst	Next Inst	Next Inst	Next Inst	Next Inst	Next Inst	Next Inst	
D Read IRF		Int Mult			Hole						Next Inst
E Read FRF			Int Mult			Hole					
FX1/M				Int Mult							
FX2/B					Int Mult						
FX3/W						Int Mult					
FX4							Int Mult				
FX5								Int Mult (Comp to Pick)			
FB											
FW											
M							Hole				
B								Hole			
W									Int Mult		

Integer Register File W2 Arbitration

Integer loads that miss the dcache and integer divides arbitrate for the W2 port of the IRF. Integer divides have the highest priority for the W2 port. The LSU holds integer loads that miss the dcache for one cycle in the event that the load conflicts with a divide for the W2 port. An integer load that hits the dcache cannot conflict with a integer divide. Timing details of these operations are given in Section , "Divide Wait Conditions " on page 3-34.

Floating-point Register File W2 Arbitration

Floating-point loads and floating-point divides share the W2 port of the FRF. Floating-point divides have the highest priority for the W2 port. The LSU holds floating-point loads that miss the dcache one cycle in the event that the load conflicts with a divide for the W2 port. A floating-point load that hits the dcache cannot conflict with a floating-point divide. Timing details of these operations are given in TABLE 3-22.

TABLE 3-25 Branch Mispredict Timing

Before Fetch					Target							
Fetch	Branch					Target						
Cache		Branch					Target					
Pick			Branch	SpecOp1	SpecOp2	SpecOp3 Pick Broadcasts Flush	SpecOp4 flushed	Target				
Decode Read Branch RF				Branch	SpecOp1	SpecOp2	SpecOp3 flushed		Target			
Execute Read FRF					Branch Taken (mispredict)	SpecOp1	SpecOp2 flushed			Target		
Memory Trap Status						Branch	SpecOp1 flushed				Target	
Bypass Load Data Forward							Branch					Target
Writeback								Branch				

In the TABLE 3-25 above, the branch mispredict is detected at the execute stage. The target is fetched the next cycle, so the design has a four cycle mispredict penalty.

TABLE 3-26 Load Synchronization Timing

Before Fetch	Load									Op1 NPC of Load		
Fetch		Load									Op1	
Cache			Load	Spec Op1	SpecOp2	SpecOp3	SpecOp4	SpecOp5	SpecOp6	SpecOp7 flushed		Op1
Pick				Load	SpecOp1	SpecOp2	SpecOp3	SpecOp4	SpecOp5 Pick Broadcasts Flush	SpecOp6 flushed		
Decode Read IRF					Load	SpecOp1	SpecOp2	SpecOp3	SpecOp4	SpecOp5 flushed		
Execute Read FRF						Load	SpecOp1	SpecOp2	SpecOp3	SpecOp4 flushed		
Memory Trap Status							Load	SpecOp1	SpecOp2	SpecOp3 flushed		
Bypass Load Data Forward								Load Misses Dcache (Mispredict)	SpecOp1	SpecOp2 flushed		
Writeback										SpecOp1 flushed		

In the TABLE 3-26 above, the LSU signals load miss at the bypass stage and the pick unit broadcasts the thread flush the next cycle. The NPC of the load is fetched. The thread is not picked until the load data is returned from the miss.

TABLE 3-27 Integer Operation to Integer Operation Timing

P	Int Op0	Int Op1					
D Read IRF		Int Op0	Int Op1 (bypass to here)				
E Read FRF			Int Op0 (bypass from here)	Int Op1			
M				Int Op0	Int Op1		
B					Int Op0 (exception sent here)	Int Op1	
W						Int Op0	Int Op1

The Int Op0 bypasses its result to Int Op1 if it is dependent. The Int Op1 has time to be flushed if the Int Op0 has an exception.

TABLE 3-28 Integer Operation to FGU Operation Timing

P	Int Op	FGU Op									
D Read IRF		Int Op	FGU Op (bypass to here)								
E Read FRF			Int Op (bypass from here)	FGU Op							
M				Int Op							
B					Int Op (exception sent here)						
W						Int Op					
FX1					FGU Op						
FX2						FGU Op					

TABLE 3-28 Integer Operation to FGU Operation Timing *(Continued)*

FX3							FGU Op				
FX4								FGU Op			
FX5									FGU Op		
FB										FGU Op	
FW											FGU Op

The EXU bypasses the result from the Int Op to the FGU Op (if needed), just as it would for a dependent integer operation. If the Int Op has an exception, the FGU Op has sufficient time to be flushed before it updates state.

TABLE 3-29 FGU Operation to Integer Operation Timing

P	FGU Op	Int Op								
D Read IRF		FGU Op	Int Op							
E Read FRF			FGU Op	Int Op						
M					Int Op					
B						Int Op (exception reported)				
W							Int Op			
FX1				FGU Op (predict FGU exception)						
FX2					FGU Op (except ion predict ed)					

TABLE 3-29 FGU Operation to Integer Operation Timing *(Continued)*

FX3						FGU Op (FCC sent here)				
FX4							FGU Op			
FX5								FGU Op		
FB									FGU Op (exception reported)	
FW										FGU Op

In the TABLE 3-29 above, the FGU Op does not generate an integer result, so there is no dependency hazard (excluding FBfcc, MOVCC, and FMOVCC). Since the FGU Op predicts the exception in FX2, the Int Op can be flushed if the FGU Op generates an exception.

TABLE 3-30 FGU Operation to FGU Operation Timing

P	FGU Op	Independent FGU Op	Dependent FGU Op	Dependent FGU Op	Dependent FGU Op	Dependent FGU Op	Dependent FGU Op					
D Read IRF		FGU Op	Indep. FGU Op					Dep. FGU Op				
E Read FRF			FGU Op	Indep. FGU Op					Dep. FGU Op (bypass to here)			
FX1				FGU Op	Indep. FGU Op					Dep. FGU Op		

TABLE 3-30 FGU Operation to FGU Operation Timing

FX2					FGU Op	Indep. FGU Op					Dep. FGU Op	
FX3						FGU Op	Indep. FGU Op					Dep. FGU Op
FX4							FGU Op	Indep. FGU Op				
FX5								FGU Op	Indep. FGU Op			
FB									FGU Op (bypass from here)	Indep. FGU Op		
FW										FGU Op	Indep. FGU Op	

In TABLE 3-31, the red FGU Op depends on the blue FGU Op, and the green FGU Op does not depend on the blue FGU Op. Pick delays the dependent operation to avoid the dependency hazard. As with the integer pipe, there is no exception hazard (subsequent instructions can be flushed).

TABLE 3-31 Floating-Point Load to FGU Operation Timing

P	Load-FGU Op		Dependent FGU Op									
D Read IRF		Load-FGU Op		Dep. FGU Op								
E Read FRF			Load-FGU Op		Dep. FGU Op							
M				Load-FGU Op								
B					Load-FGU Op (bypass from here) (exception sent here)							
W						Load-FGU Op						
FX1						Dep. FGU Op						
FX2							Dep. FGU Op					
FX3								Dep. FGU Op				

TABLE 3-31 Floating-Point Load to FGU Operation Timing *(Continued)*

FX4									Dep. FGU Op			
FX5										Dep. FGU Op		
FB											Dep. FGU Op	
FW												Dep. FGU Op

Pick delays the dependent operation to avoid the dependency hazard. As with the integer pipe, there is no exception hazard (subsequent instructions can be flushed).

Flush Timing Diagrams

TABLE 3-32 Branch Mispredict Flush

C	Branch	Any Op0	Any Op1				
P		Branch	Any Op0	Any Op1 (pick flushes internally)	(pick broadcasts flush)		
D Read IRF			Branch	Any Op0	Any Op1		
E Read FRF				Branch	Any Op0		
M					Branch		
B						Branch	
W							Branch
Mispredict Status to Pick				Branch Taken			
Flush Signals from Pick Orange=Pick duty to flush							

CHAPTER 4

Execution Unit

4.1 Overview

The Execution Unit (EXU) executes all integer arithmetic and logical operations except for integer multiplies and divides. The EXU calculates memory and branch addresses. The EXU handles all integer source operand bypassing.

The EXU is composed of the following subunits:

Arithmetic Logic Unit (ALU)

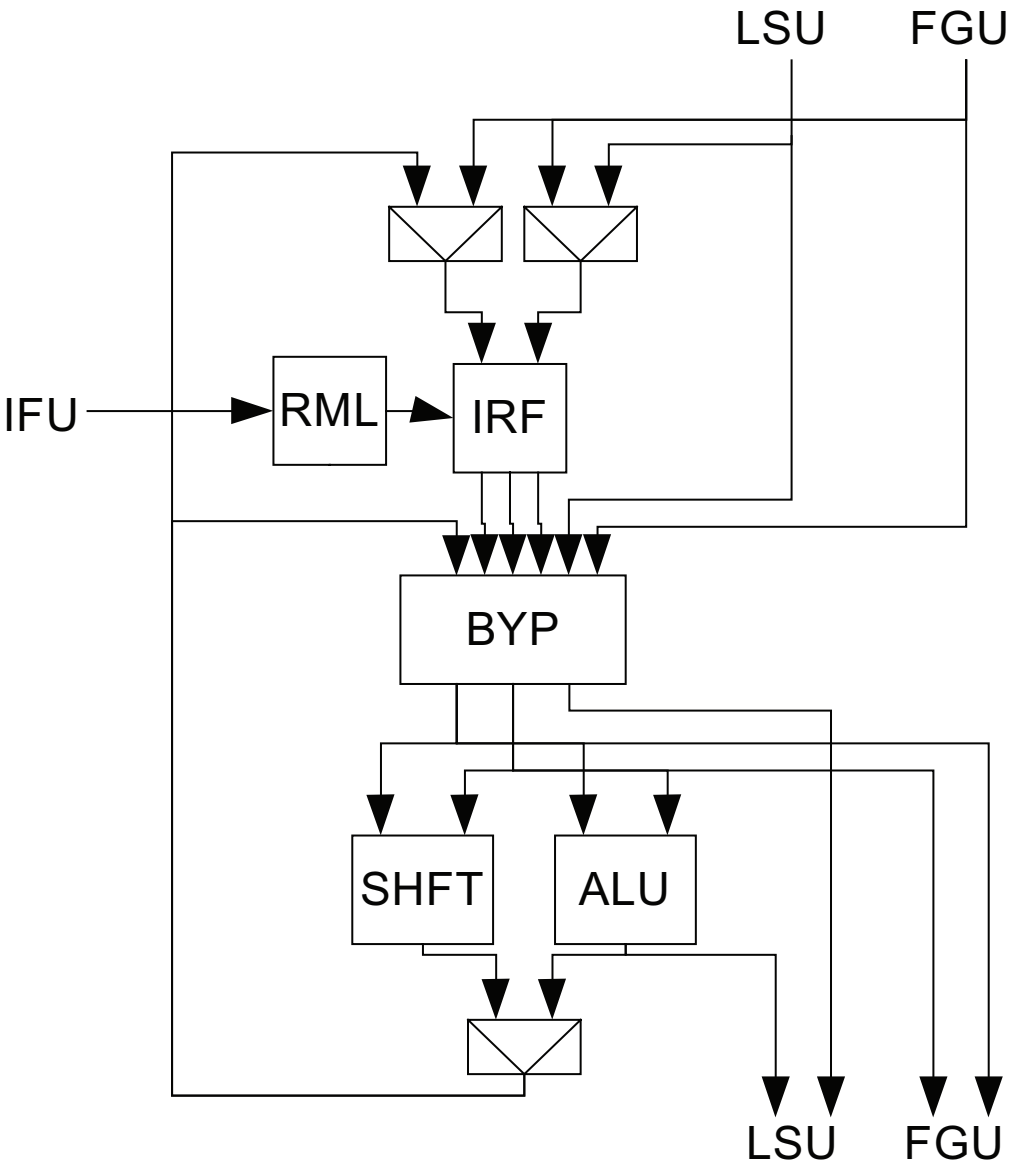
Shifter (SHFT)

Operand Bypass (BYP): rs1, rs2, rs3, and rcc bypassing.

Integer Register File (IRF)

Register Management Logic (RML)

FIGURE 4-1 EXU Block Diagram



4.2 Changes from OpenSPARC T1

4.2.1 B Stage

OpenSPARC T2 has added the B pipeline stage between the M and W stages. This adds an additional set of staging flops in BYP. The operand bypass muxing adds one early port to the rs1, rs2, rs3, and rcc flops above the E stage. The new port requires an additional set of source-destination comparators.

4.2.2 Integer Multiply

The multiplier resides in the FGU. The FGU executes integer multiplies, not the EXU. The EXU reads the IRF for integer multiplies as it does for all other integer instructions, and forwards the operands to the FGU.

4.2.3 Integer Divide

OpenSPARC T2 does not have a dedicated integer divider. The FGU executes integer divides. The EXU reads the IRF for integer divides as it does for all other integer instructions, and forwards the operands to the FGU.

4.2.4 Edge Handling Instructions

As part of VIS 2.0 support, the EXU executes Edge instructions.

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

The twelve forms of the edge instruction include: EDGE8 {L} {N}, EDGE16 {L} {N}, EDGE32 {L} {N}. This supports both left and right edge, as well as big and little-endian.

A 2 bit (EDGE32), 4 bit (EDGE16), and 8 bit (EDGE8) pixel mask is stored in the least significant bits of rd.

4.2.5 Three Dimensional Array Addressing Instructions

As part of VIS 2.0 support, the EXU executes Array instructions.

These instructions convert three dimensional (3D) fixed point addresses contained in rs1 to a blocked-byte address and store the result in rd. Fixed point addresses are typically used for address interpolation for planar reformatting operations. Blocking is performed at the 64 B level to maximize external cache block reuse, and at the 64 KB level to maximize TLB entry reuse, regardless of the orientation of the address interpolation. These instructions specify an element size of 8 (ARRAY8), 16 (ARRAY16), and 32 bits (ARRAY32). The rs2 operand specifies the power-of-two size of the X and Y dimensions of a 3D image array.

4.2.6 BMASK Instruction

As part of VIS 2.0 support, the EXU executes the BMASK instruction.

BMASK adds two integer registers, rs1 and rs2, and stores the result in rd. The least significant 32 bits of the result are stored in the GSR.mask field.

4.2.7 Thread Group Muxing for LSU Address and FGU Operands

An OpenSPARC T2 core contains two instances of the EXU. One instance supports Thread Group 0 (threads 0 through 3) and the other supports Thread Group 1 (threads 4 through 7). In addition to the two EXUs, OpenSPARC T2 has a single Load Store Unit (LSU) and a single Floating-point and Graphics Unit (FGU).

Both EXUs generate memory addresses for the LSU. To minimize global routes, a single mux below the two EXUs provides a single memory address for the LSU.

The FGU executes the following integer instructions:

Integer multiply

Integer divide

Multiply step (MULSCC)

Population count (POPC)

Both EXUs provide operands to the FGU for these instructions. Muxes below the two EXUs provide instruction and integer operand data to the FGU. The EXU formats (e.g., sign extends) integer operand data for the FGU.

CHAPTER 5

Load Store Unit

The OpenSPARC T2 Load Store Unit (LSU) handles memory references between the SPARC core, the L1 data cache, and the L2 cache. All communication with the L2 cache is through the crossbars (processor to cache and cache to processor, a.k.a. PCX and CPX) via the gasket. All SPARC V9 and VIS 2.0 memory instructions are supported with the exception of quad precision floating-point loads and stores.

The LSU ensures compliance with the TSO memory model with the exception of instructions which are not required to strictly meet those requirements (block stores, for example). Like OpenSPARC T1, OpenSPARC T2 does not support an explicit RMO mode.

The LSU is responsible for handling all ASI operations including the decode of the ASI and initiating transactions on the ASI ring. The LSU is also responsible for detecting the majority of data access related exceptions.

5.1 Overview

FIGURE 5-1 LSU Subunits and Dataflow

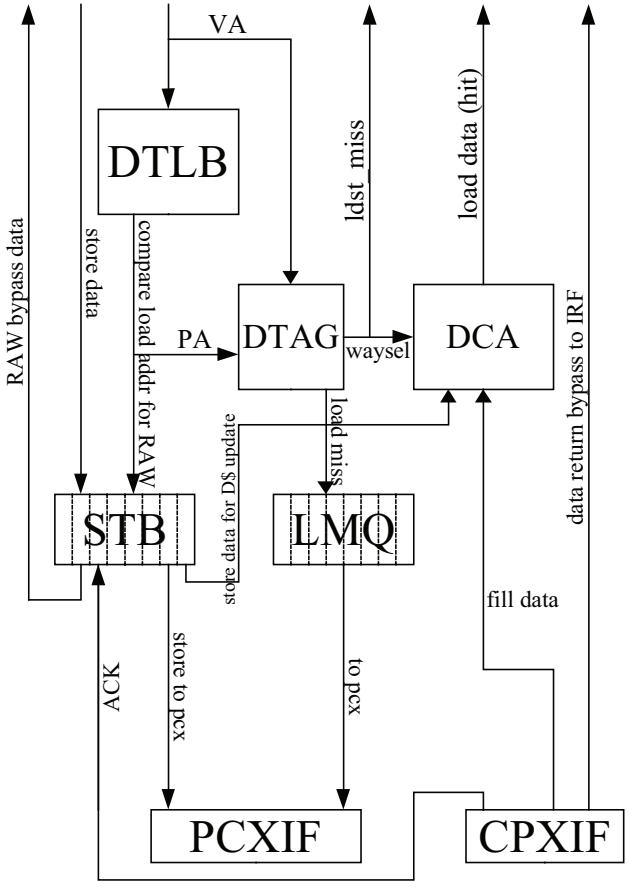


FIGURE 5-1 shows the major functional blocks in the LSU. The DCA and DTAG make up the level 1 data cache. The DTLB provides virtual to physical and real to physical address translation for memory operations. The Load Miss Queue (LMQ) stores the currently pending load miss for each thread (each thread can have at most one load miss at a time). The Store Buffer (STB) contains all outstanding stores. The PCX interface (PCXIF) controls outbound access to the PCX and ASI controller. The CPX interface (CPXIF) receives CPX packets (load miss data, store updates, ifill data, and invalidates), stores them in a FIFO (the CPQ), and sends them to the dcache.

5.1.1 Changes from OpenSPARC T1

- OpenSPARC T2 supports store pipelining (see Section 5.3.4, "Store Buffer (STB)" on page 5-14). OpenSPARC T1 requires any store to receive an ACK before the next store could issue to the PCX.
- The STB supports eight threads.
- The LMQ supports eight threads.
- The DTLB is 128 entries.
- Only load operations access the dcache from the pipeline. This reduces conflicts with the CPQ.
- Partial Store Instructions from the VIS 2.0 ISA are supported.
- Pipeline is E/M/B/W vs. OpenSPARC T1's E/M/W/W2. Pipeline timings are different than Niagara's.
- OpenSPARC T2 has additional RAS features and enhanced error detection and protection.

5.2 LSU Pipeline

The basic LSU pipeline is as follows:

E: The virtual address and store data are received from the EXU. Most control signals from decode arrive in this cycle.

M: The TLB performs address translation. D$tags and data are accessed here. Tag comparison with the PA is performed at the end of the cycle in the TLB. FP store data comes in this cycle.

B: For loads, way select, data alignment, and sign extension done before the load result is transmitted to EXU/FGU. The store buffer is checked this cycle for RAW hazards. For stores, the PA is written into the store buffer and store data is formatted and ECC generated.

W: Store data is written into the buffer. Instructions which were not flushed prior to this point are now committed.

Load data can bypass from the B stage to an instruction in the D stage of the pipeline. This means that a dependent instruction can issue two cycles after a load. If load hit controlled instruction pick, the penalty would be four cycles. The load hit is speculated to reduce the penalty to two cycles. If the load misses and there are instructions from the same thread in the M, E, D, or P state of the pipe, the pipeline flushes (for the missed thread only) and the thread is refetched.

5.2.1 Store

The dcache is write-through, so the LSU sends all stores to the L2. The L2 maintains a copy of the L1 tags for coherency. Hit or miss in the L1 for stores is determined by the L2. Stores which hit the L1 will update the dcache. Stores which miss do not allocate. Cache updates and invalidations for stores occur after the ack has been received from the L2.

All stores within a thread are processed in order. The following diagram shows the best case timing for a store to issue to the L2. Stores must arbitrate against other threads and against loads for access to the L2 (see the PCXIF section for more details). This diagram assumes arbitration is won immediately.

TABLE 5-1 Store Timing – Outbound (no conflicts)

E	Store						
M		Store					
B			Store stb_cam write				
W/ P1				Store stb_ram write valid state set			
P2					arb with other threads' stores for pcx		
P3						stb_cam read stb_ram read load/store arb	
P4							Send store packet

The following hazards will prevent any store from arbitrating for pcx access.

A load in the pipe hit in the STB and will read the STB to bypass data (STB has single read port)

A thread may be reading the STB to initiate a block store.

A store in the pipe is writing the stb_cam (stb_cam cannot read and write simultaneously)

An STD instruction is writing the stb_cam

Another thread is performing a diagnostic read of the STB

The diagram below illustrates the hazard where a store in the pipe blocks a store pending to the L2 from requesting pcx access.

TABLE 5-2 Store Miss - Outbound (hazard on output)

E	Store			Store				
M		Store			Store			
B			Store stb_cam write			Store stb_cam write		
W/ P1				Store stb_ram write valid state set				
P2					arb with other threads' stores for pcx	*hazard*		
P3							stb_cam read stb_ram read load/store arb	
P4								Send store packet

When the L2 sends the store ack, the LSU writes the ack into the CPQ FIFO. (If the FIFO is empty, the packet passes around the queue.) When the ack reaches the head of the CPQ, there are two possibilities. If the ack indicates a cache update is required (if the store hit to the L1 cache) it must wait for a hole to open in the dcache pipe before the update can proceed and the store dequeued from the store buffer. If the store missed the cache and no update is indicated, the store can be immediately dequeued from the store buffer. (Store misses do not allocate in the L1 dcache.) The L2 directory controls allocation since it has the most current copy of the L1 tags and valid status. The allocation information is embedded in the invalidation vector that is part of the store ack packet.

The following diagram shows the timing of a store ack which proceeds immediately upon receipt (i.e., the CPQ is empty and no load is blocking the dcache pipe).

TABLE 5-3 Store Ack Timing - no blocking

E	Store acked							
M		Update D$						
B			STB entry invalidated					

The next diagram shows the timing of a store ack which must update the cache, but is blocked by loads in the pipe.

TABLE 5-4 Store Ack Timing - load blocking

E	Store acked Load	Wait Load	Ok to go					
M		Load	Load	Update D$				
B			Load	Load	STB entry invalidated			

5.2.2 Load Hit

A load that hits the dcache does not make a request to the L2. Floating-point and integer load timings are identical.

TABLE 5-5 Load Hit Pipeline

E	Load			
M		Load		

TABLE 5-5 Load Hit Pipeline

B			Load	
W				Load
	EA calculated in EXU	TLB translation Tag lookup/Data array read Tag compare	Check STB for RAW hazard Way select & Data formatting Send data to EXU/FGU	

5.2.3 Load Miss

On a load miss, a request for data is sent to the L2 via the PCX. The load miss request can be sent as early as W, once the miss has been detected.

When the load data returns from L2, it writes into the CPQ FIFO. (If the FIFO is empty, the packet passes around the queue.) Once a hole opens in the dcache pipe and the register file port is free, the fill and data transfer proceed.

TABLE 5-6 Load Miss Timing (request in B - no hazards on return)

P					...	Op	Op	
D					...			Op
E	Load				...	load return packet arrives lsu_complete to pick		
M		Load			...		Write D$	
B			Load hit/miss detect		...			Data to IRF/FRF
W				Load	...			
W2 /PQ					...			
W3 /PA					...			
			Arb for pcx	Request to gasket				

The load miss path shares the W2 ports of the FGU register file with the divide pipeline. When a divide is near completion, the FGU signals the LSU, causing the data return to stall for one cycle. This creates a hole for the divide to write into the register file. Since divides have higher priority at the W2 register file ports, the LSU buffers data in the CPQ until the W2 port is free. The LSU is the sole source for the W2 port of the IRF, so no such blocking is required for integer loads.

Because the cache arrays are single ported and because there is only one return bus to the register files, load misses can be delayed by other loads in the pipe. As the diagram below shows, a load hit and load miss cannot both enter the pipe simultaneously. Loads from the instruction stream always have priority over load miss returns.

TABLE 5-7 Load Miss Timing (request in B - hazard on return)

P				...	Op	Op	Op	Op	Op	
D				...						Op
E	Load			...	Load return pkt Load	Return blocked Load	Return blocked Load	lsu_complete to pick		
M		Load		...		Load	Load	Load	Write D$	
B			Load hit/miss detect	...			Load	Load	Load	Data to IRF/FRF

5.2.4 RAW Bypass

Each load checks its thread's STB for pending stores. A full read after write (RAW) occurs if the load address matches that of the store and the full data is present. A partial RAW occurs if the load address matches but only partial data is present. (See Section Section 5.3.4, “Store Buffer (STB)” on page 5-14 for more details.) A full RAW bypasses data. A partial RAW is treated as a load miss and is forwarded to the PCX after its corresponding store(s) have been issued to the PCX. Since the RAW bypass cannot fit into the normal pipeline (data ready by B stage), OpenSPARC T2 treats it as a load miss. The pipeline flushes and the IFU refetches subsequent instructions if necessary. The pipe diagram below shows the case of a full RAW.

TABLE 5-8 Full Raw Bypass Timing

BF					Int Op				
F						Int Op			
C							Int Op		
P								Int Op	
D	Int Op								Int Op
E	Load	Int Op					*Load bypass*		
M	Store	Load	Int Op					*Load bypass*	
B		Store write stb_cam	Load RAW hit	Int Op flushed					*Load bypass* data sent to IRF/FRF
W			Store write stb_ram	Load					
W2					Load read STB				
W3						Load STB data available			
W4							LMQ bypass ready		
			lsu_sync				lsu_cmplt		

In the best case, the bypass data is ready at B just in time to bypass to the next instruction in D. However, the RAW bypass data arbitrates for the return path along with load hit data from the dcache and load miss data from the L2. RAW bypass has the lowest priority, so this latency is not guaranteed. The timing of a bypass blocked by a higher priority load is shown below.

TABLE 5-9 Full Raw Bypass Timing With Blocking

BF	Load	Load	Load		Int Op						
F		Load	Load	Load		Int Op					
C			Load	Load	Load		Int Op				
P				Load	Load	Load		Int Op	Int Op	Int Op	
D	Int Op				Load	Load	Load				Int Op
E	Load	Int Op				Load	Load	Load	*Load bypass*		
M	Store	Load	Int Op				Load	Load	Load	*Load bypass*	
B		Store write stb_cam	Load RAW hit	Int Op flushed				Load	Load	Load	*Load bypass* data sent to IRF/FRF
W			Store write stb_ram	Load							
W2					Load read STB						
W3						Load STB data available					
W4							LMQ bypass ready				
			lsu_sync						lsu_cmplt		

5.3 Functional Units

5.3.1 Data Cache (DCA/DTA/DVA/LRU)

The data cache is an 8 KB, 4-way set associative cache with 16 B lines. The DCA array stores the data, the DTAG array stores the tags, the DVA array stores the valid bits, and the LRU array stores the used bits. DCA and DTA are single ported memories. Each line requires a physical tag of 29 bits (40 bit PA minus 11 bit cache index) plus one parity bit.

The dcache is write-through, which implies there are no dirty lines to evict. All stores are sent to the L2 regardless of hit or miss. Only stores that hit the dcache allocate once the ack is returned from the L2.

The dcache is parity protected with one parity bit for each byte of data. Byte parity prevents the need for read-modify-write operations. The tags have one parity bit for the entire 29 bit tag.

If a parity error is detected in the data, tag, or valid bits, all ways of that line are first invalidated. (This is done by sending an invalidation request to the L2, which responds with an invalidate ack. Upon receiving the invalidate ack, all valid bits for the line are cleared.) Once the invalidation is sent, the load is treated like a miss, and data is fetched from the L2. Since the dcache is write-through, no data is lost on a parity error. A disrupting trap will be taken so software can log the error, but hardware continues execution.

Data access related exceptions, including TLB miss, are reported in the B stage of the pipeline along with the load hit or miss. Exceptions and load misses behave identically in terms of pipeline control and flushing.

A load that misses in the dcache can make a request to the gasket in the W stage. Load misses arbitrate for access to the gasket along with stores. When data for the load returns on the CPX, the data bypasses to the pipeline and the dcache fills once a hole opens in the dcache pipe (no load instruction in the pipe).

The LSU performs data alignment, endian ordering, and sign extension before returning data to the IRF and FRF. Loads of less than a doubleword require alignment since the partial data needs to be right-justified in rd. Signed loads of less than a doubleword require sign extension. Unsigned loads of less than a doubleword require zero fill. Atomic operations such as CASA and LDSTUB as well as LDD/LDDA also require zero filling.

The L2 directory manages dcache coherency. The L2 directory maintains a copy of the L1 (dcache and icache) tags and issues invalidation commands as necessary to keep all L1 caches in the system coherent. The coherency methodology also guarantees that a line never exists in the icache and dcache simultaneously. A load or store to an address present in the icache causes that line to be invalidated in the icache. Similarly, an ifetch of a line present in the dcache invalidates that line in the dcache.

Because the L2 directory controls store allocation, store operations do not need to check the tags. The LSU sends all stores to the L2 regardless of dcache hit or miss. By not reading the tag and data arrays for a store, the LSU makes them available to service load miss fills and store updates from the L2.

On a load miss, a replacement way is calculated using an LRU method. The load request to the L2 includes the replacement way so that the L2 can update the directory.

5.3.1.1 Valid Bit Handling

The dcache requires 128 lines x 4 ways = 512 bits of valid data. Valid bits are cleared initially through diagnostic ASI or MBIST.

The array is organized as a 32 by 32 array with bit enabled write input. The line size is double the required 16 because valid bits are duplicated for RAS protection. For a read, address bits [10:6] select the index and bits [5:4] select a group of four bits from 16. For a write, address bits [10:6] select the index and the bit enables and write data control which bits are updated.

A load miss fill sets a single valid bit. An invalidate from the CPQ clears one or more valid bits. Single way invalidations result from another SPARC storing to a shared line or from the icache fetching a line that resides in the dcache. Invalidations caused by an L2 line eviction can clear up to four valid bits in a single cycle (there are four 16 B dcache lines per 64 B L2 eviction).

The valid bit array is dual ported to allow load accesses from the pipe (via the read port) to occur simultaneously with updates and invalidations (via the write port) from the CPQ. This means invalidations from the CPQ occur regardless of what is in the M stage of the pipe.

5.3.2 DTLB

The DTLB is specified in the MMU specification. It is located in the LSU because of its physical proximity and close linkage with the dcache and store buffers.

5.3.3 Load Miss Queue

The LMQ contains loads which have missed the dcache and are waiting on load return data from the L2 or NCU. All internal ASI loads are also placed into the LMQ. The LMQ also holds loads which RAW in the store buffer while they wait for resolution. If the STB holds the complete data, the load bypasses data to the pipe, but it does not allocate in the dcache. If only partial data is available, the load is treated as a miss and is forwarded to the gasket after the store with the common address. In either case, the load is treated as a dcache miss from a scheduling perspective.

A load may RAW against multiple stores to the same line in the STB. In this case, the load is treated as a miss in the same manner as a partial raw and is forwarded to the PCX once all stores in the STB have issued to the pcx. No bypass occurs.

The LMQ contains one entry for each thread since only one load can be outstanding at any time for a given thread.

When multiple threads are waiting for access to the gasket, the thread with the oldest miss is granted access. (This uses a psuedo algorithm, so perfect age ordering is not guaranteed.)

The LMQ checks for common addresses for load misses across threads to prevent duplication of tags in the dcache. The LMQ compares an incoming load from any thread against all valid entries in the LMQ. If there is a match, the incoming load is termed a secondary miss. Secondary misses cause a request to the L2, but they are marked as non-cacheable to prevent cache pollution.

5.3.4 Store Buffer (STB)

All store instructions and instructions that have store semantics (atomics, wrsr, wrpr, wrhpr) are inserted into the STB after address translation through the DTLB, assuming the stores do not generate an exception. The STB is threaded and contains eight entries per thread.

All loads check the store buffer (same thread only) for read after write (RAW) hazards. A full RAW occurs when the dword address of the load matches that of a store in the STB and all bytes of the load are valid in the store buffer. A partial RAW occurs when the dword addresses match, but all bytes are not valid in the store buffer. (Ex., a ST (word store) followed by an LDX (dword load) to the same address results in a partial RAW, because the full dword is not in the store buffer entry.)

Stores are issued in program order (per thread) to the gasket. There is no implied ordering across threads. A pseudo-LRU algorithm selects which thread to issue. The store remains in the STB until an acknowledgment is received via the CPX from the L2 and the dcache is updated with the store data (if necessary). All STB entries

continue to be RAW checked until they are dequeued. Acknowledgement and dequeue/cache update are two separate events. Acknowledgement occurs when the store ack is visible at the input of the CPQ. The sole purpose of acknowledgement is to verify that the L2 has processed the store and it is globally observable. Once a store has been acknowledged, the store following can issue. Dequeue and cache update/invalidate occur when the store reaches the head of the CPQ. At this time the dcache is updated or invalidated as necessary and the store is removed from the store buffer. After this point, the result of the store is visible to all threads that share the dcache.

RMO stores follow different ordering rules and are thus handled differently. Instead of waiting for the acknowledgment from L2, RMO stores dequeue once they are issued to the gasket. An RMO store never causes a dcache update. If an RMO store hits a line that is resident in the dcache, it will be invalidated once the acknowledgment returns from L2. A side count is kept of the number of RMO stores outstanding. This is required so that it is known when the store buffer is truly emtpy. Synchronizing instructions such as flush and membar must wait until the stb is empty before releasing the thread. The store buffer is considered emtpy when there are no valid entries in the buffer AND there are no outstanding RMO stores.

Stores from the same thread to the same L2 line are pipelined. If the L2 cache line address of the entry in the 2nd position of the STB matches that at the head of the STB, that entry issues to the gasket without waiting for an acknowledgment from the L2. Then, if the 3rd entry matches the 2nd, it issues as well, etc. An entry whose L2 line differs from the entry ahead of it must wait for the previous store's ack before it can be issued to the gasket. Internal ASI stores can be pipelined as long as they are to the same ASI ring. Stores to IO never pipeline. The rule for store pipelining are summarized as follows:

An ASI following a memory store cannot pipeline (and vice-versa)

RMO stores following non-RMO stores cannot pipeline (and vice-versa)

Stores to IO including IO mapped ASI stores cannot pipeline

To prevent problems in the STB state machine, RMO stores are not allowed to pipeline behind non-RMO stores.

PKU keeps track of the number of stores in the STB plus the number of stores in the pipe to ensure that the STB never overflows. This tracking is done per thread. The STB notifies PKU when an entry is deallocated from the STB.

5.3.5 PCX Interface (PCXIF)

The PCX interface, in conjunction with the LMQ and STB, arbitrates between load and store requests and manages outgoing packets.

The arbitration between loads and stores is done as follows. The goal is to minimize load miss latencies while avoiding store buffer full occurrences. To achieve that, a weighted favor system is used.

Loads will be favored over stores by default.

If a store has been waiting for 4 cycles, it will be favored

If any thread's store buffer is full, stores have favor every other cycle.

Stores are eligible to begin requesting for access once they reach the W stage and are inserted into the STB. Load misses are committed into the LMQ and can begin requesting access once they reach the W stage of the pipe. If no load in the LMQ requires access to the pcx, a load in the pipe is allowed to be sent early. Because the physical address and cache miss status is not known until B, that is the earliest that a load can request access. The B stage of the load corresponds to the P3 arbitration cycle.

The arbitration timing is shown in TABLE 5-10.

TABLE 5-10 PCX Arbitration Timing

P1	Each thread's STB determines which entry is ready			
P2		STB picks one thread for store issue (if any are ready) LMQ picks one thread for load issue (if any are ready)		
P3			Store buffer is read Arbitration between load and store performed	
P4				Load or store packet sent to gasket

The PCXIF is responsible for adhering to the gasket packet protocol. The gasket contains a two entry FIFO for storing request packets. A grant will be sent back in the cycle following a request at the earliest. It is the responsibility of the PCXIF to ensure that requests are not sent while the FIFO is full.

CASA requests require two packets, one to send the compare data and one to send the swap data. These packets must be sent back to back. To simplify the arbitration logic, a CASA will not be allowed to issue until both entries in the gasket are available. To prevent a livelock case where loads and stores (which only require one free entry in the gasket) prevent a CASA from proceeding, all accesses must wait for the gasket FIFO to have both entries free once a thread is trying to issue a CASA. This guarantees that the CASA will eventually be selected.

5.3.6 CPX Interface (CPXIF)

The CPX interface monitors all packets from the CPX. Packets destined for units other than the LSU ignored by this interface. The CPXIF receives all CPX packets that affect the dcache. This includes load returns, store acknowledgments, and invalidation requests.

The cache to processor queue (CPQ) is a 32 entry FIFO which stores all incoming packets. All entries are processed in order and are non-threaded. This maintains memory ordering with minimal complexity. Only the packet at the head of the queue is eligible for processing. Because the dcache is a single ported array, and because there is only one data bus to the register files, hardware hazards must be avoided. If the packet requires data be sent to a register file (all loads) or that the dcache be written (cacheable loads and store updates), then that packet may only be processed when there is no load in the pipe at the E stage (since loads in the pipe require reading the cache and send data to the register files on hits).

If the FIFO reaches the high-water mark of 15, the LSU signals to the decode unit to stall. By blocking load instructions from the pipe, this guarantees that the CPQ can drain. The high water mark is set to account for instructions in the pipe and to allow for outstanding block loads.

As an additional guard against livelock, a counter is maintained which counts cycles in which an entry at the head of the CPQ is not allowed to proceed due to a load in the pipe. Once an entry has waited for 8 cycles, a stall request is sent to the decode. Decode will block any load or store instruction from entering the pipe, thus opening a hole for the return packet to proceed.

The CPX interface is also responsible for decoding invalidation vectors and forwarding the appropriate information to the dcache. Invalidations only access the valid array which is dual-ported, so they can proceed regardless of what is in the pipeline.

5.4 Special Memory Operation Handling

Special memory operations are those that do not conform to the standard pipeline or require additional functionality beyond standard loads and stores.

5.4.1 CASA and CASXA

Compare and Swap instructions have load and store semantics. The value in rs2 is compared with the value in memory at [rs1]. If the values are the same, the value in memory is swapped with the value in rd. If the values are not the same, the value at [rs1] is loaded into rd, but memory is not updated.

CASA assumes a dcache miss. The IFU treats this like any other load miss.

Both sets of data are sent from the EXU in the E cycle. The LSU internally takes two cycles to align the data (each takes a cycle to save formatting logic). Decode will guarantee that no other instruction comes on the second cycle of the CASA. The LSU creates an entry in the store buffer containing the compare (rs2) value. At the same time, a load miss is inserted into the LMQ. A buffer in the LMQ holds the swap data (rd). CAS instructions are sent to the L2 as atomic two packet transactions. The first packet contains the compare data and address, the second packet contains the swap data. The L2 responds with a load return packet whose data is sent to the IRF (no L1 allocation). The L2 also follows up with a store acknowledgment based on the results of the compare. If the line was swapped and it was resident in the dcache, that line will be invalidated.

The thread is restarted only upon receiving the store ack.

5.4.2 LDSTUB, LDSTUBA, and SWAP

Load and Store Unsigned Byte and Swap instructions have load and store semantics. A byte from memory is loaded into rd and the memory value replaced with either all 1's (for LDSTUB) or the value from rd (for SWAP).

These instructions assume a dcache miss. The IFU treats them like any other load miss.

The LSU creates an entry in the store buffer containing the swap value (rd or all 1's). At the same time, a load miss is inserted into the LMQ. A packet containing the address and swap data is sent to the L2. The L2 responds with a load return packet

whose data is sent to the IRF (no L1 allocation). The L2 also follows up with a store acknowledgment. Like CASA, if the line stored is in the dcache, that line will be invalidated.

The thread is restarted only upon receiving the store ack.

5.4.3 Atomic Quad Loads

Atomic quad load instructions load 128 bits of data into an even/odd register pair. Since there is no path to bypass 128 bits of data to the IRF, atomic quads force a miss in the L1 cache. One 128 bit load request is made to the L2 cache.

The return data is written to the IRF over two cycles, once for the lower 64 bits and once for the upper 64 bits. Load completion is signaled on the second write. The load does not allocate in the L1 cache.

5.4.4 Block Loads and Stores

Block loads and stores are loads and stores of 64 B of data. Memory ordering of block operations is not enforced by the hardware – i.e., they are RMO. A block load requires a MEMBAR #sync before it to order against previous stores. A block store requires a MEMBAR #sync after it to order against following loads.

Block loads and stores force a miss in the L1 cache and they do not allocate.

5.4.4.1 Block Loads

Block loads are handled internally by the LSU as 4 loads of 128 bits each. Upon receiving the block load instruction, the LSU signals a dcache miss to the IFU during the B stage. The IFU treats block loads the same as any other load that misses.

The LSU will send four load requests to the L2. Since these are all to the same 64B cache line, they are guaranteed to remain in order.

The LSU sends data to the FRF as it comes back from the L2 subject to the W2 arbitration guidelines detailed in Section 5.2.3, “Load Miss” on page 5-8. Each load packet must be processed twice, once to send the lower 64 bits of data and once to send the upper 64 bits of data. (Each of the four load returns is treated identically to an atomic quad load.) Once the last dword of data is written to the FRF, the LSU signals load completion to the IFU.

5.4.4.2 Block Stores

The datapath between the FGU and LSU and the PCX store datapath is 64 bits. A block store is executed as eight 64 bit store operations in order to store 64 bytes of data. The LSU decodes the ASI of each store to determine whether it is a block store or not. When the LSU encounters a block store at the B pipe stage, it signals a LSU synchronization to the IFU. This effectively flushes all instructions after the block store and transitions the relevant thread to WAIT state. The thread remains in this state until the LSU signals that the block store has completed or the thread is flushed.

The LSU ensures the following prior to executing a block store instruction:

- the store buffer is empty in order to guarantee room for the 8 store operations that comprise the block store
- no prior FGU op exists that may create a hazard with the sources of the block store

The LSU avoids prior FGU dependency hazards by waiting 3 cycles after the LSU synchronization before requesting a block store read from the decode unit. Decode creates a block store stall the cycle after the LSU signals a block store read request. A block store stall prevents the decode of instructions from either TG0 or TG1. This stall remains in effect for 8 cycles. Decode generates sequential register addresses to the FRF over 8 consecutive cycles. The FGU reads the 64 bytes of store data over 8 consecutive cycles and sends it to the LSU. The LSU generates eight 8 byte store operations and writes each into the STB.

Block stores operate under relaxed memory order (RMO), so entries are retired from the store buffer as soon as they are sent to the PCX.

The LSU signals complete to the IFU the same cycle a block store read request is made.

If the FRF encounters an ECC error on any of the 8 data reads, the entire block store operation will be cancelled and a precise trap will be taken. This means the store buffer will not begin sending stores to the L2 until all eight dwords are known to have been read without error.

TABLE 5-11 shows a timing diagram for block stores.

TABLE 5-11 Timing Diagram for Block Stores (assume STB empty)

BF								Any Op0												
F									Any Op0											
C										Any Op0										
P	Any FGU Op	Block Store									Any Op0	Any Op1	Any Op1	Any Op1	Any Op1	Any Op1				
D Read IRF		Any FGU Op	Block Store									Any Op0	Any Op0	Any Op0	Any Op0	Any Op0	Any Op0	Any Op0	Any Op1	
Block Store Stall										Stall All Decode	Stall All Decode	Stall All Decode	Stall All Decode	Stall All Decode	Stall All Decode	Stall All Decode	Stall All Decode			
E Read FRF			Any FGU Op	Block Store							FRF Rd0	FRF Rd1	FRF Rd2	FRF Rd3	FRF Rd4	FRF Rd5	FRF Rd6	FRF Rd7	Any Op0	Any Op1
M/FX1				Any FGU Op	Block Store							FRF Rd0	FRF Rd1	FRF Rd2	FRF Rd3	FRF Rd4	FRF Rd5	FRF Rd6	FRF Rd7	
LSU Sync Flush																				
LSU Sync						lsu_sync							lsu_complete							
Request Block Store Read								lsu_block_store_stall	p1	p2	p3	p4								
B/FX2					Any FGU Op	Block Store			STB CAM read				FRF Rd0	FRF Rd1	FRF Rd2	FRF Rd3	FRF Rd4	FRF Rd5	FRF Rd6	FRF Rd7
W/FX3						Any FGU Op								FRF Rd0	FRF Rd1	FRF Rd2	FRF Rd3	FRF Rd4	FRF Rd5	FRF Rd6
FX4							Any FGU Op													

TABLE 5-11 Timing Diagram for Block Stores (assume STB empty) *(Continued)*

FX5								Any FGU Op											
FB									Any FGU Op										
FW										Any FGU Op									

5.4.5 FLUSH

The IFU postsyncs FLUSH instructions, so no LSU synchronization is necessary. Once all stores prior to the FLUSH have been committed, which implies all previous stores have been ack'ed and necessary invalidations performed, the LSU signals the TLU to redirect the thread to the instruction following the FLUSH via a trap sync.

Because hardware enforces icache/dcache exclusivity, any stores to an address in the icache are automatically invalidated. Therefore, the FLUSH instruction doesn't actually do anything to the caches. It acts solely as a synchronization point, much like MEMBAR.

5.4.6 MEMBAR

MEMBAR (all forms) and STBAR are all executed identically.

MEMBAR instructions behave identically to the FLUSH. The IFU postsyncs the instruction, so no LSU synchronization is required. Once all stores for that thread have been committed, the LSU signals the TLU to redirect the thread to the instruction following the MEMBAR via a trap sync.

5.4.7 PREFETCH

Prefetch instructions load data into the L2 cache, but do not update the L1 caches.

When the LSU receives a prefetch instruction, it signals LSU synchronization to the IFU and inserts the entry into the LMQ. A load request packet is sent to the L2 with the prefetch indicator asserted. Once the packet is sent to the PCX, lsu_complete can be signaled and the entry in the LMQ retired. The L2 does not return any data.

Except when used with illegal function codes, PREFETCH instructions do not cause exceptions, including mmu miss exceptions. If the PREFETCH encounters an exception condition, it will be dropped.

CHAPTER 6

Cache Crossbar

The cache crossbar (CCX) connects the 8 SPARC cores to the 8 banks of the L2 cache. An additional port connects the SPARC cores to the IO bridge. A maximum of 8 load/store requests from the cores and 8 data returns/acks/invalidations from the L2 can be processed simultaneously.

The crossbar is divided into two separate pieces, the processor to cache crossbar (PCX) and the cache to processor crossbar (CPX). Sources issue requests to the crossbar. These requests are queued to prevent head of the line blocking. The crossbar queues requests and data to the different targets.

Since multiple sources can request access to the same target, arbitration within the crossbar is required. Priority is given to the oldest requestor(s) to maintain fairness and ordering. Requests appearing to the same target from multiple sources in the same cycle are processed in a manner that does not consistently favor one source.

Other than changing the number of ports, the cache crossbar does not differ significantly from N1.

6.1 Functional Description

Each crossbar is essentially a N x M bussed mux structure. For the PCX, N=8 (SPARC cores) and M=9 (8 L2 banks + IO). For the CPX, N=9 and M=8.

The eight L2 banks are addressed interleaved. Given 8 banks and a 128 B line size, this means L2 bank selection is performed by bits <8:6> of the physical address.

For each destination point on the crossbar there is arbitration and steering that is independent of the other destination points. Each source-target pair has a queue for data packets. This queue depth is 2 to allow for atomic (two packet) transactions.

For each cycle that a source requires access to a target it asserts a request. In the cycle immediately following the request, it sends its data packet to the crossbar. The crossbar arbitrates among the sources and sends a grant back to the source that wins arbitration. It is the responsibility of each source to use this grant signal to track the state of the packet queues. Sources do not issue requests or packets when these queues are full as packets are dropped under these conditions. (See speculation discussion in the timing section.)

FIGURE 6-1 PCX Slice and Dataflow

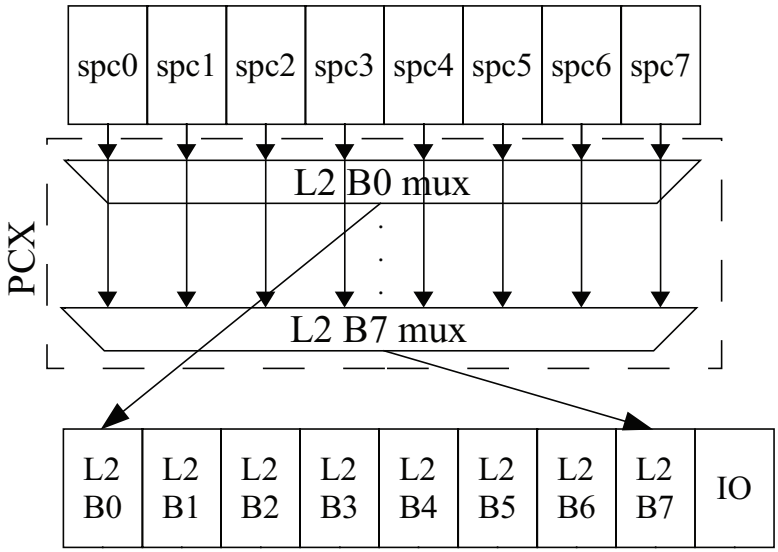


FIGURE 6-1 shows an example of the destination point muxes in the PCX and the associated dataflow.

6.1.1 Timing

The basic crossbar pipeline is defined as three cycles, request, arbitration, and transmit. In reality, there are multiple transmit cycles because of the distances involved in transmitting the packets to the destinations.

The following shows pipeline actions for the PCX. The CPX is similar except that the L2 banks are the requesters and the SPARC cores are the targets. CPX stages are named CQ, CA, and CX.

TABLE 6-1 PCX Pipeline

PQ	*PA*	*PX*
SPARC cores issue requests	SPARC cores send packets to PCX Queue the packets Arbitration for target Send the grant to the muxes	Transmit grant to SPARC core Perform data muxing

The PCX needs one additional cycle, PX2, to drive the packet to the appropriate L2 bank. The CPX needs two additional cycles, CX2 and CX3. Packets returning to the SPARC core must reach multiple destinations within the core and must undergo predecoding. If the chip floorplan causes the distances between SPARC cores, L2 sctag blocks, and the crossbar to increase, additional cycles may be required.

Without additional functions, the timing of the requests and grants in this protocol does not support non-speculative pipelining. As shown in TABLE 6-2 the grant for the request issued in cycle 1 does not return until cycle 3. The grant comes late enough such that the source cannot use it to generate another request. Therefore, since two ungranted requests have been issued prior to cycle 3, the queue is full and another request cannot be sent.

TABLE 6-2 Request and Grant Signal Timing

Cycle	*1*	*2*	*3*	*4*	*5*	*6*
Request	A	B		C	D	
Data		A	B		C	D
Grant			A	B		C
FIFO count		1	2	1	1	2

To achieve full pipeline capability, sources are allowed to speculatively issue requests to the crossbar when the 2 entry queue is full. Looking at TABLE 6-2, we can see that if we send request C in cycle 3, we'll know by cycle 4 whether or not the queue had a free entry. If no grant was received in cycle 3 (which would be processed in cycle 4) then we know that the queue is still full. If the queue is still full, the request can be ignored and the packet dropped. But, as long as the source maintains the data, it can resend the request and data in later cycles.

TABLE 6-3 shows the new timing with the speculation. Since grant A was received in cycle 3, we know that request and packet C were accepted. However, since no grant was received in cycle 4, request and packet D were dropped and must be resent.

TABLE 6-3 Request and Grant Sequence Showing Speculative Request

Cycle	*1*	*2*	*3*	*4*	*5*	*6*
Request	A	B	C_{spec}	D_{spec}		
Data		A	B	C	~~D~~	
Grant			A	-	B	
FIFO count		1	2	2	2	1

6.1.2 Arbitration

The arbitration requirements of the PCX and CPX are identical except for the numbers of sources and targets that must be handled. The CPX must also be able to handle multicast transactions. To facilitate reuse, the arbitration logic is designed as a superset which can handle PCX or CPX functionality.

The arbiter performs the following functions:

Queue transactions from each source to a depth of 2.

Issue grants in age order, with oldest having highest priority.

Resolve requests of the same age without persistent bias to any one source.

Have the ability to stall grants based on input from the target.

Stall the source if the queue is full.

Handle two packet transactions atomically.

Each target has its own arbiter. All targets can arbitrate independently and simultaneously.

An arbiter receives a request packet of M bits every cycle, where M is the number of sources. It also receives M bits which indicate whether the request is for an atomic (2 packet) transaction. This packet is processed until all requests are granted. The arbiter then moves on to the next packet. There can be from 0 to M requests in any cycle. Since one request per cycle is granted, and requests can be for 2 cycle transactions, processing a request packet can take from 1 to 2M cycles. Valid request packets received while a previous packet is being processed are placed into the request FIFO.

The FIFO structure ensures packets are processed in age order. Within a packet, if multiple sources have outstanding requests, arbitration is performed by alternating ascending and descending priority. This eliminates a bias toward any single source.

FIGURE 6-2 Crossbar Arbitration.

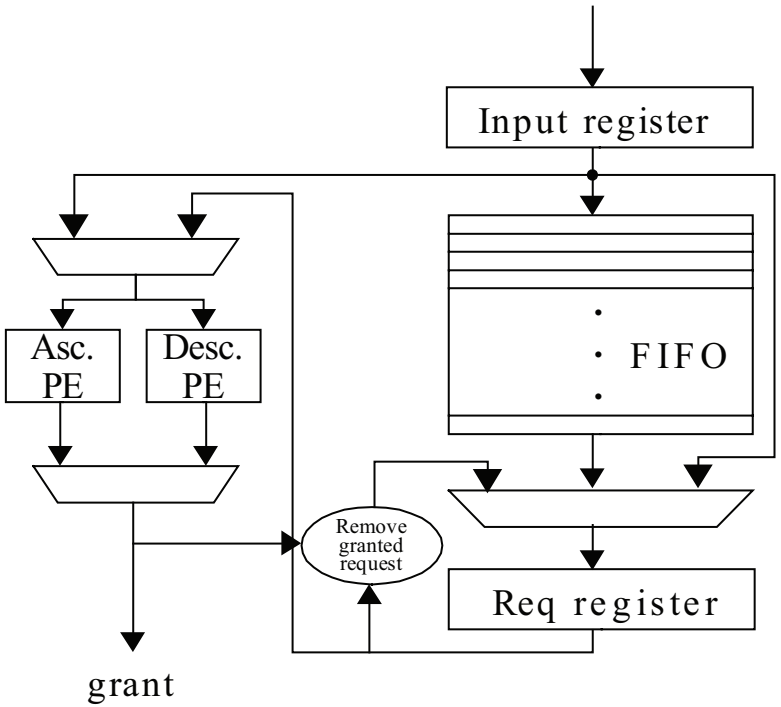


FIGURE 6-2 shows how arbitration is performed. All request vectors into the crossbar are flopped into the input register. The logic on the left side performs the arbitration, alternating between the ascending and descending priority encoders. The source for the PEs is either the request register or the input register. The input register is chosen if both the FIFO and request register are empty. The request register loads from three sources in the following priority order:

The request vector currently being processed.

1. Request vectors are processed until all requests have been granted.
2. The FIFO
3. The input register.

The FIFO depth is set at 2xM entries. For the CPX, M=9, so the FIFO depth is 18. This handles the worst case situation where all sources issue atomic requests simultaneously. In this case, 2xM entries are required to hold later incoming requests while the "full" packet is processed to completion.

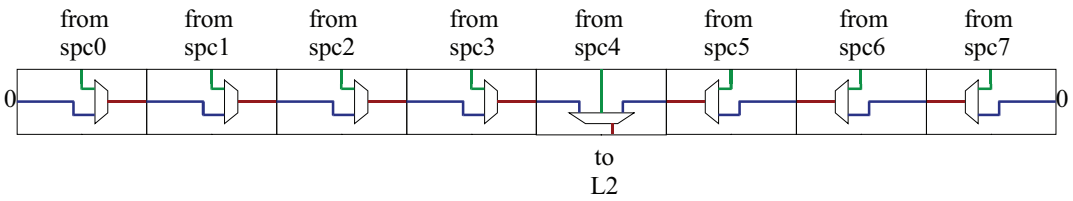
6.2 Datapath

Each target is associated with a datapath slice. A datapath slice steers data from one of M sources to one target. It does so based on the grant signals from the arbiters.

While the function of a datapath slice is essentially a M:1 mux, bringing together M busses, each of over 100 bits, to a common point for muxing is not physically practical. Therefore, the slices are composed of 2:1 and 3:1 muxes that pick data from a "current" source or a neighboring source. Datapath slices also contain the 2 entry packet queue for each source-target pair.

FIGURE 6-3 shows the muxing portion of one PCX datapath slice. Each source feeds one mux. If that source receives the grant, its data passes through the mux. Otherwise, the neighboring mux is chosen.

FIGURE 6-3 PCX Datapath Slice



6.3 Packet Formats

TABLE 6-4 PCX Packet Formats

Instruction	*129*	*128:124*	*123*	*122:120*	*119:117*	*116*	*115*	*114*	*113:112*	*111:104*	*103:64*	*63:0*
	valid	*rqtyp*	*nc*	*cpu_id*	*thread_id*	*inv*	*pf*	*bis/l1way*	*l1way*	*size*	*addr*	*data*
Load	1	00000	V	V	V	0	0	0	way<1:0>	-	V	-
Prefetch	1	00000	1	V	V	0	1	0	00	-	V	-
Diagnostic Load	1	00000	0	V	V	0	0	0	00	-	V	-
Dcache Invalidate	1	00000	0	V	V	1	0	0	00	-	V	-
Instruction Fill	1	10000	V	V	V	0	0	way<2>	way<1:0>	-	V	-
Icache Invalidate	1	10000	0	V	V	1	0	0	00	-	V	-
Store	1	00001	V	V	V	0	0	0	00	V	V	V
Block Init Store	1	00001	V	V	V	0	0	1	00	V	V	V
Diagnostic Store	1	00001	0	V	V	0	0	0	00	V	V	VDS
CAS (1)	1	00010	1	V	V	0	0	0	00	V	V	V
CAS (2)	1	00011	1	V	V	0	0	0	00	V	V	V
Swap/Ldstub	1	00110	1	V	V	0	0	0	00	V	V	V
Stream Load	1	00100	1	V	V	0	0	0	00	-	V	-
Stream Store	1	00101	1	V	V	0	0	0	0	V	V	V
MMU load	1	01000	1	V	V	0	0	0	0	-	V	-
Interrupt	1	01001	0	target	V	0	0	0	00	-	-	VINT

size is an 8 bit byte mask which indicates which of the 8B of store data should be updated

From the L2 perspective, load, stream load, and MMU load are all identical (stream and MMU loads will always be non-cacheable in L1).

TABLE 6-5 CPX Packet Formats

Instruction	*145*	*144:141*	*140*	*139:138*	*137*	*136:134*	*133*	*132:131*	*130*	*129*	*128*	*127:0*
	valid	*rtntyp*	*l2miss*	*err*	*nc*	*thr_id*	*wv*	*way*	*F4B/way*	*atomic*	*pf*	*data*
Load Return	1	0000	V	V	V	V	V	iway<2:1>	iway<0>	0	0	V
Prefetch Return	1	0000	V	V	1	V	V	iway<2:1>	iway<0>	0	1	V
Diagnostic Load Return	1	0000	0	0	0	V	0	-	-	0	0	V
Dcache Invalidate Ack	1	0100	-	0	0	V	-	-	-	0	0	V
IFill Return (1)	1	0001	V	V	V	V	V	dway<1:0>	0	0	0	V
IFill Return (2)	1	0001	0	V	V	V	V	dway<1:0>	0	0	0	V
Icache Invalidate Ack	1	0100	-	0	0	V	-	-	-	0	0	V
Store Ack	1	0100	V	V(pst)	V	V	-	-	-	0	0	V_{ACK}
Block Init Store Ack	1	0100	V	0	V	V	-	-	-	0	0	V_{ACK}
Diagnostic Store Ack	1	0100	0	0	0	V	-	-	-	0	0	V_{ACK}
CAS Return	1	0000	-	V	1	V	0	-	-	1	0	V
CAS Ack	1	0100	-	0	1	V	-	-	-	1	0	V_{ACK}
Swap/Ldstub Return	1	0000	-	V	1	V	0	-	-	1	0	V
Swap/Ldstub Ack	1	0100	-	0	1	V	-	-	-	1	0	V_{ACK}
Stream Load Return	1	0010	V	V	1	V	0	-	0	0	0	V
Stream Store Ack	1	0110	0	V(pst)	V	V	0	-	0	0	0	V_{ACK}

TABLE 6-5 CPX Packet Formats *(Continued)*

Interrupt Return	1	0111	0	0	0	V	-	-	-	-	0	V
Eviction Invalidation	1	0011	0	0	-	-	-	-	-	-	0	V_{INV}
Error Indication	1	1100	0	V	-	V	-	-	-	-	-	-

Load return and like packets always return 16B of data.

The tables below show the detail of the data fields for V_{ACK} and V_{INV}.

TABLE 6-6 Store Ack Data Field (V_{ACK})

127:126	***125***	***124:123***	***122:121***	***120:118***	***117:112***	***111:105***	***104***	***103:96***	***95:64***	***63:0***
0	BIS	I$ inval, D$ inval (inval all sets of the given line)	addr[5:4]	cpu_id	addr[11:6]	0	addr[3]	byte mask	inval. vector 31:28 CPU7 – xx00: no inval ww10: D$inval www1: I$inval 27:24 CPU6 ... 3:0 CPU0	data

The data, address, byte mask, and bis (block init store) fields reflect the same information as was received in the pcx packet.

The I$inval and D$ inval bits are asserted on invalidation requests from the SPARC core. These indicate that all ways of a line are to be invalidated.

TABLE 6-7 Invalidation Packet Data Field (V_{INV})

127:118	***117:112***	***111:88***	***87:56***	***55:32***	***31:0***
0	addr[11:6]	inval. vector for addr[5:4] = 11 111:109 CPU7 – xx0: no inval ww1: D$inval 108:106 CPU6 . . 90:88 CPU0	inval. vector for addr[5:4] = 10 87:84 CPU7 – xx00: no inval ww10: D$inval www1: I$inval 83:80 CPU6 . . 59:56 CPU0	inval. vector for addr[5:4] = 01 55:53 CPU7 – xx0: no inval ww1: D$inval 52:50 CPU6 . . 34:32 CPU0	inval. vector for addr[5:4] = 00 31:28 CPU7 – xx00: no inval ww10: D$inval www1: I$inval 27:24 CPU6 . . 3:0 CPU0

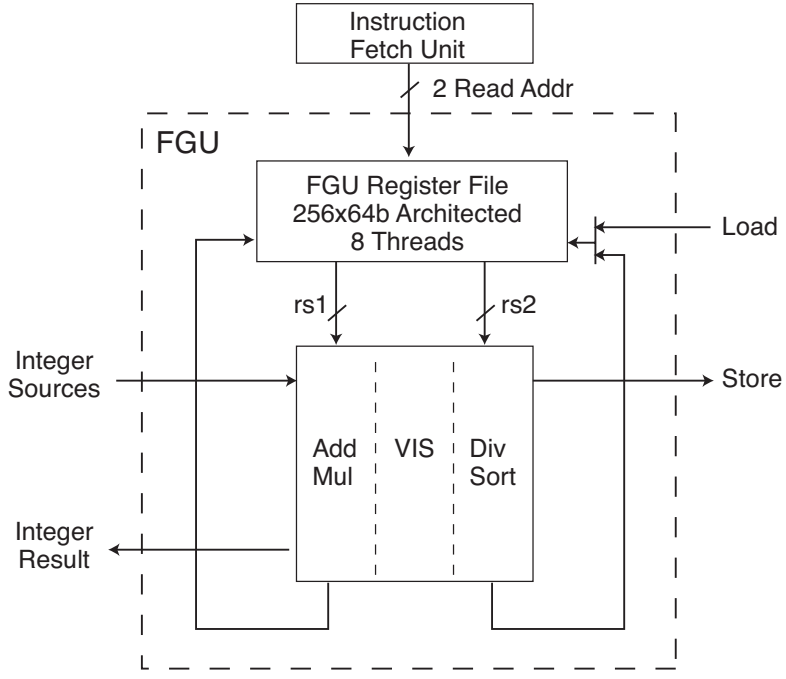
Invalidation packets are sent from the L2 to the cores when a line being evicted from the L2 requires that line(s) be evicted from the L1 caches to maintain inclusion.

CHAPTER 7

Floating Point Unit

7.1 Overview

FIGURE 7-1 FGU Block Diagram



- The OpenSPARC T2 floating-point and graphics unit (FGU) implements the SPARC V9 floating-point instruction set, the SPARC V9 integer multiply, divide, and population count (POPC) instructions, and the VIS 2.0 instruction set, with the following exception:
  - All quad precision floating-point instructions are unimplemented (including LDQF{A} and STQF{A})
- OpenSPARC T2 contains one dedicated FGU per core.
- Compliant with the IEEE 754 standard.
- Support for IEEE 754 single precision (SP) and double precision (DP) data formats. All quad precision floating-point operations are unimplemented.
- Support for all IEEE 754 floating-point data types (normalized, denormalized, NaN, zero, infinity). Certain denormalized operands or expected results may generate an unfinished_FPop trap to software, indicating that the FGU was unable to generate the correct results. The conditions which generate an unfinished_FPop trap are consistent with UltraSPARC I/II.
- FGU includes three execution pipelines:
  - Floating-point execution pipeline (FPX)
  - Graphics execution pipeline (FGX)
  - Floating-point divide and square root pipeline (FPD)
- Up to one instruction per cycle can be issued to the FGU. Instructions for a given thread are executed in-order. Floating-point register file bypassing is supported for FGU results having a floating-point register file destination (excluding FDIV/FSQRT results).
- A precise exception model is maintained. The FGU uses an exception prediction technique to support full floating-point single thread pipelining, independent of IEEE trap enables. FGU operations are also pipelined across threads. A maximum of two FGU instructions (from different threads) may writeback into the FRF in a given cycle (one FPX/FGX result, and one FPD result).
- 256 entry x 64 bit floating-point register file (FRF). 2R/2W ports. FRF supports eight-way multithreading (eight threads) by dedicating 32 entries for each thread. Each register file entry also includes 14 bits of ECC for a total of 78 bits per entry. Correctable ECC errors (CEs), and uncorrectable ECC errors (UEs) result in a trap if the corresponding enables are set. CEs are never corrected by hardware, but may be corrected by software following a trap.
- One FRF write port (W2) is dedicated to floating-point loads and FPD floating-point results. FPD results always have highest priority for W2 and are not required to arbitrate. The other FRF write port (W1) is dedicated to FPX and FGX results. Arbitration is not necessary for the FPX/FGX write port because of single instruction issue and fixed execution latency constraints. FPX, FGX, and FPD pipelines never stall.
- Independent upper/lower half-word (32 bit) write enables are provided for each FRF write port (W1 and W2).

- The two FRF read ports (R1 and R2) always read from the same thread in a given cycle. The two write ports (W1 and W2) can write to the same thread or different threads in a given cycle.
- An attempt to concurrently read and write the same FRF half-word entry produces a successful write, but the half-word is read as X. In this case the FRF read data is bypassed externally to FRF.
- Floating-point store instructions share an FRF read port with the execution pipelines.
- To avoid stalling FPX or FGX, integer multiply, MULScc, pixel compare and POPC results are guaranteed integer register file (IRF) write port access by the Instruction Fetch Unit (IFU).
- FGU pipelines are focused on area and power reduction:
  - Merged floating-point and VIS datapaths where possible (partitioned add/subtract, partitioned compare, 8x16 multiply)
  - Merged floating-point add, multiply, and divide datapaths where possible (format, exponent)
  - Integer multiply and divide implementations utilize the respective floating-point datapaths
  - POPC implementation leverages the PDIST datapath
  - Simplified floating-point adder pipeline (no independent LED/SED organization, no dedicated i2f pre-normalization)
  - Eliminate OpenSPARC T1 denormalized operand and result handling
  - No floating-point quad precision support
  - Clock gating strategy for dynamic power management
- All FGU executed instructions have the following characteristics
  - Fully pipelined, single-pass
  - Single cycle throughput
  - Fixed six cycle execution latency, independent of operand values

with the exception of

- Floating-point and integer divides and floating-point square root
- Pixel distance (PDIST)

Divide and square root are not pipelined, but execute in a dedicated datapath, and are non-blocking with respect to FPX and FGX. Floating-point divide and square root have a fixed latency. Integer divide has a variable latency, dependent on operand values.

PDIST is a three source instruction, and requires two cycles to read the sources from the FRF which has only two read ports. No FGU executed instruction may be issued the cycle after PDIST is issued. PDIST has a fixed six cycle execution latency, and a throughput of one instruction every two cycles.

- Complex instruction helpers are not used in the OpenSPARC T2 design. Some UltraSPARC implementations use helpers to support instructions such as pixel distance (PDIST) and floating-point block loads and stores.
- FPX uses a parallel normalize/round organization, eliminating the serial delay of a post-normalizer followed by a post-normalization increment by performing the normalization and round function in parallel.
- Execution pipelines are multi-precision in that SP scalar, DP scalar, VIS/integer scalar and VIS/integer SIMD values are stored in the FRF and interpreted by the execution pipeline as unique formats.
- Floating-point State Register (FSR) for IEEE control and status.
- Graphics Status Register (GSR) for VIS control.
- Underflow tininess is detected before rounding. Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded (inexact condition).
- IEEE exception and non-standard mode support (FSR.ns=1) are consistent with UltraSPARC I/II.

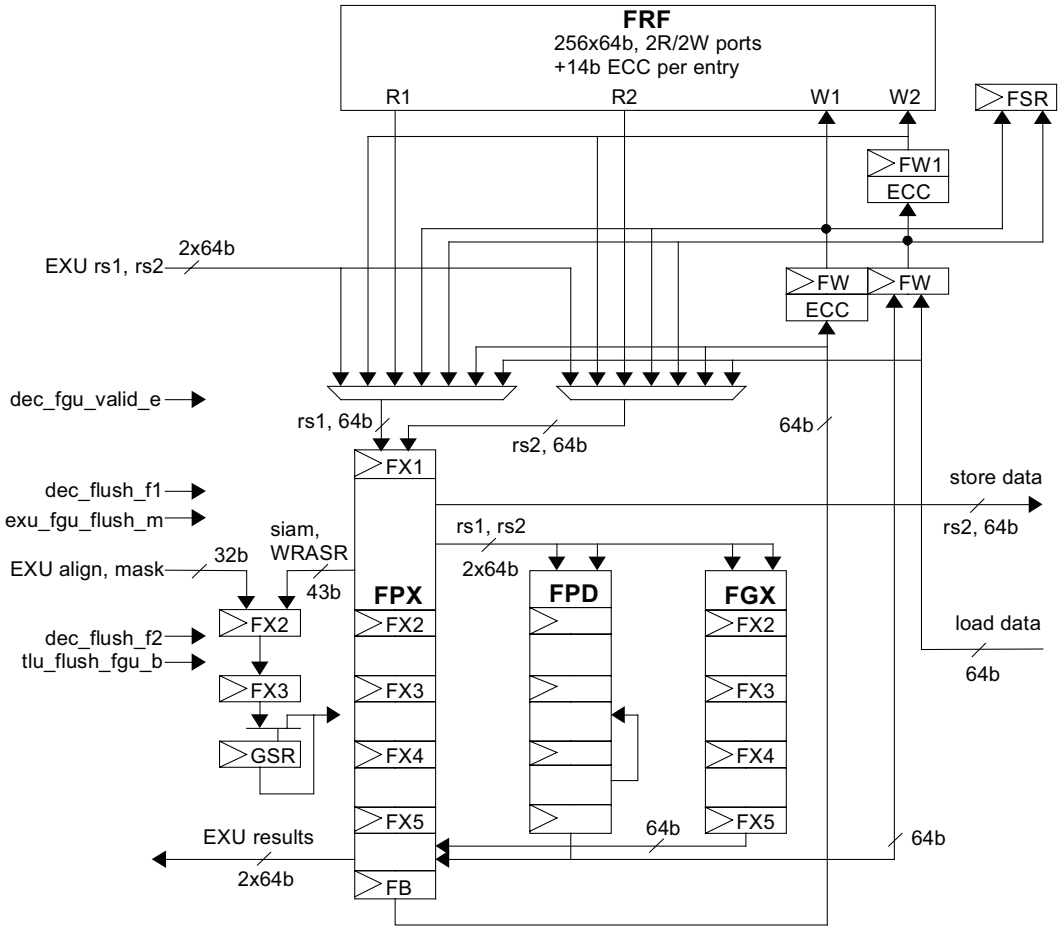
TABLE 7-1 OpenSPARC T2 FGU Feature Summary

	OpenSPARC T2	OpenSPARC T1
ISA, VIS	SPARC V9, VIS 2.0	SPARC V9, subset of VIS 2.0
Core multithreading	8 threads	4 threads
Core issue	1	1
Out-of-order execution (per thread)	No	No
FGU instantiations	1 per core	1 per chip
FGU issue	1	1
FGU architected register file	256 x 64b for 8 threads + 14b ECC per entry 2R/2W ports	128 x 64b for 4 threads + 14b ECC per entry 1 port
FSQRT implemented	Yes	No
IMUL/IDIV execute in FGU	Yes	No
POPC executes in FGU	Yes	No
Number of instructions executed in FGU	129	23

TABLE 7-1 OpenSPARC T2 FGU Feature Summary *(Continued)*

Execution latency:		
FADD, FSUB	6	4
FCMP	6	4
FP/integer convert types	6	4 or 5
FMOV, FABS, FNEG	6	1
FMULs	6 (1/1 throughput)	7 (1/2 throughput)
FGU IMUL, IMULScc	5 (1/1 throughput)	n/a
FDIV	19 SP, 33 DP	32 SP, 61 DP
FGU IDIV	12-41	n/a
FSQRT	19 SP, 33 DP	unimplemented
FMUL 8x16	6 (1/1 throughput)	n/a
FGU FPADD, FPSUB	6	n/a
PDIST	6 (1/2 throughput)	n/a
FGU VIS other	6	n/a
POPC	5	n/a
Single thread throughput:		
FADD, FSUB	1/1	1/27
FMUL	1/1	1/30
FDIV/FSQRT/IDIV blocking	No	No
Hardware quad implemented	No	No
Full hardware denorm implemented	No	Yes

FIGURE 7-2 FGU Pipeline Diagram



7.2 Performance Considerations

While the OpenSPARC T1 to OpenSPARC T2 microarchitecture evolution offers many performance enhancements, in some rare cases performance may decrease.

Certain denormalized operands or expected results may generate an unfinished_FPop trap to software on OpenSPARC T2 (see TABLE 7-16). Unlike other UltraSPARC implementations, a denormalized operand or result never generates an unfinished_FPop trap to software on OpenSPARC T1.

A small set of floating-point and VIS instructions are executed by the OpenSPARC T1 SPARC core FFU (not the off-core FPU). This includes: FMOV, FABS, FNEG, partitioned add/subtract, FALIGNDATA, and logical instructions. The OpenSPARC T2 instruction latency is equivalent to or less than OpenSPARC T1 for these instructions.

7.3 Instruction Set

TABLE 7-2 SPARC V9 Single and Double Precision FPop Instruction Set

Mnemonic	*OpenSPARC T1 Unit*	*OpenSPARC T2 Subunit*	*OpenSPARC T2 Instruction Latency, Single Thread Throughput*		*OpenSPARC T2 Execution Latency, Execution Throughput*	
FABS(s,d)	FFU	FGX	6	1/1	6	1/1
FADD(s,d)	FPU	FPX	6	1/1	6	1/1
FCMP(s,d)	FPU	FPX	6	1/1	6	1/1
FCMPE(s,d)	FPU	FPX	6	1/1	6	1/1
FDIV(s,d)	FPU	FPD	22 SP, 36 DP		19 SP, 33 DP	
FiTO(s,d)	FPU	FPX	6	1/1	6	1/1
FMOV(s,d)	FFU	FGX	6	1/1	6	1/1
FMOV(s,d)cc	FFU	FGX	6	1/1	6	1/1
FMOV(s,d)r	FFU	FGX	6	1/1	6	1/1
FMULs	FPU	FPX	6	1/1	6	1/1
FMULd	FPU	FPX	6	1/1	6	1/1
FNEG(s,d)	FFU	FGX	6	1/1	6	1/1
FsMULd	FPU	FPX	6	1/1	6	1/1
FSQRT(s,d)	n/a	FPD	22 SP, 36 DP		19 SP, 33 DP	
F(s,d)TOi	FPU	FPX	6	1/1	6	1/1
F(s,d)TO(s,d)	FPU	FPX	6	1/1	6	1/1
F(s,d)TOx	FPU	FPX	6	1/1	6	1/1
FSUB(s,d)	FPU	FPX	6	1/1	6	1/1
FxTO(s,d)	FPU	FPX	6	1/1	6	1/1

TABLE 7-3 FGU Integer Multiply, Divide, and Population Count Instructions

Mnemonic	*OpenSPARC T1 Unit*	*OpenSPARC T2 Subunit*	*OpenSPARC T2 Instruction Latency, Single Thread Throughput*		*OpenSPARC T2 Execution Latency, Execution Throughput*	
SMUL{cc}	EXU	FPX	7	1/7	5	1/1
SMUL{cc}i	EXU	FPX	7	1/7	5	1/1
UMUL{cc}	EXU	FPX	7	1/7	5	1/1
UMUL{cc}i	EXU	FPX	7	1/7	5	1/1
MULX	EXU	FPX	7	1/7	5	1/1
MULXi	EXU	FPX	7	1/7	5	1/1
MULScc	EXU	FPX	7	1/7	5	1/1

TABLE 7-3 FGU Integer Multiply, Divide, and Population Count Instructions *(Continued)*

MULScci	EXU	FPX	7	1/7	5	1/1
SDIV{cc}	EXU	FPD	16-44		13-41	
SDIV{cc}i	EXU	FPD	16-44		13-41	
UDIV{cc}	EXU	FPD	16-44		13-41	
UDIV{cc}i	EXU	FPD	16-44		13-41	
SDIVX	EXU	FPD	15-43		12-40	
SDIVXi	EXU	FPD	15-43		12-40	
UDIVX	EXU	FPD	15-43		12-40	
UDIVXi	EXU	FPD	15-43		12-40	
POPC	n/a	FGX	7	1/7	5	1/1
POPCi	n/a	FGX	7	1/7	5	1/1
SAVE (64-bit ADD only)	EXU	FPX			5	1/1
SAVEi (64-bit ADD only)	EXU	FPX			5	1/1
RESTORE (64-bit ADD only)	EXU	FPX			5	1/1
RESTORE (64-bit ADD only)	EXU	FPX			5	1/1

TABLE 7-4 VIS 2.0 FGU Instruction Set

Mnemonic	*OpenSPARC T1 Unit*	*OpenSPARC T2 Subunit*	*OpenSPARC T2 Instruction Latency, Single Thread Throughput*		*OpenSPARC T2 Execution Latency, Execution Throughput*	
BSHUFFLE	n/a	FGX	6	1/1	6	1/1
FALIGNDATA	FFU	FGX	6	1/1	6	1/1
FANDNOT1{s}	FFU	FGX	6	1/1	6	1/1
FANDNOT2{s}	FFU	FGX	6	1/1	6	1/1
FAND{s}	FFU	FGX	6	1/1	6	1/1
FCMPEQ{16,32}	n/a	FPX	6	1/1	6	1/1
FCMPGT{16,32}	n/a	FPX	6	1/1	6	1/1
FCMPLE{16,32}	n/a	FPX	6	1/1	6	1/1
FCMPNE{16,32}	n/a	FPX	6	1/1	6	1/1
FEXPAND	n/a	FGX	6	1/1	6	1/1
FMUL8SUx16	n/a	FPX	6	1/1	6	1/1
FMUL8ULx16	n/a	FPX	6	1/1	6	1/1

TABLE 7-4 VIS 2.0 FGU Instruction Set *(Continued)*

FMUL8x16	n/a	FPX	6	1/1	6	1/1
FMUL8x16AL	n/a	FPX	6	1/1	6	1/1
FMUL8x16AU	n/a	FPX	6	1/1	6	1/1
FMULD8SUx16	n/a	FPX	6	1/1	6	1/1
FMULD8ULx16	n/a	FPX	6	1/1	6	1/1
FNAND{s}	FFU	FGX	6	1/1	6	1/1
FNOR{s}	FFU	FGX	6	1/1	6	1/1
FNOT1{s}	FFU	FGX	6	1/1	6	1/1
FNOT2{s}	FFU	FGX	6	1/1	6	1/1
FONE{s}	FFU	FGX	6	1/1	6	1/1
FORNOT1{s}	FFU	FGX	6	1/1	6	1/1
FORNOT2{s}	FFU	FGX	6	1/1	6	1/1
FOR{s}	FFU	FGX	6	1/1	6	1/1
FPACKFIX	n/a	FGX	6	1/1	6	1/1
FPACK{16,32}	n/a	FGX	6	1/1	6	1/1
FPADD{16,32}{s}	FFU	FPX	6	1/1	6	1/1
FPMERGE	n/a	FGX	6	1/1	6	1/1
FPSUB{16,32}{s}	FFU	FPX	6	1/1	6	1/1
FSRC1{s}	FFU	FGX	6	1/1	6	1/1
FSRC2{s}	FFU	FGX	6	1/1	6	1/1
FXNOR{s}	FFU	FGX	6	1/1	6	1/1
FXOR{s}	FFU	FGX	6	1/1	6	1/1
FZERO{s}	FFU	FGX	6	1/1	6	1/1
PDIST	n/a	FGX	6	1/2	6	1/2
SIAM	FFU	FGX	6	1/1	6	1/1

7.4 Interface with Other Blocks

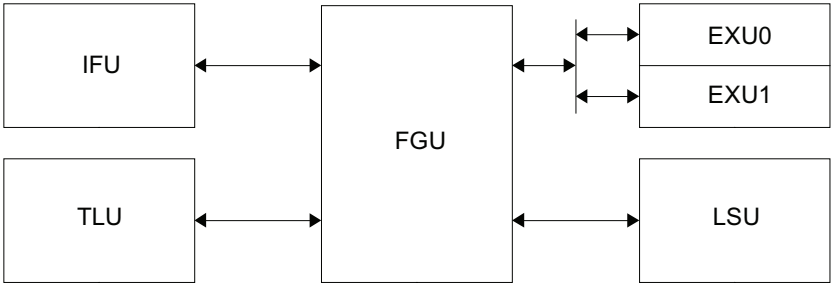
7.4.1 Interface with IFU

IFU (Instruction Fetch Unit) provides instruction control information as well as rs1, rs2 and rd register address information.

Up to one instruction per cycle can be issued to the FGU.

Divide and square root are long latency instructions executed in the FPD pipeline. Up to two FPD instructions from different threads can be outstanding; FPD Up to two FPD instructions from different threads can be outstanding; FPD generally executes the two instructions serially (see 6.10.1.1 for exceptions). FPD provides early completion signals to the IFU and LSU a short number of fixed cycles prior to completion. FPD never stalls due to write port arbitration. The IFU and LSU guarantee that FPD has access to the appropriate FRF/IRF write port by ensuring that a load miss/hit is not utilizing the shared write port.

FIGURE 7-3 FGU Top-Level Interface Block Diagram



FGU provides appropriate FSR.fcc information to the IFU during FX2 and FX3 (including load FSR). The information includes a valid bit, the fcc data and thread ID (TID) and is non-speculative.

IFU maintains copies of fcc for each thread.

FGU provides the FPRS.fef bit to the IFU for each TID (used by IFU to determine fp_disabled).

IFU provides a single FMOV valid bit to FGU indicating whether the appropriate icc, xcc, fcc or ireg condition is true or false.

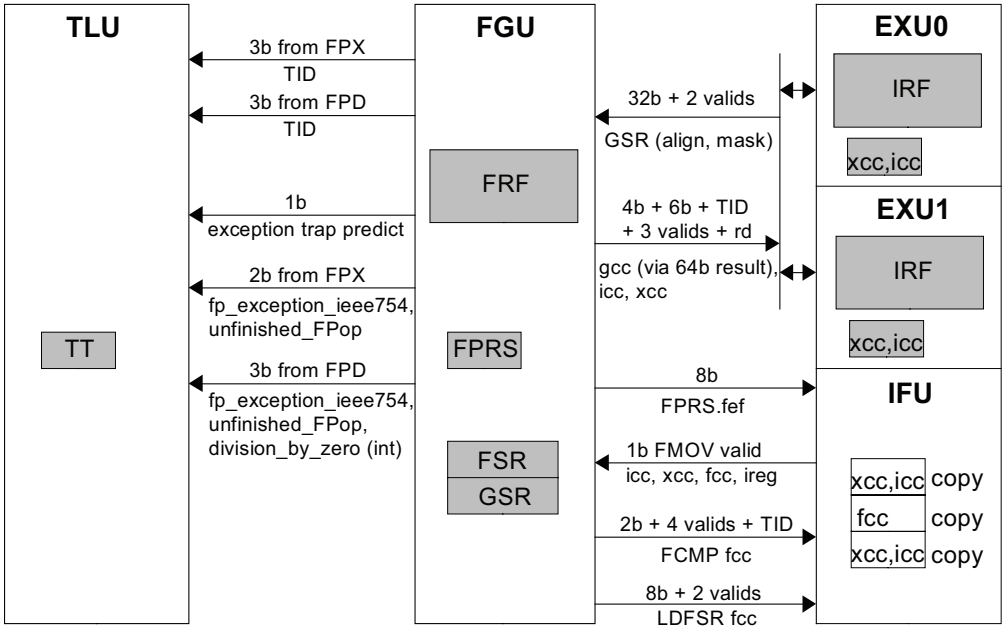
FGU receives the following flush signals from IFU:

flush execution pipeline stage FX2 (transmitted during FX1/M stage)

flush execution pipeline stage FX3 (transmitted during FX2/B stage)

The FGU flushes FPD based on the IFU and TLU initiated flush signals. Once an FPD instruction has executed beyond FX3 it cannot be flushed by an IFU or TLU initiated flush.

FIGURE 7-4 Trap, Condition Code, and GSR Related Interfaces



7.4.2 Interface with TLU

FGU provides the following trap information to the TLU (Trap Logic Unit):

unfinished_FPop

fp_exception_ieee_754

fp_cecc (FRF correctable ECC error)

fp_uecc (FRF uncorrectable ECC error).

division_by_zero (integer)

exception trap prediction

FGU receives the following flush signal from TLU:

flush execution pipeline stage FX3 (transmitted during FX2/B stage)

7.4.3 Interface with LSU

Floating-point load instructions share an FRF write port with FPD floating-point results. FPD results always have priority for the shared write port. FPD notifies the IFU and LSU when a divide or square root is near completion to guarantee that load data does not collide with the FPD result. Loads update the FRF or FSR directly, without proceeding down the execution pipeline. Load FSR is a serializing operation for a given thread (all previous FPops have completed, then load FSR completes prior to issuing subsequent FPops).

LSU (Load Store Unit) always delivers 32 bit load data replicated on both the upper (even) and lower (odd) 32 bit halves of the 64 bit load data bus. ECC information is generated by the FGU prior to updating the FRF.

Floating-point store instructions share an FRF read port with the execution pipelines. Store FSR is a serializing operation for a given thread (all previous FPops have completed, then store FSR completes prior to issuing subsequent FPops).

FGU always delivers 32 bit store data on the upper (even) 32 bits of the 64 bit store data bus. The lower (odd) 32 bits are undefined. FGU delivers FRF ECC UE/CE information to the LSU one cycle after the data.

FGU does not perform any byte swapping based on endianness (handled by LSU), or load data alignment for 32, 16, and 8 bit loads (also handled by LSU).

7.4.4 Interface with EXUs

Each EXU can generate the two 64 bit source operands needed by the integer multiply, divide, POPC, SAVE, and RESTORE instructions. The EXUs provide the appropriate sign extended immediate data for rs2, and provide rs1 and rs2 sign extension and zero fill formatting as required. The IFU provides a destination address (rd) which the FGU provides to the EXUs upon instruction completion.

The architected Y register for each thread is maintained within the EXUs. MULScc and 32 bit IMUL instructions write the Y register. MULScc and 32 bit IDIV instructions read the Y register

FGU provides a single 64 bit result bus to the EXUs, along with appropriate icc and xcc information. The same result bus provides appropriate 64 bit formatted "gcc" information to the EXUs upon completion of the VIS FCMP (pixel compare) instructions. The result information includes a valid bit, TID, and destination address (rd). FGU clears the valid bit under the following conditions: division_by_zero trap (IDIV only), enabled FRF ECC UE/CE (VIS FCMP only), EXU, IFU or TLU initiated flush.

Each EXU provides GSR.mask and GSR.align fields to the FGU. The EXUs also provide individual valid bits for GSR.mask and GSR.align, and the TID.

7.5 Power Management

FGU power management is accomplished via two methods: (1) block controllable clock gating, and (2) reduced switching activity on major interface busses when clocks are enabled. Power management is provided without affecting functionality or performance (software and performance transparent).

FGU clocks are dynamically disabled or enabled as needed, thus reducing clock power and signal activity when possible. The FGU has independent clock control for four clock domains:

1. Main. Any instruction requiring an FGU action will enable this domain. This includes, but is not limited to: FGU related load/store instructions, BMASK or ALIGNADDRESS instructions, ASI instructions, any SPARC V9 or VIS 2.0 instruction executed in the FGU.
2. Multiply. Any SPARC V9 or VIS 2.0 floating point or integer multiply type instruction.
3. Divide. Any SPARC V9 floating point or integer divide or square root type instruction.
4. VIS. Any VIS 2.0 instruction executed in the FGX subunit.

FGU register file (FRF) power management is accomplished indirectly. Instructions which do not access the FRF automatically disable all FRF read/write enables. Register file clocks are not disabled.

Clocks are gated for a given domain when it is not in use. A domain will have its clocks enabled only under one of the following conditions:

- The domain is executing a valid instruction
- An instruction is issuing to the domain
- Test mode is active (MBIST, Macro Test)
- FGU power management disable is active

The FGU clock gating feature automatically powers up and powers down each of the four clock domains, based on the contents of the instruction stream. Given domains are clocked only when required. For example, if no divide or square root instructions are executing, the divide clock domain is automatically powered down.

Switching activity on major FGU input and output busses is reduced by holding the bus to a constant value when the bus is not in use, even while the FGU is clocking. For example, the 64-bit FGU store data bus to the LSU is held constant while the FGU is processing non-store instructions.

7.6 FRF ECC Error Handling

Floating-point register file (FRF) correctable ECC errors (CEs), and uncorrectable ECC errors (UEs) result in a trap if the corresponding enables are set. CEs are never corrected by hardware, but may be corrected by software following a trap.

- If CETER.PSCCE=0 (thread specific), or if CERER.FRF=0 (non thread specific), then FGU continues operation. FGU does not report the error to TLU (error is not logged). This applies to both CE and UE.
- The 14-bit ECC generated value (includes 7-bits for odd, 7-bits for even) from the FRF entry causing the ECC error is logged. This applies to DP and SP sources.
- A 7-bit ECC mask input and two enables are used to inject ECC errors. If enabled, the 7-bit mask is XORed into the normally generated ECC check bits for both the odd and even 32-bit words.
- FRF ECC priority: (1) rs1_ue (2) rs1_ce (3) rs2_ue (4) rs2_ce (5) rs3_ue (6) rs3_ce
- FRF will never log multiple errors. Only the highest priority error gets logged.
- PDIST reports UE/CE on the 2nd beat if rs1 and rs2 do not have an error and rs3 does.
- FRF ECC error logs the 6-bit register number (see V9 page 40). Information indicating whether the source was SP/DP isn't logged.
- FGU exception trap prediction is never asserted for CE or UE.

7.6.1 ASI Read Access for FRF ECC Check Bits

- ASI read access is provided for all FRF ECC check bits stored in the FRF.
- Write access via ASI is not provided for the FRF ECC check bits stored in the FRF. Write access is provided only via the 7-bit ECC mask used for error injection.
- The 5 MSBs of the 6-bit register number (see V9 page 40) of the FRF address to be read is provided in bits [7:3] of the 65-bit ASI ring control/data bus. The ECC check bits for both the odd and even 32-bit words are always provided. The even ECC check bits are placed in bits [13:7] of the 65-bit ASI ring control/data bus, while the odd ECC check bits are placed in bits [6:0].
- ECC check bits are always read using the FRF rs1 port.
- ECC check bit ASI reads will never produce a CE or UE, and will never report an exception via bits [55:48] of the 65-bit ASI ring control/data bus.

7.7 Floating-Point Execution Pipeline (FPX)

7.7.1 Functionality

FPX executes the following instruction types:

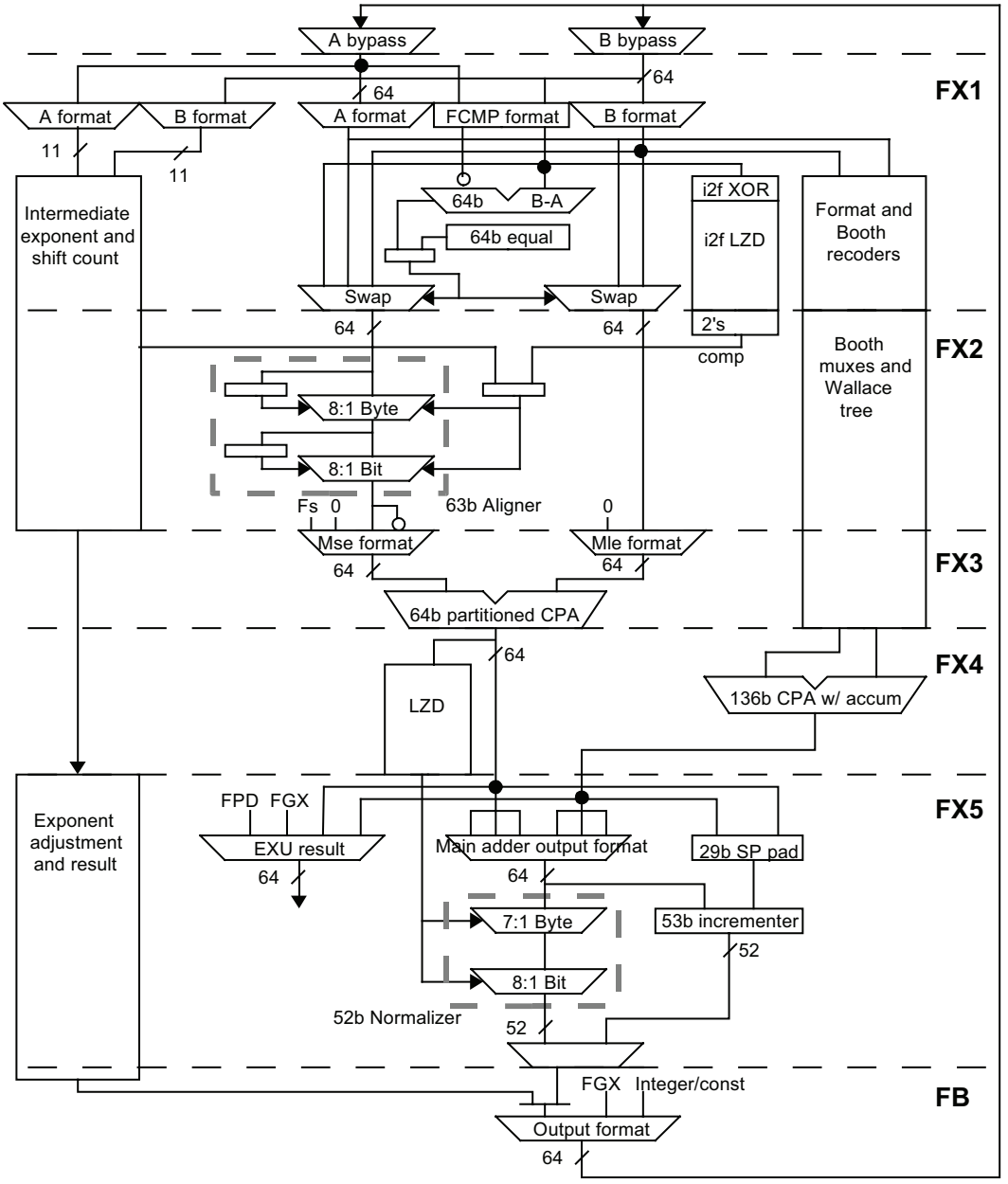
- SPARC V9 single and double precision FPops (with the exception of FDIV and FSQRT)
- SPARC V9 integer multiply instructions, including MULScc
- SPARC V9 SAVE and RESTORE instructions (64-bit add portion only) instructions
- VIS 2.0 partitioned compare, 8x16 multiply, and partitioned add/subtract instructions
- FPX supports FPD executed instructions (IDIV, FDIV, FSQRT):
- exponent result generation
- mantissa input formatting
- floating-point default response result detection
- mantissa output formatting including default response result generation
- QNaN
- max floating-point
- infinity
- zero floating-point
- All FPX instructions are fixed latency, independent of operand values.

- Simplified mantissa datapath organization (no independent LED/SED datapaths, no dedicated i2f pre-normalization).
- FPX uses a parallel normalize/round organization, eliminating the serial delay of a post-normalizer followed by a post-normalization increment by performing the normalization and round function in parallel
- A single-pass implementation is used for all multiply instructions, producing a throughput of one instruction every cycle.
- NaN source propagation is supported by steering the appropriate NaN source (see SPARC V9 manual section B.2) through the datapath to the result.
- Certain denormalized operands or expected results may generate an unfinished_FPop trap to software, indicating that the FGU was unable to generate the correct results. The conditions which generate an unfinished_FPop trap are consistent with UltraSPARC I/II.

7.7.2 Mantissa Datapath

The floating-point execution pipeline (FPX) is implemented in six pipeline stages (FX1-FX5 and FB). The FB stage includes a final result format mux and the timing delay associated with the FRF bypass muxes.

FIGURE 7-5 FPX Execution Datapath Block Diagram



7.7.2.1 FPX Unified Datapath

Other UltraSPARC implementations use a LED/SED mantissa datapath organization to decrease latency. The LED/SED organization includes two independent datapaths. The LED (large exponent difference) datapath requires a large alignment shifter but minimal normalization shifter, while the SED (small exponent difference) datapath requires a large normalization shifter but minimal alignment shifter. Latency is decreased because every instruction executes in LED or SED and never requires both a large alignment shift and a large normalization shift.

For area efficiency, FPX organizes the datapath in a conventional unified manner (no independent LED/SED datapaths). In addition, because only one instruction per cycle may be issued to the FGU, independent add and multiply pipelines are not required. The OpenSPARC T2 FGU realizes additional efficiency by merging the add and multiply pipelines into a single execution pipeline (FPX).

TABLE 7-5 FPX Mantissa Datapath Stages

Stage	Add Action	Multiply Action
FX1	Format input operands	
	Compare fractions	Booth encoder
FX2	Align smaller operand to larger operand	Generate partial products using a radix-4 Booth algorithm
	Invert smaller operand if a logical (effective) subtraction is to be performed	Accumulate partial products using a Wallace tree configuration
FX3	Compute intermediate result (A+B)	
FX4	Determine normalization shift amount	Add the two Wallace tree outputs using a carry-propogate adder
FX5	Normalize and round	
FB	Bypass	

7.7.2.2 Aligner

In floating-point arithmetic, when two numbers are added the exponents must first be equal. To prepare for the addition in the main adder, the operand mantissa with the smaller exponent (MSE) is shifted in the alignment shifter to produce equal A and B exponents. The exponent logic computes the MSE alignment shift count (SC). SC is used to produce the selects for the alignment shifter muxes. The alignment shifter is required to perform a right shift ranging from 0 to 53 bits (63 bits for convert to 64 bit integer instructions). A data forward path is used to forward the operand mantissa with the larger exponent (MLE) directly to the main adder,

bypassing the alignment shifter altogether. Sticky bit gathering is performed for bits that are shifted past the operand mantissa width (the integer data width for convert to integer instructions).

The output of the alignment shifter is inverted when a logical subtract (effective subtract) is to be performed. Note that if a logical subtract is to take place, M_{SE} is inverted, never M_{LE}.

The maximum SC possible if a non-zero M_{SE} is to participate in the main add is 63 bits. If SC is greater than 63 then M_{SE} shifts beyond M_{LE}, in which case M_{SE} only affects the sticky bit calculation. If the intermediate SC is greater than 63 the output of the alignment shifter is ignored, and M_{SE}=0 is selected prior to the main adder. A negative SC is not possible.

A logical subtract operation is defined by the equation:

logical_subtract = (Sa XOR Sb) XOR Si

Where:

Sa=1 if operand A is negative, else Sa=0

Sb=1 if operand B is negative, else Sb=0

Si=1 if the instruction executing is FSUB(s,d) or FPSUB{16,32}{s}, else Si=0

A conventional swap mantissa alignment method is used. M_{SE} is always aligned to M_{LE}. The M_{SE} alignment shift count can be determined from the equation:

IF (Eb Ea) THEN (SC= (Eb-Ea)); ELSE (SC= (Ea-Eb))

Where:

Ea represents the biased exponent of operand A

Eb represents the biased exponent of operand B

SC represents the alignment shift count

FIGURE 7-6 Mantissa Input Format Muxes

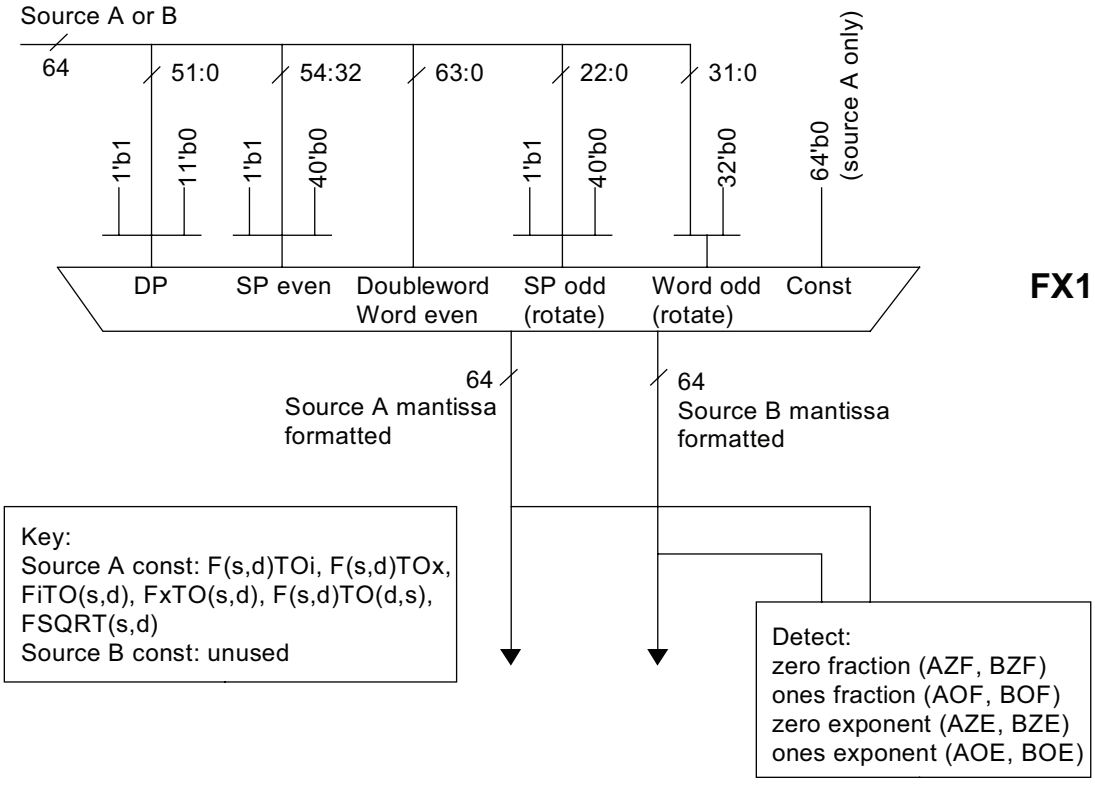


FIGURE 7-7 Swap Determination and Partitioned Compare

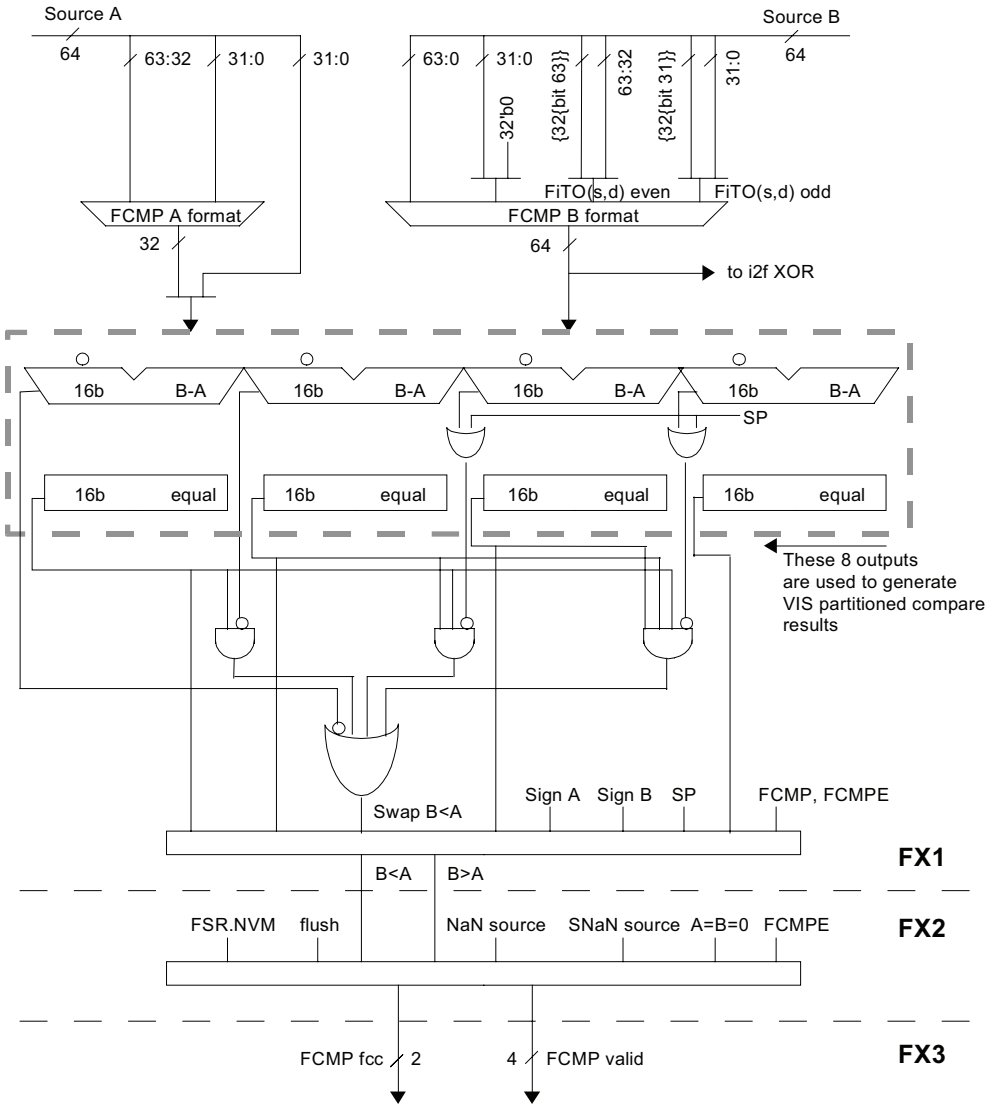


FIGURE 7-8 Aligner

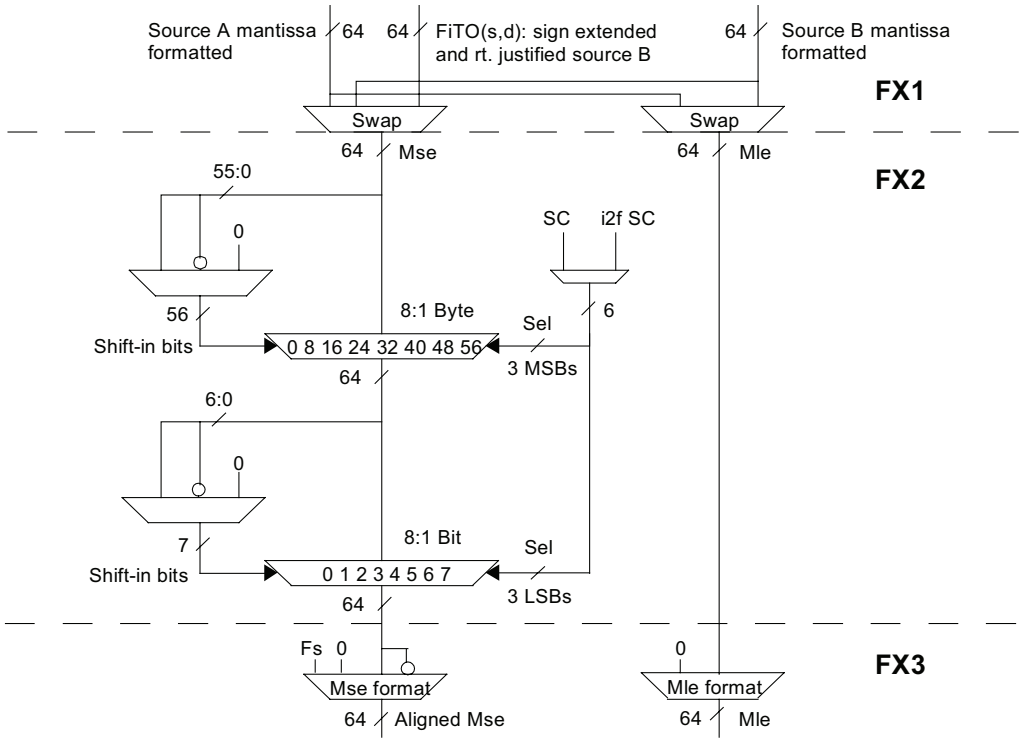
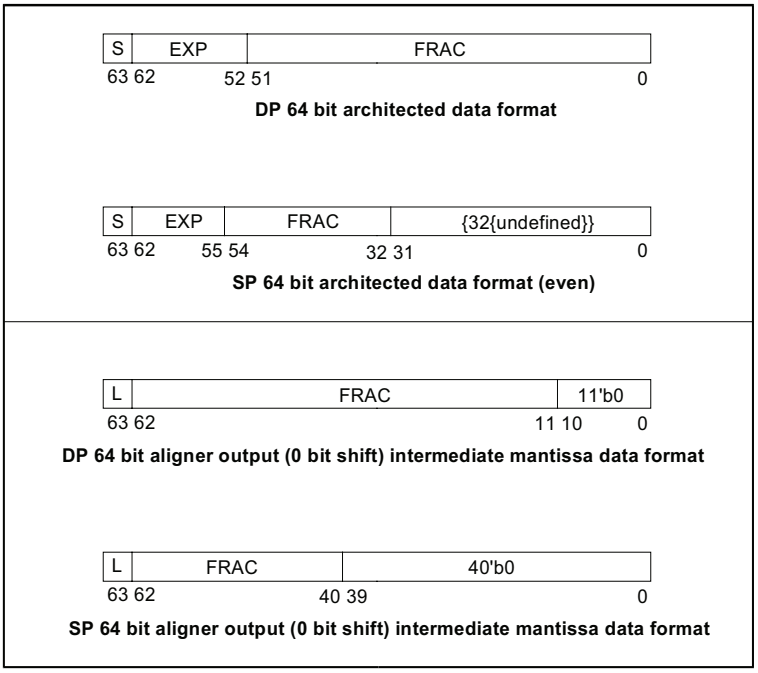


FIGURE 7-9 Architected and Internal Data Formats



7.7.2.3 Main Adder

The main adder uses a partitioned 64 bit to compute the sum of the two inputs:

- the unmodified operand mantissa with the larger exponent (M_{LE})
- the aligned operand mantissa with the smaller exponent (M_{SE})

The 64 bit width is required for convert to 64 bit integer instructions as well as VIS instructions.

For logical subtract operations, the aligned M_{SE} input is inverted prior to the main adder, and the main adder carry in is asserted. The carry in may or may not propagate to the LSB position depending on the guard, round, and sticky bits. Note that M_{LE} is always treated as positive relative to the aligned M_{SE}, and thus is never inverted.

The result produced by the main adder is always positive. A full comparison (including exponents and mantissas) of the two input operands ensures that M_{LE} has a mantissa which is greater than or equal to M_{SE}. A logical subtract operation always produces a main adder carry out.

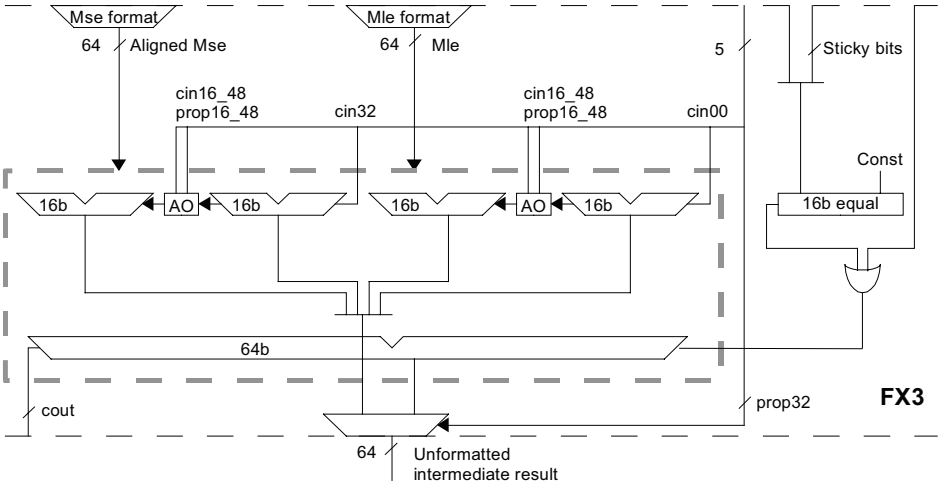
The main adder produces the output:

M_{LE} + (M_{SE} XOR logical_subtract) + SUBinc

Where:

SUBinc = (logical_subtract & G & R & ~align_sticky)

FIGURE 7-10 Main Adder



7.7.2.4 Normalizer and Rounder

A parallel normalize/round organization is used, eliminating the serial delay of a post-normalizer followed by a post-normalization increment by performing the normalization and round function in parallel.

Post-normalization is accomplished by shifting the mantissa left while decrementing the exponent for each bit shifted until the leading mantissa bit (the first bit to the left of the binary point) becomes a 1'b1. The normalizer is required to perform a left shift of 0 to 52 bits (63 bits for integer to floating-point conversions) on the 64 bit unnormalized intermediate result to obtain a normalized intermediate result.

FPX requires normalization to be performed only for logical subtract operations. Logical add operations do not require normalization since the unnormalized intermediate mantissa result is in the format 01.XX or 1X.XX, assuming pre-normalized operands. For a given operation, the leading zero detector (LZD) circuit must find the leading one (find one) to properly produce shift control signals for the normalizer. Finding the leading zero (find zero) is not required given that the result produced by the main adder is always positive.

The outputs of the LZD must be encoded into an 11 bit value (the width of the exponent datapath) and two's complemented since the number of leading zeros in the intermediate mantissa result must be subtracted from the intermediate exponent (Eint) to calculate the exponent result.

An example of a logical subtract operation which requires normalization is given below. Assume that Ea>Eb and that the B operand has been aligned accordingly.

```
 10000000   A
- 01111111   B aligned
----------
   00000001   (A-B) unnormalized
   10000000   (A-B) normalized, 7 bit left shift
```

The function of the rounder is to increment the normalized intermediate mantissa result by adding 1'b1 to the least significant mantissa bit if it is determined that the normalized intermediate result should be incremented due to rounding. After normalization, if the infinitely precise intermediate result cannot be represented in the precision required by the instruction, then the intermediate result is inexact. Rounding may or may not cause the intermediate result to be incremented. The incremented or non-incremented intermediate result is chosen based on the least significant fraction bit, guard bit, round bit, sticky bit, sign of the result and round mode.

The parallel normalize/round technique requires that the approximate location (one of three positions) of the round increment position be known prior to the normalization/round step. Because FPX uses a swap mantissa alignment method (not an offset mantissa alignment method) and supports only normalized operands, it can be shown that if the intermediate result should be incremented due to rounding, then the unnormalized intermediate mantissa result must be in one of three possible formats:

1.1X.XX

2.01.XX

3.00.1X

A mux located above the normalizer and rounder changes formats 1X.XX and 00.1X into format 01.XX by performing a 1 bit shift (this mux also selects between intermediate results from the main adder and multiplier). Thus, if the intermediate result is to be incremented due to rounding, then no additional normalization shifting is required.

If the round increment produces a carry out (Rcout) and the incremented intermediate result is selected, then the exponent must be incremented, and {Rcout, mantissa} must be logically shifted right by one bit position. However, a right shift is not physically required. If Rcout=1 then each mantissa bit must have been 1'b1 prior to the round increment, and must be 1'b0 after the round increment. Thus, a right shift is not required as long as the leading mantissa bit is set to 1'b1.

Following normalization and rounding, FPX formats the result as required, including default response results as shown in TABLE 7-6.

TABLE 7-6 Default Response Results

Output Constant Type	*64 bit Default Response Result*		
	DP	***SP***	***Integer***
QNaN	64'h7fffffffffffffff	{32'h7fffffff,{32{1'bx}}}	
Max floating-point	{Sign,11'b11111111110, 52'hfffffffffffff}	{Sign,8'hfe,{23{1'b1}},{32{1'bx }}}	
Infinity	{Sign,{11{1'b1}},52'b0}	{Sign,8'hff,23'b0,{32{1'bx}}}	
Zero	{Sign,63'b0}	{Sign,31'b0,{32{1'bx}}}	
32 bit max +integer			{32'h7fffffff,{32{1'bx}}}
32 bit max - integer			{32'h80000000,{32{1'bx}}}
64 bit max +integer			64'h7fffffffffffffff
64 bit max - integer			64'h8000000000000000
64 bit 2^{32}-1			64'h00000000ffffffff
64 bit 2^{31}-1			64'h000000007fffffff
64 bit -2^{31}			64'hffffffff80000000

FIGURE 7-11 Normalizer and Rounder

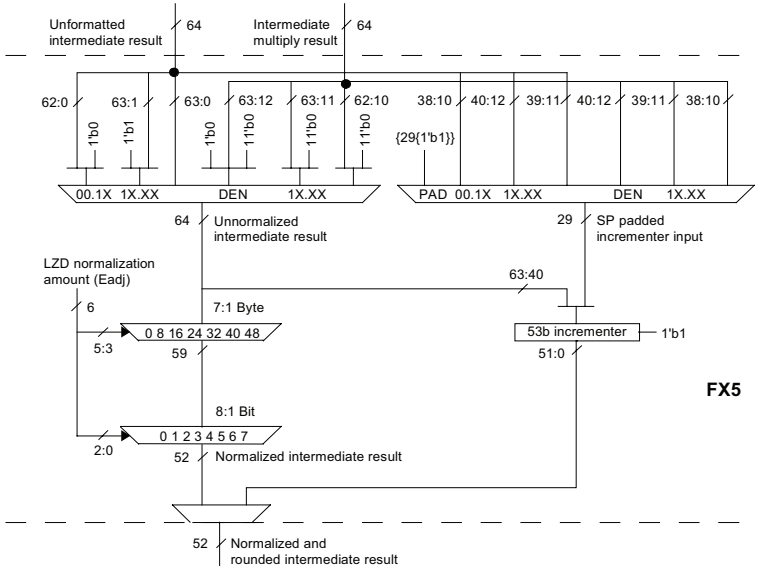
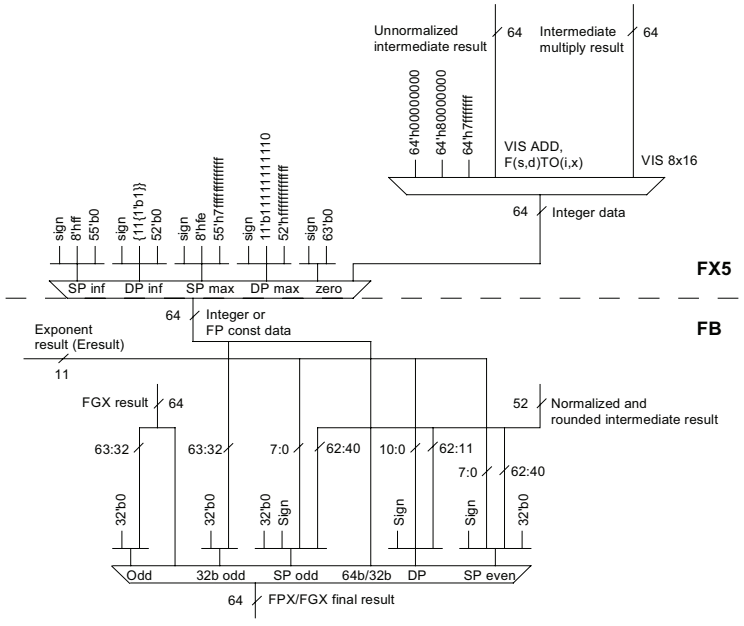


FIGURE 7-12 Output Format Muxes



7.7.2.5 Non-Arithmetic Instruction Implementation

Compare

Compare instructions include FCMP(s,d) and FCMPE(s,d).

FPX datapath behavior and considerations:

- Compare mantissas and exponents.
- Generate condition code result.

Convert to Integer

Convert to integer instructions include F(s,d)TOi and F(s,d)TOx.

FPX datapath behavior and considerations:

- Source proceeds down the M_{SE} path (unless invalid integer convert is detected).
- Based on the source exponent, the aligner appropriately positions the source mantissa to form a 32 or 64 bit integer. Because inexact convert to integer results are always truncated, preserving a guard bit position for rounding is not required.
- Convert negative sign-magnitude source to negative signed integer via two's complement. Use logical subtract signal to perform inversion and the main adder to perform +1.
- Main adder performs (aligned M_{SE}) + 0 if positive sign-magnitude source, or (aligned M_{SE}) + 1 if negative sign-magnitude source. F(s,d)TOi and F(s,d)TOx support requires a +1 at bit positions 32 and 0 of the 64 bit intermediate result respectively, if the source is a negative sign-magnitude value.
- All of the convert to integer instructions may round (produce an inexact result). Always round toward zero (truncate), ignoring FSR.rd and GSR.irnd. Because inexact convert to integer results are always truncated, the intermediate result is never incremented due to rounding.
- Because inexact convert to integer results are always truncated, support for suppression of round increment for intermediate results $7ff...fff_{16}$ and $800...000_{16}$ is not required.
- Because inexact convert to integer results are always truncated, reverse round support (rounding a negative signed integer requires a decrement) is not required.
- No normalization is required. Normalizer is bypassed (override of normalizer shift control signals is not required).

Convert to integer instructions are executed as an add operation. For example, a double precision floating-point operand to be converted to a 32 bit signed integer is added to a constant $0.0x2^{31}$. This shifts the integer portion of the floating-point operand to the upper 32 bit portion of the 64 bit main adder output.

A negative floating-point value when converted to signed integer must be represented in two's complement form (invert and add one). The invert is accomplished by the alignment shifter. The add one is performed by the main adder.

Invalid integer convert detection must also be performed. When a NaN, infinity, large positive operand 2^N, or large negative operand $-(2^N+1)$ is converted to an integer, the datapath is signaled to deliver the appropriate default response result ($7ff...fff_{16}$ for positive operand, $800...000_{16}$ for negative operand). Where N=31 for F(s,d)TOi, N=63 for F(s,d)TOx.

Convert From Integer

Convert from integer instructions include FiTO(s,d) and FxTO(s,d).

FPX datapath behavior and considerations:

Source proceeds down the M_{SE} path (by ensuring that Eb<Ea).

Based on the signed integer source, the aligner, organized as a rotator to support a left shift function, appropriately positions the source to form a normalized floating-point value. Adjust exponent accordingly.

Convert negative signed integer source to negative sign-magnitude value via two's complement. Use logical subtract signal to perform inversion and the main adder to perform +1.

Main adder performs (normalized M_{SE}) + 0 if positive signed integer source, or (normalized M_{SE}) + 1 if negative signed integer source. FiTO(s,d) and FxTO(s,d) support requires a +1 at bit positions 32 and 0 of the 64 bit intermediate result respectively, if the source is a negative signed integer.

FiTOs and FxTO(s,d) may round (produce an inexact result), and the intermediate result may be incremented due to rounding. FiTOd never rounds (always produces an exact result). The round function is performed by the rounder. The round increment occurs at bit position 40 of the 64 bit intermediate result for F(i,x)TOs, and bit position 11 for FxTOd.

Normalize as usual, adjust exponent accordingly (input to normalizer is already normalized via the aligner, thus, the normalizer always shifts 0 bits).

Convert from integer instructions are executed as an add operation where A=0. The B operand is a 32 or 64 bit signed integer value which is to be converted to a floating-point value in the destination format. The intermediate exponent is forced to a constant

Negative integers must first be two's complemented (invert and add one) to produce a sign-magnitude value. The invert is accomplished by the alignment shifter. The add one is performed by the main adder

FPX uses a parallel normalize/round technique which requires that the approximate location (one of three positions) of the round increment position be known prior to the normalize/round step. Convert from integer instructions do not meet this requirement because the round increment position is not known to be one of three positions. To ensure that the convert from integer instructions maintain the same fixed latency as other FPX executed instructions, the aligner is organized as a rotator to support a left shift function capable of normalizing the integer source prior to the main adder. The normalized intermediate result is then rounded by the rounder as specified by FSR.rd or GSR.irnd.

Some UltraSPARC implementations utilize a dedicated i2f (integer to floating-point) pre-normalization shifter to perform normalization of the integer source prior to the main adder. N1 allows the instruction to execute one additional cycle to allow normalization and rounding to be performed serially with existing hardware. For the N2 implementation, the aligner/rotator solution is more area efficient and does not impact instruction latency or cycle timing.

Convert Double to Single

Convert double to single instructions include FdTOs.

FPX datapath behavior and considerations:

Convert source exponent from DP to SP (-896).

Source proceeds down the M_{LE} path, bypassing the aligner. Alignment SC is a don't care (aligner input is zero).

Main adder performs ((aligned M_{SE}) + M_{LE}) = (0 + M_{LE}).

FdTOs may round (produce an inexact result), and the intermediate result may be incremented due to rounding. The round function is performed by the rounder. The round increment occurs at bit position 40 of the 64 bit intermediate result.

Normalize as usual, adjust exponent accordingly (input to normalizer is already normalized, thus, always shifts 0 bits).

Convert Single to Double

Convert single to double instructions include FsTOd.

FPX datapath behavior and considerations:

- Convert source exponent from SP to DP (+896).

- Source proceeds down the M_{LE} path, bypassing the aligner. Alignment SC is a don't care (aligner input is zero).
- Main adder performs ((aligned M_{SE}) + M_{LE}) = (0 + M_{LE}).
- FsTOd never rounds (always produces an exact result).
- Normalize as usual, adjust exponent accordingly (input to normalizer is already normalized, thus, always shifts 0 bits).

7.7.2.6 Multiply Step Instruction Implementation

Multiply step instructions include MULScc.{i}

FPX datapath behavior and considerations:

- This instruction has no significant impact on FPX datapath behavior.
- An EXU/FGU interface already exists to support instructions such as IMUL and IDIV. The interface includes source operand buses, result bus, and icc/xcc fields.
- EXU0 or EXU1 provides two source operands. The EXUs provide the appropriate sign extended immediate data for rs2, and provide rs1 and rs2 zero fill formatting to produce 64 bit sources. The EXUs format operand A with a pre-shifted (1 bit) rs1 and include Y[0] and (icc.N XOR icc.V) within the operand supplied to the FGU. FGU selects operand B, choosing rs2 or zero based on Y[0]. If Y[0]=0 then zero is selected by the existing MSE format mux located above the main adder. If Y[0]=1 then rs2 is selected.
- Ea and Eb are forced to the same constant. Alignment SC=0 because Ea=Eb.
- Main adder performs a 64-bit add ((aligned MSE) + MLE).
- Provide 64 bit result to the EXUs, along with appropriate icc/xcc information.

7.7.2.7 Save and Restore Instruction Implementation

Save and Restore instructions include SAVE{i} and RESTORE{i}.

FPX datapath behavior and considerations:

- These instructions have no significant impact on FPX datapath behavior.
- The FGU only performs the 64-bit add portion of these instructions.
- Ea and Eb are forced to the same constant. Alignment SC=0 because Ea=Eb.
- Main adder performs a 64-bit add ((aligned M_{SE}) + M_{LE}).
- FGU provides 64 bit result to the EXUs.

7.7.2.8 FPX VIS Instruction Implementation

Partitioned Add/Subtract Instructions

Partitioned add/subtract instructions include FPADD{16,32}{s} and FPSUB{16,32}{s}.

FPX datapath behavior and considerations:

- These instructions have no significant impact on FPX datapath behavior.
- Ea and Eb are forced to the same constant. Alignment SC=0 because Ea=Eb.
- Main adder performs partitioned add/subtract ((aligned M_{SE}) M_{LE}).
- Subtraction is accomplished via two's complement (invert and add one). The invert is performed by the existing M_{SE} format mux located above the main adder. The main adder performs the +1.
- No normalization is required. Normalizer is bypassed (override of normalizer shift control signals is not required).

Partitioned Compare Instructions

Partitioned compare instructions include FCMPEQ{16,32}, FCMPGT{16,32}, FCMPLE{16,32}, and FCMPNE{16,32}.

FPX datapath behavior and considerations:

- Compare partitioned signed integer sources using partitioned comparators. Comparators are common with the floating-point compare instructions (FCMP and FCMPE).
- Provide 64 bit formatted condition code (gcc) result to the EXUs.

8x16 Multiply Instructions

8x16 multiply instructions include FMUL8SUx16, FMUL8ULx16, FMUL8x16, FMUL8x16AL, FMUL8x16AU, FMULD8SUx16, and FMULD8ULx16.

FPX datapath behavior and considerations:

- Multiplier Booth encoding algorithm and Wallace tree are partitioned accordingly.

7.7.2.9 NaN Source Propagation

FPX supports NaN source propagation by steering the appropriate NaN source (see SPARC V9 manual section B.2) through the datapath to the result.

FPX datapath behavior and considerations:

- No special action is required by the aligner. The propagating NaN may enter the aligner (as M_{SE}) only if Ea=Eb=111...111_2. If Ea=Eb then SC=0.
- Main adder performs propagating NaN + 0. M_{LE} or M_{SE} is forced to zero prior to the main adder.
- No special action is required by the normalizer. The input to the normalizer is already normalized (L bit is always set to 1'b by the input format muxes), thus, the normalizer always shifts 0 bits.
- For multiplication, divide, and square root, if Ea=111...1112 or Eb=111...1112 then the appropriate NaN propagates through the main adder and normalizer (FPX pipeline is always used, never FPD.
- The output format logic always sets the NaN quiet bit (the most significant bit of the result fraction) to 1'b1.

7.7.2.10 Multiplier

The multiplier performs a 64 bit x 64 bit multiplication. Multiplication involves two basic operations (1) the generation of partial products and (2) their accumulation. Radix-4 Booth encoding is used to reduce the number of partial products. The radix-4 Booth encoding algorithm reduces the number of additions required to multiply two N bit two's complement numbers from N to N/2. A Wallace tree configuration of carry-save adders (CSAs) accumulates the N/2 partial products, producing 128 bit carry and sum outputs. Because Booth encoding requires signed numbers with an even number of bits, N = (64 + 1 sign bit + 1 bit to create even result) = 66. Thus, there are 66/2=33 partial products produced. A carry-propagate adder (CPA) is used to add the Wallace tree carry and sum outputs.

Radix-4 Booth encoding looks at 3 bits at a time (i+1, i, and i-1) from the B operand to determine what multiple of the A operand to use for the partial product. The block that determines what multiple of the A operand to use is referred to as a Booth encoder. The Booth encoder forms the selects for the Booth mux which outputs the selected multiple of the A operand. One Booth encoder and Booth mux pair is required per partial product.

TABLE 7-7 Radix-4 Booth Encoding

i+1	*i*	*i-1*	*Multiple*
0	0	0	0
0	0	1	1A
0	1	0	1A
0	1	1	2A
1	0	0	-2A
1	0	1	-1A
1	1	0	-1A
1	1	1	0

A single-pass implementation is used for all multiply instructions, producing a throughput of one instruction every cycle.

If the unnormalized intermediate mantissa result is in the format 1X.XX, post-normalization is accomplished by shifting the mantissa right by 1 bit and adding one to the intermediate exponent.

The multiplier contains an independent multiplier adder, Normalization and round functions share the existing datapath used by add instructions.

FIGURE 7-13 Multiplier Block Diagram

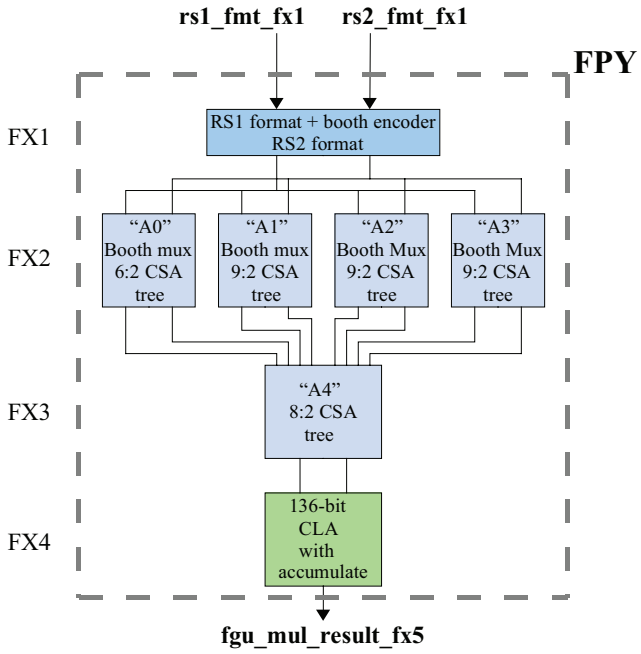
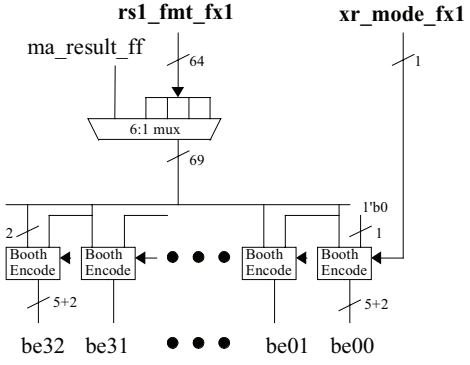


FIGURE 7-14 Multiplier Operand Format and Booth Encode



6:1 rs1 mux
0 : fmul8SUx16 or fmuld8SUx16
1 : fmul8x16AU or fmul8x16AL or fmul8x16
2 : fmul8ULx16 or fmuld8ULx16
3 : MA Int / XOR forward of last MA result
4 : MA Int / XOR unforwarded
5 : 64-bit data

FIGURE 7-15 Multiplier Operand Format and 9:2 Array

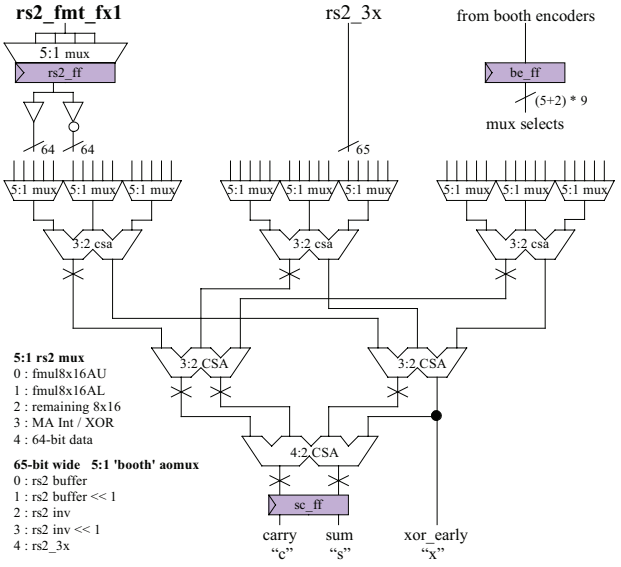


FIGURE 7-16 Multiplier 6:2 Array

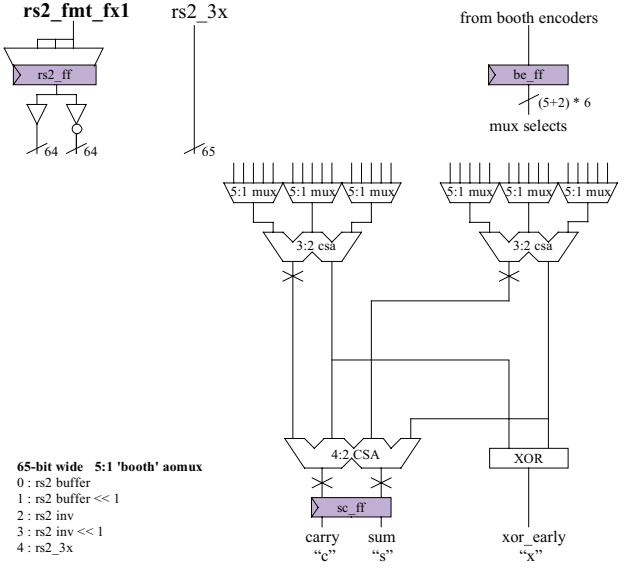


FIGURE 7-17 Multiplier 8:2 Array

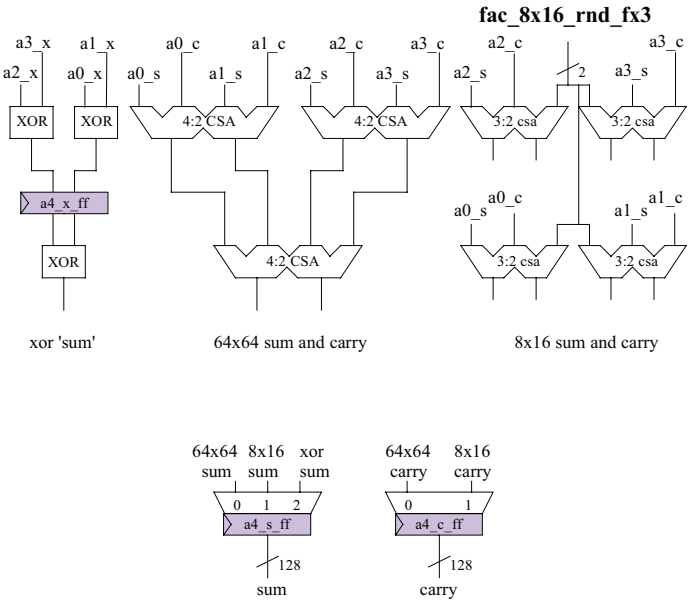
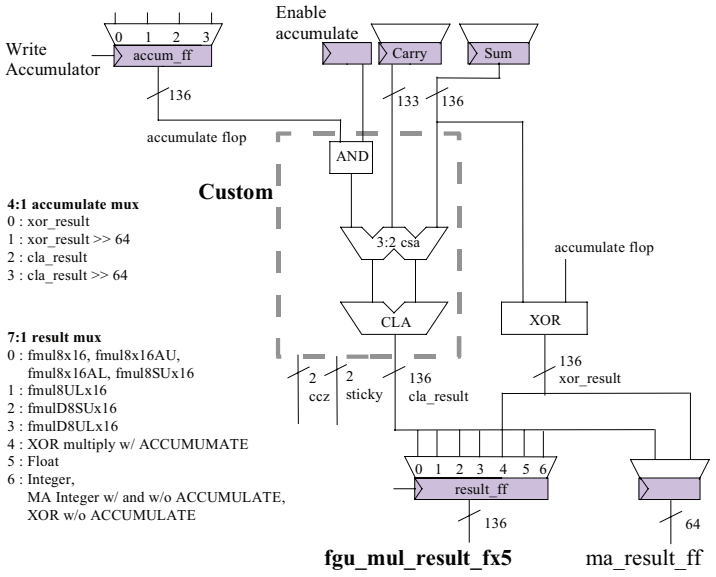


FIGURE 7-18 Multiplier 136-bit Adder with Accumulate



7.7.3 Exponent Datapath

TABLE 7-8 FPX Exponent Datapath Steps

Step	*Add Action*	*Multiply Action*	*Divide Action*

1	Compute the alignment shift count (SC) required to align the two operand mantissas:
	IF (Eb Ea) THEN (SC= (Eb-Ea)); ELSE (SC= (Ea-Eb))
	Compute the intermediate exponent:
	IF (Eb Ea) THEN (Eint=Eb); ELSE (Eint= Ea)
		Compute the intermediate exponent:
		Eint=Ea+Eb-bias
			Compute the intermediate exponent:
			Eint=Ea-Eb+bias

2	Compute the exponent adjust: Eadj=(number of leading zeros in the unnormalized intermediate mantissa)		Compute the exponent adjust: Eadj=1 if the unnormalized intermediate mantissa result is in the format 0.1X
3	Compute the exponent result: Eresult= Eint-Eadj+inc Where inc=1 if the mantissa rounder carry out is a one (Rcout=1) and the incremented intermediate mantissa result is selected, or if the unnormalized intermediate mantissa result is in the format 1X.XX.	Compute the exponent result: Eresult=Eint+inc Where inc=1 if the mantissa rounder carry out is a one (Rcout=1) and the incremented intermediate mantissa result is selected, or if the unnormalized intermediate mantissa result is in the format 1X.XX.	Compute the exponent result: Eresult=Eint-Eadj

The following definitions apply to TABLE 7-6:

Ea represents the biased exponent of operand A

Eb represents the biased exponent of operand B

bias=127 if SP, bias=1023 if DP

FIGURE 7-19 Exponent Input Format Muxes

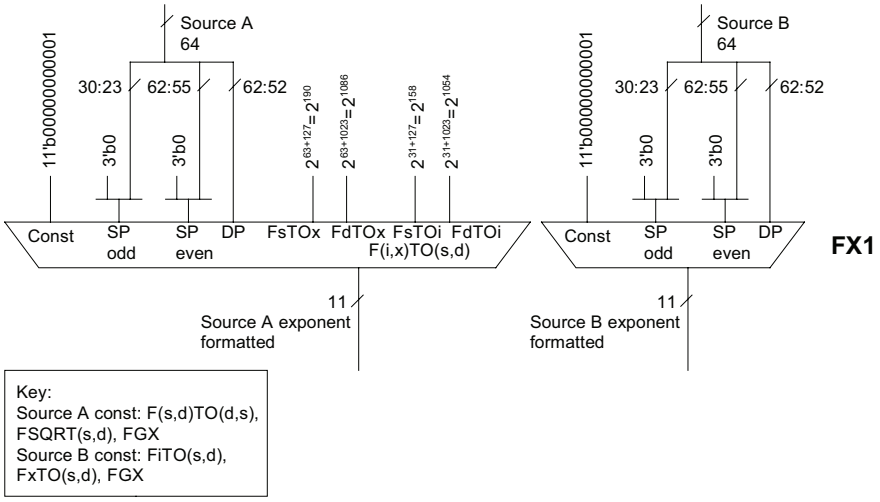


FIGURE 7-20 Auxiliary Exponent Input Format Muxes (FMUL/FDIV/FSQRT)

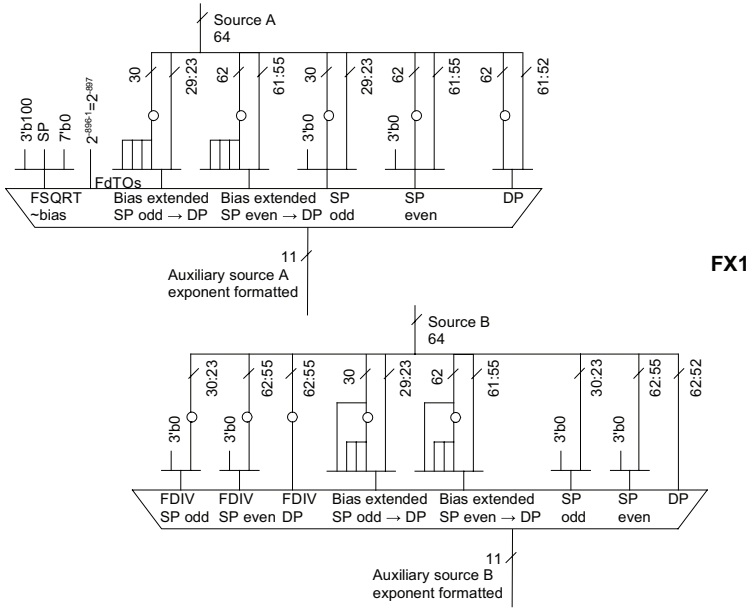
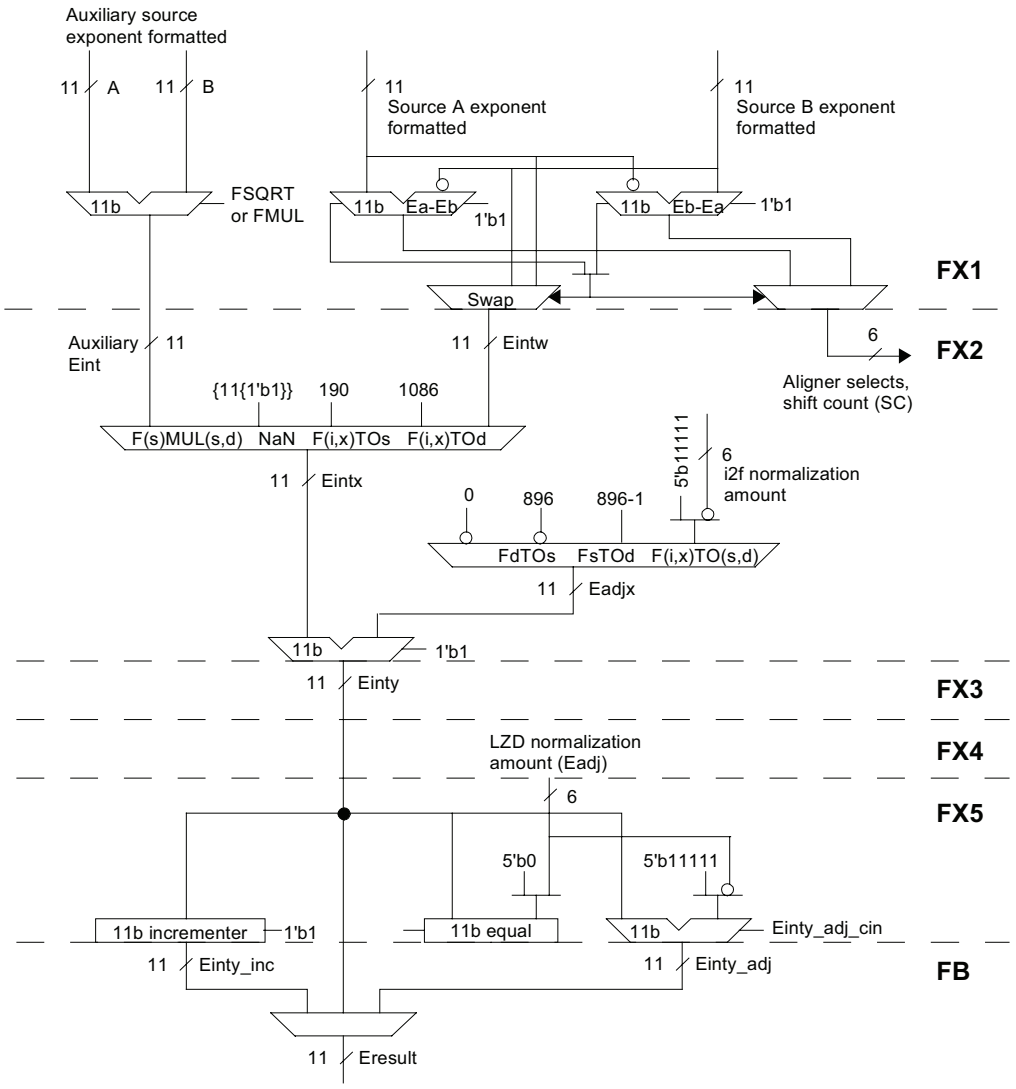


FIGURE 7-21 Exponent



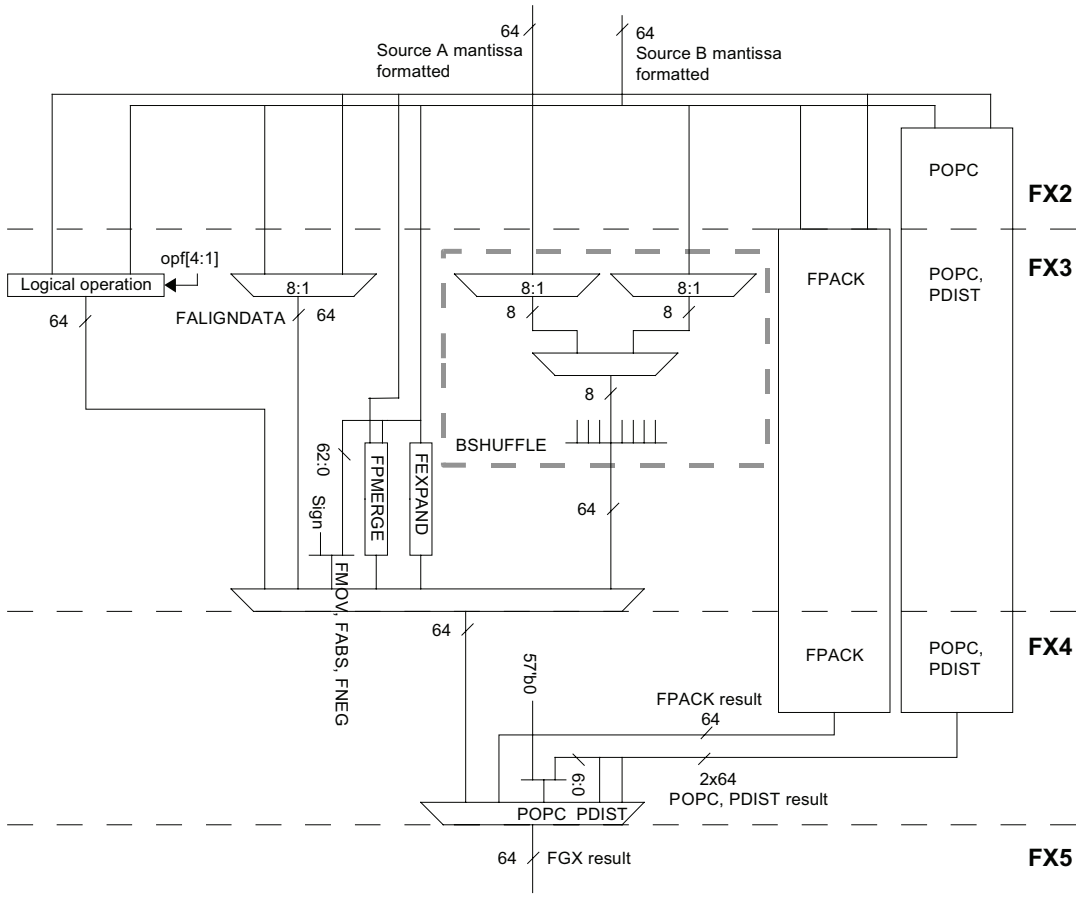
7.8 Graphics Execution Pipeline (FGX)

7.8.1 Functionality

- VIS 2.0 byte shuffle and data alignment instructions:
- BSHUFFLE, FALIGNDATA
- VIS 2.0 pixel formatting instructions:
- FEXPAND, FPMERGE, FPACKFIX, FPACK{16,32}
- VIS 2.0 logical instructions (32):
- FANDNOT1{s}, FANDNOT2{s}, FAND{s}, FNAND{s}, FNOR{s}, FNOT1{s}, FNOT2{s}, FONE{s}, FORNOT1{s}, FORNOT2{s}, FOR{s}, FSRC1{s}, FSRC2{s}, FXNOR{s}, FXOR{s}, FZERO{s}
- SPARC V9 FP absolute value, move, and negate instructions:
- FABS(s,d), FMOV(s,d), FMOV(s,d)cc, FMOV(s,d)r, FNEG(s,d)
- SPARC V9 population count instructions, and VIS 2.0 pixel distance instruction:
- POPC, POPCi, PDIST
- All FGX instructions are fixed latency, independent of operand values.
- Source operand format muxes (FX1) and output format muxes (FX5) are located in FPX.

7.8.2 Execution Datapath

FIGURE 7-22 FGX Execution Datapath Block Diagram



7.8.2.1 Byte Shuffle Instruction

BSHUFFLE concatenates two 64 bit floating-point registers, rs1 and rs2, to form a 16-byte value. rs1 is the upper half and rs2 is the lower half of the concatenated value. Bytes in the source value are numbered from most significant to least significant, with the most significant byte being byte zero. Eight bytes are extracted from the

source value based on the GSR.mask field.

TABLE 7-9 BSHUFFLE Destination Byte Selection

rd Byte	*rd Bit Range*	*Source*
0	63:56	rs byte[GSR.mask<31:28>]
1	55:48	rs byte[GSR.mask<27:24>]
2	47:40	rs byte[GSR.mask<23:20>]
3	39:32	rs byte[GSR.mask<19:16>]
4	31:24	rs byte[GSR.mask<15:12>]
5	23:16	rs byte[GSR.mask<11:8>]
6	15:8	rs byte[GSR.mask<7:4>]
7	7:0	rs byte[GSR.mask<3:0>]

7.8.2.2 Data Alignment Instruction

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

TABLE 7-10 Data Alignment Instruction

GSR.align[2:0]	*rd*
0	rs1[63:0]
1	rs1[55:0],rs2[63:56]
2	rs1[47:0],rs2[63:48]
3	rs1[39:0],rs2[63:40]
4	rs1[31:0],rs2[63:32]
5	rs1[23:0],rs2[63:24]
6	rs1[15:0],rs2[63:16]
7	rs1[7:0],rs2[63:8]

7.8.2.3 Pixel Formatting Instructions

Pixel formatting instructions include FEXPAND, FPMERGE, FPACKFIX, and FPACK{16,32}.

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

FPMERGE interleaves four corresponding 8 bit unsigned integers in rs1 and rs2, to produce a 64 bit value in the rd register.

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

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

FPACK32 takes two 32 bit fixed values in rs2, scales, truncates and clips them into two 8 bit unsigned integers. The two 8 bit integers are merged at the corresponding least significant byte positions of each 32 bit word in rs1 left shifted by 8 bits. The 64 bit result is stored in the rd register.

FIGURE 7-23 FPACK {FIX, 16, 32} Data Result Implementation

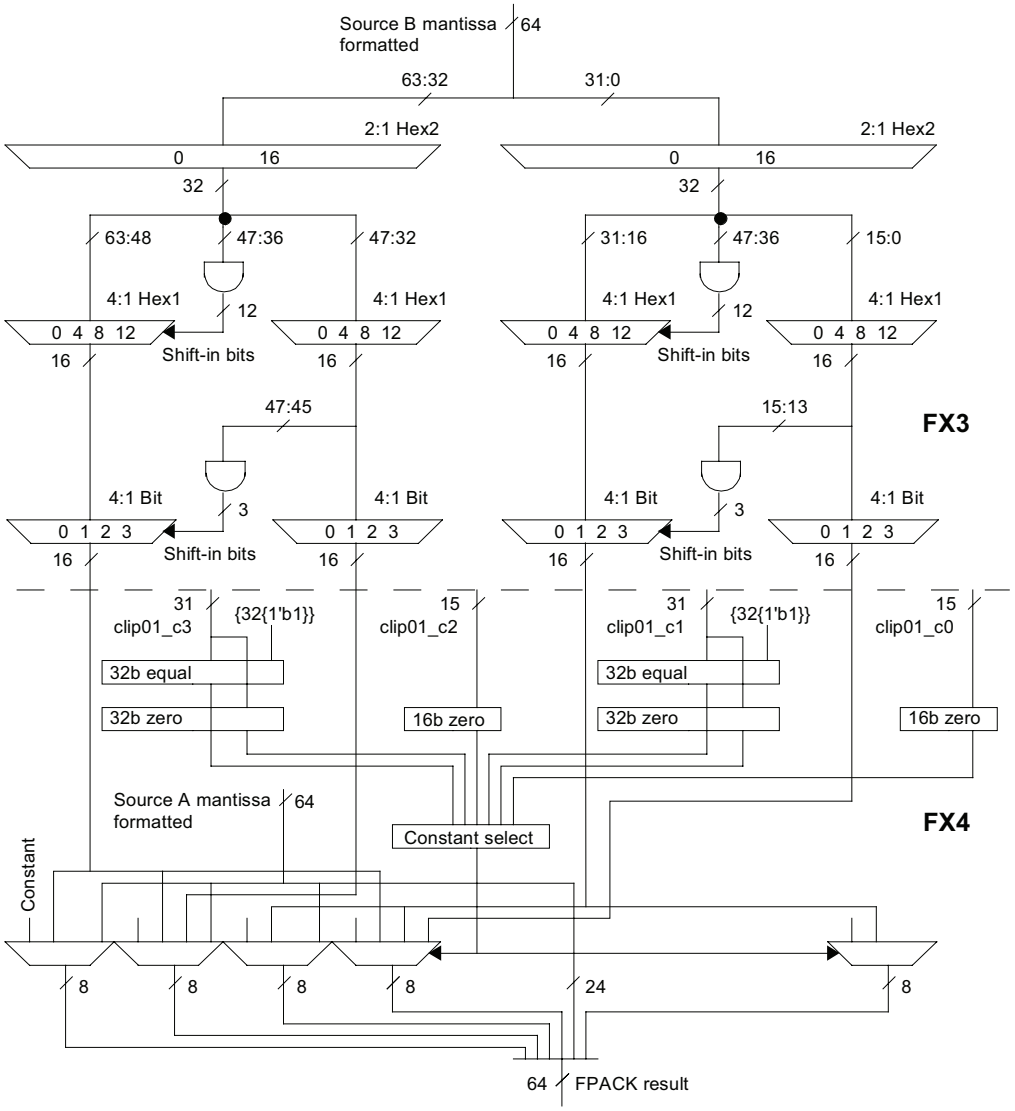
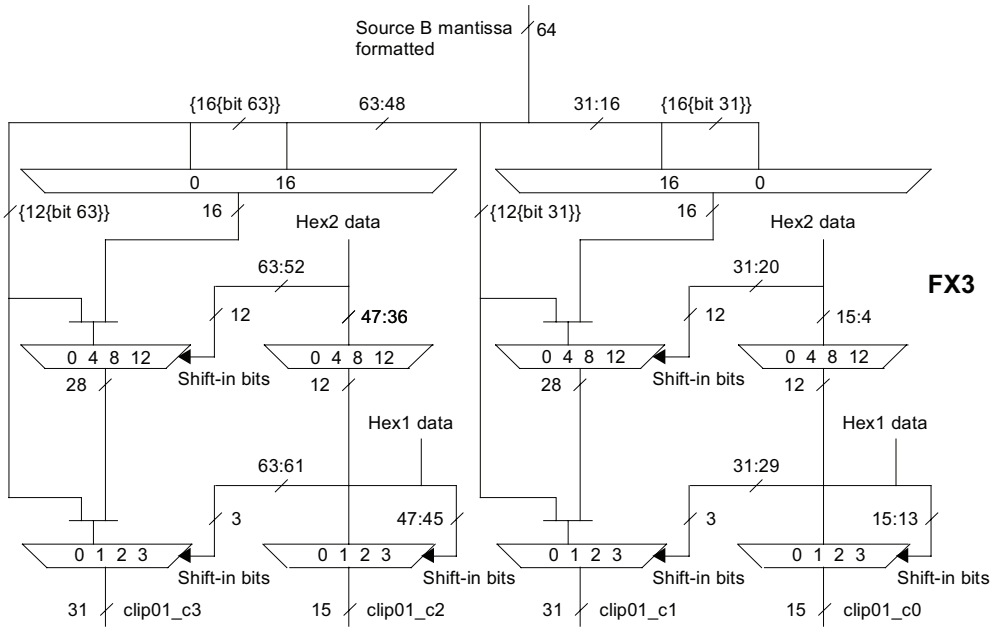


FIGURE 7-24 FPACK {FIX, 16, 32} Clipping Implementation



7.8.2.4 Logical Instructions

TABLE 7-11 Logical Instructions

Instruction	*Opcode*	*Description*
FZERO	0000	Zero fill
FNOR	0001	Logical NOR
FANDNOT2	0010	rs1 AND (negated rs2)
FNOT2	0011	Negate rs2
FANDNOT1	0100	(negated rs1) AND rs2
FNOT1	0101	Negate rs1
FXOR	0110	Logical XOR
FNAND	0111	Logical NAND
FAND	1000	Logical AND
FXNOR	1001	Logical XNOR

TABLE 7-11 Logical Instructions *(Continued)*

FSRC1	1010	Copy rs1
FORNOT2	1011	rs1 OR (negated rs2)
FSRC2	1100	Copy rs2
FORNOT1	1101	(negated rs1) OR rs2
FOR	1110	Logical OR
FONE	1111	One fill

Note – The single precision version of these logical instructions are executed the same way. The operands are given to FGX in the upper 32 bits of rs1 and rs2. The FGX result is in the upper 32 bits of rd.

These logical instructions are implemented with a 4:1 mux structure built from a NAND-NAND (AND-OR) gate organization.

7.8.2.5 Move Instructions

Move instructions include FMOV(s,d), FMOV(s,d)cc, FMOV(s,d)r, FABS(s,d), and FNEG(s,d).

FMOV(s,d) copies the contents of rs2 to the destination register rd.

FMOV(s,d)cc and FMOV(s,d)r copy the contents of rs2 to the destination register rd if the condition is satisfied by the selected condition code. If the condition is false, then the destination register is not written. The status of the condition is specified to FGX by the IFU. Note that the single precision version of this instruction is executed the same way. The operand is given to FGX in the upper 32 bits of rs2 and the FGX result is in the upper 32 bits of rd.

FABS(s,d) copies the contents of rs2 to the destination register rd with the sign bit (bit 63) cleared to zero. Note that the single precision version of this instruction is executed the same way. The operand is given to FGX in the upper 32 bits of rs2 and the FGX result is in the upper 32 bits of rd.

FNEG(s,d) copies the contents of rs2 to the destination register rd with the sign bit (bit 63) negated. Note that the single precision version of this instruction is executed the same way. The operand is given to FGX in the upper 32 bits of rs2 and the FGX result is in the upper 32 bits of rd.

7.8.2.6 Population Count and Pixel Distance Instructions

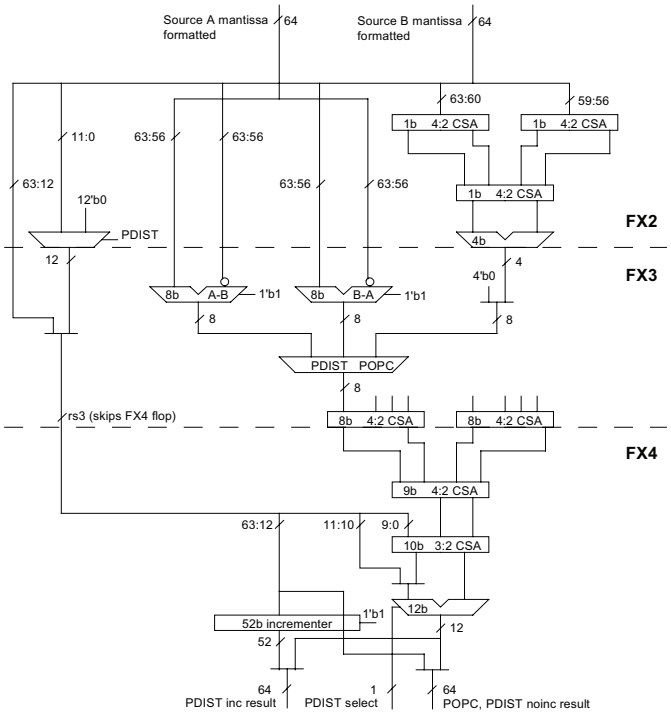
Population count and pixel distance instructions include POPC, POPCi, and PDIST.

PDIST is a three source instruction, and requires two cycles to read the sources from the FRF which has only two read ports. No FGU executed instruction may be issued the cycle after PDIST is issued. PDIST has a fixed six cycle execution latency, and a throughput of one instruction every two cycles.

PDIST takes eight unsigned 8 bit values contained in the 64 bit rs1 and rs2 registers, subtracts the corresponding 8 bit values in rs1 and rs2 (that is, rs1-rs2), sums the absolute values of each of the eight differences, then adds the integer in the 64 bit rd register. The result is stored in rd.

POPC{i} implementation leverages the PDIST datapath. POPC counts the number of one bits in rs2 if i=0, or the number of one bits in sign_ext(simm13) if i=1, and stores the count in rd.

FIGURE 7-25 POPC and PDIST Implementation



7.9 Floating-Point Divide and Square Root Pipeline (FPD)

7.9.1 Functionality

- FDIV/FSQRT always produce a fixed execution latency: 19 SP, 33 DP.
- IFDIV/FSQRT early completion is provided for special cases and IEEE exceptions
- In addition to the floating-point divide instructions, FPD implements the SPARC V9 integer divide instructions. The EXU provides two 64 bit source operands with the appropriate sign extended immediate data for rs2, and provides rs1 and rs2 sign extension and zero fill formatting when required.

IDIV produces a data dependent variable execution latency of between 12 and 41.

FPD generates the following IDIV default response results:

- 64 bit max -integer
- 64 bit 2^{32}-1
- 64 bit 2^{31}-1
- 64 bit -2^{31}

FPD instructions execute in a dedicated datapath and are non-blocking with respect to FPX and FGX.

FPD uses an SRT algorithm generating 2 bits per cycle (radix-4) for divide and square root.

FGU can handle up to two outstanding FDIV/IDIV/FSQRT instructions.

7.9.2 Early Completion

Results for certain floating point divide and square root instructions can be determined without requiring that a divide or square root calculation be performed using the source fraction(s). In these cases the source's sign, exponent, and a zero fraction detection is sufficient to generate the expected result. The FPX pipeline leverages existing hardware to generate all exponent results, handle special cases (sources or results which are zero, denormal, NaN, or infinity) and handle IEEE exceptions.

Early completion is provided for special cases and IEEE exceptions to decrease floating point divide and square root instruction latency. Divide latency is especially important in N2 given that eight threads share a single non-pipelined FPD, and a given thread is switched out until the divide instruction is complete.

Given the source(s) for a floating point divide or square root instruction, FPX uses the sign(s), exponent(s), and a zero fraction detector to calculate (A) the result sign and the intermediate exponent, or (B) the final sign, exponent and fraction result. Case B applies to special cases and IEEE exceptions, otherwise case A applies.

If case A applies, the unified FPX execution pipeline calculates the intermediate exponent for the divide or square root instruction as if the instruction were fully pipelined. It then stores the result sign and intermediate exponent until FPD has completed calculation of the fraction result. The sign, exponent and fraction are then merged and written into the FRF.

The FGU can handle up to two outstanding divide or square root instructions. It is possible for the two instructions to complete execution out of order, or simultaneously, with respect to each other if the first instruction issued to the FGU is case A and the second is case B. By definition, the two outstanding divide or square root instructions must have different TIDs.

If case B applies, the divide or square root instruction executes in a fully pipelined fashion within FPX. FPD is not required to participate in the generation of the result.

Case B for FDIV applies under the following conditions:

1. Either source (or both) is NaN
2. Either source (or both) is infinity
3. Either source (or both) is zero
4. Either source (or both) is denormalized
5. Overflow exception occurs. This can be fully detected early, however, N2 has a 1-bit uncertainty in the early detection such that Eint=Emax+1 won't create an early overflow detection in case it is later determined that Eadj=1. Full early detection may be accomplished as follows (see TABLE 7-7):

   Eadj=0 if mantissa_a ≥ mantissa_b

   Eadj=1 if mantissa_a < mantissa_b

6. Underflow exception occurs. Like overflow, this can be fully detected early, however, N2 has a 1-bit uncertainty in the early detection such that it is not known if Eint=Emin will create an underflow. The final determination is made later when Eadj is known. (See TABLE 7-7).

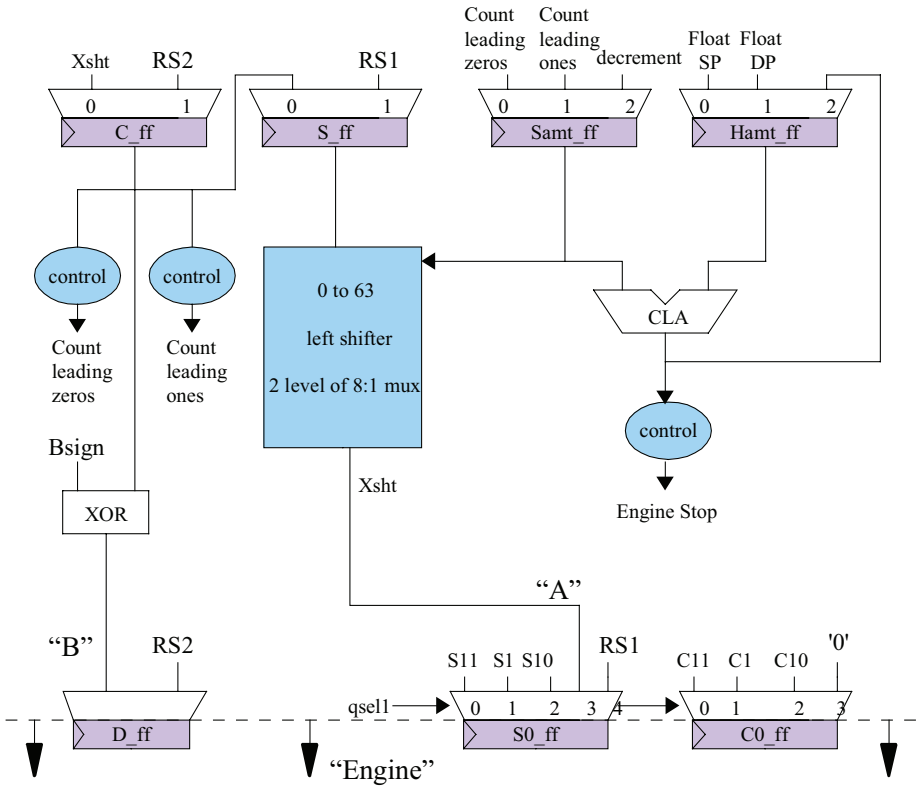
Case B for FSQRT applies under the following conditions:

1. Source is NaN
2. Source is infinity
3. Source is zero
4. Source is denormalized
5. Source is negative (not including negative zero)

TABLE 7-12 FDIV and FSQRT Special Cases

rs 1	rs 2	FDIV(s,d) result	FSQRT(s,d) result
Norm	0	Infinity	
Norm	Infinity	0	
Norm	NaN	QNaN	
Denorm	0	Infinity	
Denorm	Infinity	0	
Denorm	NaN	QNaN	
0	Norm	0	
0	Denorm	0	
0	0	QNaN	
0	Infinity	0	
0	NaN	QNaN	
Infinity	Norm	Infinity	
Infinity	Denorm	Infinity	
Infinity	0	Infinity	
Infinity	Infinity	QNaN	
Infinity	NaN	QNaN	
NaN	Norm	QNaN	
NaN	Denorm	QNaN	
NaN	0	QNaN	
NaN	Infinity	QNaN	
NaN	NaN	QNaN	
	+0		+0
	-0		-0
	+Infinity		+Infinity
	-Infinity		QNaN
	NaN		QNaN
	+Denorm		+0
	-Denorm		QNaN
	Negative and non-zero		QNaN

FIGURE 7-26 Integer Divide Pre-Engine



7.9.3 SRT Algorithm

The SRT algorithm follows the basic form:

```
For i = 1 to N
  Begin
    PR(i+1) = 2 * PR(i) - q(i+1) * D    // This is a single SRT step
  End
```

Where:

PR = partial remainder

D = divisor

q is the quotient digit based on the sign of D and PR(i) {+1,0,-1}

In a conventional implementation, the operation above is performed using a carry-lookahead adder (CLA) to compute the next partial remainder (PR). However, in the N2 implementation the CLA is replaced with a carry save adder (CSA). Relative to the CLA, the CSA has both area and timing advantages. Using the CSA, the next PR is left in the redundant form of sum and carry. By using CSA addition, multiple SRT steps are possible in a single cycle (see FIGURE 7-27). This results in a low latency floating-point divide.

Utilizing this low latency SRT divide algorithm for integer division presents two problems:

1. Integer numbers do not always fall within the strict range required by the algorithm.

Determination of q(i+1) poses some difficulty. If the PR were kept in a non-redundant form (CLA), we could examine the sign of PR(i) and D to determine the q(i+1). By examining the upper 4 bits of the redundant form sum and carry, it has been shown that sufficient information exists to determine q(i+1). However, this requires that that the divisor be kept within a very strict range of numbers. For floating-point, given that both operands are normalized, this restriction is not a problem.

Integer divisors do not always fall within this strict range. For example, consider the range possible with 8-bit signed numbers. A divisor of zero is not included since this answer is known without having to perform the SRT loop.

0000_0001 (+1)

0111_1111 (+127)

The solution is to include a left shifter prior to the divide engine to remove the leading sign bits from the divisor. This is called the integer divide pre-engine (see FIGURE 7-26). In this way, the integer divisor looks similar to a normalized floating-point number. This normalized divisor satisfies the algorithm's divisor range requirement.

The pre-engine executes over two cycles for a given operand. In the first cycle, the number of leading sign bits is determined. Call this quantity LS2. In the second cycle, the divisor is left shifted by LS2.

2. The final quotient must be right shifted to correct for the divisor normalization performed by the pre-engine.

0101_0000 ÷ 0000_0001 = 0101_0000 (LS2 = 7)

0101_0000 ÷ 0001_0000 = 0000_0101 (LS2 = 3)

In the two examples above, the normalized divisor in both cases is 1000_0000. The final quotient must be right shifted to compensate for the left shifting done by the pre-engine. The right shifting of the quotient, if implemented, is based on LS2 and adds one additional cycle at the end of the divide. This also requires a separate shifter from the one used within the pre-engine (right shifter vs. left shifter).

Alternatively, the dividend, in addition to the divisor, is passed through the pre-engine normalization shifter. Both operands are normalized using the same pre-engine hardware over a total of three cycles (instead of two cycles if only the divisor is passed through the pre-engine).

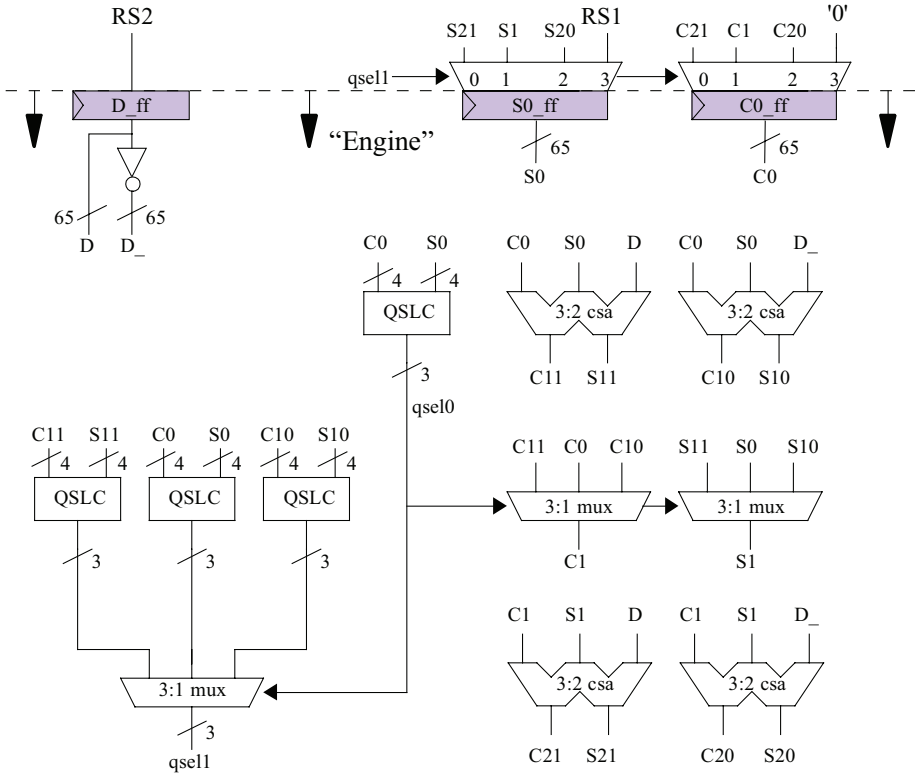
Cycle 1: Count leading sign bits for divisor (LS2)

Cycle 2: Left shift divisor by LS2; Count leading sign bits for dividend (LS1)

Cycle 3: Left shift dividend by LS1

By having both the dividend and divisor in a normalized form, the SRT loop will immediately compute significant quotient digits.

FIGURE 7-27 Divide Engine



7.10 State Registers, Exceptions, and Traps

7.10.1 Floating-point Registers State (FPRS) Register

The architected FPRS for each thread is maintained within the FGU.

The FPRS is accessed with RDASR and WRASR instructions using ASR 6.

FPRS access via WRASR and RDASR is serializing for a given thread (all previous FPops have completed, then WRASR/RDASR access of FPRS completes prior to issuing subsequent operations which access the FPRS).

FGU provides the FPRS.fef bit to the IFU for each TID (used by IFU to determine fp_disabled).

FPRS.du and FPRS.dl are maintained precisely by the FGU. Only instructions which successfully complete and update the architected FRF will set FPRS.du and FPRS.dl. These bits are never set pessimistically.

7.10.2 Graphics State Register (GSR)

The architected GSR for each thread is maintained within the FGU.

The GSR is accessed with implementation dependent RDASR and WRASR instructions using ASR 19 (13_{16}). Reserved bits are read as zero.

Each EXU provides GSR.mask and GSR.align fields to the FGU. GSR.scale is set by WRASR only. GSR.im and GSR.irnd can be set by SIAM.

GSR access via WRASR and RDASR is serializing for a given thread (all previous FPops have completed, then WRASR/RDASR access of GSR completes prior to issuing subsequent operations which access the GSR).

GSR access via BMASK, ALIGNADDRESS, or SIAM, is not required to be serializing. The FGU pipelines these GSR accesses to avoid serializing requirements.

7.10.3 Floating-Point State Register (FSR)

The architected FSR for each thread is maintained within the FGU.

The lower 32 bits of the FSR are accessed by the STFSR and LDFSR instructions. All 64 bits of the FSR are accessed by the STXFSR and LDXFSR instructions. The ver, ftt, and reserved fields are not modified by LDFSR or LDXFSR. Reserved bits are read as zero.

LD(X)FSR does not modify FSR.ftt. ST(X)FSR clears FSR.ftt if the store completes without error. LD(X)FSR may set FSR.TEM and FSR.cexc fields, but will never cause a fp_exception_ieee_754 trap.

Each of the five IEEE exception status flags and associated trap enables are supported (invalid operation, zero divide, overflow, underflow, inexact).

All four IEEE round modes are supported in hardware.

IEEE traps enabled mode: if an instruction generates an IEEE exception for which the corresponding trap enable is set, then a fp_exception_ieee_754 trap is generated and results are inhibited by the FGU.

A conditional FMOV instruction clears FSR.ftt and FSR.cexc, regardless of whether the condition is true or false.

7.10.3.1 Non-Standard Floating-Point Mode

Non-standard floating-point mode (FSR.ns=1) is available for flushing denormalized operands and results to signed zero.

If a floating-point source operand is denormalized, it is replaced by a floating-point zero value of the same sign. An inexact or invalid exception may be signaled, or if the divisor is flushed to zero then a division by zero exception may be signaled.

Dual operand instructions do not generate an inexact exception if one operand is denormalized, the other operand is NaN, infinity, or zero, and the expected result is NaN, infinity, or zero. For example:

- denorm + QNaN = QNaN
- denorm + SNaN = QNaN (with invalid exception)
- denorm + ∞ = ∞
- denorm × 0 = 0
- denorm × ∞ = QNaN (with invalid exception, since denorm operand is flushed to zero).

If a floating-point operation generates a denormalized value, the value is replaced with a floating-point zero value of the same sign and inexact and underflow exceptions are signaled.

If GSR.im=1, then the value of FSR.ns is ignored and the processor operates as if FSR.ns=0.

Non-standard floating-point mode does not apply to the instructions listed below. Thus, a denormalized source operand is never flushed to zero for these instructions

- FCMP(s,d)
- FCMPE(s,d)
- FABS(s,d)
- FMOV(s,d), FMOV(s,d)cc, FMOV(s,d)r
- FNEG(s,d)

7.10.4 Exceptions and Traps

There are five IEEE exception status flags:

- invalid (nv)

- overflow (of)
- underflow (uf)
- division-by-zero (dz)
- inexact (nx)

The FSR contains a 5 bit field for current exceptions (FSR.cexc) and a 5 bit field for accrued exceptions (FSR.aexc). Each IEEE exception status flag has a corresponding trap enable mask (TEM) in the FSR:

- NVM
- OFM
- UFM
- DZM
- NXM

FSR.TEM bits are required for the following cases:

1. fp_exception_ieee_754 trap detection. If a FPop generates an IEEE exception (nv, of, uf, dz, nx) for which the corresponding trap enable (TEM) is set, then a fp_exception_ieee_754 trap is caused. FSR.cexc field has one bit set corresponding to the IEEE exception, and FSR.aexc field is unchanged.

2. Clear FSR.nxc if an overflow (underflow) exception does trap because FSR.OFM (FSR.UFM) is set, regardless of whether FSR.NXM is set. Set FSR.ofc (FSR.ufc).

3. Clear FSR.ofc (FSR.ufc) if overflow (underflow) exception traps and FSR.OFM (FSR.UFM) is not set and FSR.NXM is set. Set FSR.nxc.

The FPX and FPD execution pipelines do not receive FSR.TEM bits and always assume that each of the five TEM bits are zero (all traps are disabled). The FGU FSR logic handles cases where one or more of the FSR.TEM bits are non-zero.

There are three FGU related trap types tracked in the architected trap type (TT) register located in the TLU:

1. fp_disabled. fp_disabled is detected by the IFU.

2. fp_exception_ieee_754. If an FPop generates an IEEE exception (nv, of, uf, dz, nx) for which the corresponding trap enable (TEM) is set, then an fp_exception_ieee_754 trap is caused. This is detected by the FGU.

3. fp_exception_other. In the OpenSPARC T2 implementation, fp_exception_other is always due to unfinished_FPop. unfinished_FPop is detected by the FGU.

FSR.ftt is set to identify the cause of the trap: fp_exception_ieee_754, or unfinished_FPop.

Certain denormalized operands or expected results may generate an unfinished_FPop trap to software, indicating that the FGU was unable to generate the correct results. The conditions which generate an unfinished_FPop trap are consistent with UltraSPARC I/II..

7.10.4.1 Exception Trap Prediction

OpenSPARC T2 has different pipeline depths for integer and FGU operations. Specifically, the FGU pipeline is four pipeline stages (FX4, FX5, FB, FW) longer than the integer pipeline. The different pipeline depths create potential exception hazards between FGU operations and integer operations. An integer operation subsequent to an FGU operation can update architectural state before the FGU exception trap status is known.

One method to eliminate this hazard is to ensure that an integer operation does not start execution (is not picked) until five cycles after an FGU operation has begun execution (has been picked). This ensures that the integer operation does not update architectural state in the W stage if the FGU operation takes an exception trap. The performance impact makes this potential solution unattractive

A second method is to equalize the depth of the integer and FGU pipelines. The required die area to increase the depth of the integer pipelines, along with the added complexity of bypassing integer results from each stage of the deep pipeline makes this potential solution unattractive.

Floating point exception trap prediction is one form of thread speculation supported by N2. It is used to maintain a precise exception model, and allow the FGU to support full floating-point single thread pipelining, independent of IEEE trap enables, for all IEEE exception trap types (invalid, overflow, underflow, division-by-zero and inexact). During FX1, the FGU performs a fast (one pipeline stage) and accurate prediction to determine whether an FGU operation may generate an exception trap. The FGU transmits the prediction to the TLU in the FX2/B stage. (Other units also transmit trap status during the B stage.) If the FGU signals an exception trap prediction then subsequent instructions in that thread are flushed. If the FGU does not signal an exception trap prediction then it is guaranteed that an exception trap is not possible during the execution of the instruction (does not apply to IDIV, FDIV, or FSQRT instructions because these instructions stall the thread). The final exception trap detection is reported to the TLU in the FW stage. If an instruction reports an exception trap in the FW stage then the appropriate trap is taken.

A performance penalty for a given thread is incurred if the FGU signals an exception trap prediction, but the instruction does not actually generate an exception trap. The performance penalty is a result of the subsequent instructions in that thread being flushed and the core pipeline being restarted.

Upon receiving an exception trap prediction from the FGU, the TLU determines if the subsequent instruction is from the same thread. If it is, the TLU sends a flush to the EXUs, the LSU, and the FGU in the FX3/W stage of the predicting instruction (which is the FX2/B stage of the subsequent instruction). The TLU also informs the IFU in the FX3/W stage that the processor must flush the relevant thread. The IFU detects and flushes any later instructions from this thread that may be in the machine. The TLU sends the PC and NPC of the instruction after the FGU predicting instruction to the IFU to minimize the mispredict penalty (in the case where an exception trap did not occur). The IFU can immediately start fetching the PC and NPC; the thread does not stall. If the FGU prediction is correct, the TLU reports a flush to the IFU in the FW+1 cycle of the predicting FGU instruction, and the IFU flushes the refetch of the subsequent instructions, and starts fetching the PC of the FGU exception trap handler. If the FGU prediction is incorrect, the TLU does not flush again and does not send any other PC or NPC to the IFU.

The TABLE 7-13 below summarizes the possible prediction and detection combinations. A fatal case exists if the FGU doesn't predict an FGU exception trap but an FGU exception trap is later detected and taken (does not apply to IDIV, FDIV, or FSQRT instructions). This case is fatal since an integer operation subsequent to the predicting FGU operation may update architectural state.

TABLE 7-13 FGU Exception Trap Prediction and Detection Cases

Speculation Enable	*Predict FP Exception Trap*	*Detect FP Exception Trap*	*Action*
0	0	0	Prediction Ignored
0	0	1	Prediction Ignored (non-fatal)
0	1	0	Prediction Ignored (No performance Penalty)
0	1	1	Prediction Ignored
1	0	0	Desired Behavior
1	0	1	Fatal
1	1	0	Performance Penalty
1	1	1	Desired Behavior

With speculation disabled the FGU operates in non-pipelined mode for a given thread (threads stall on FGU instructions), sustaining a rate of one FGU instruction every 7 cycles. No FGU instruction issues until the final exception trap status of the previous FGU instruction in that thread is known

Integer divide, floating-point divide, and square root instructions do not participate in exception trap prediction. These long latency instructions are executed in the FPD pipeline. Once an FPD instruction has been issued, no other instruction (from that thread) can be issued until the older FPD instruction has completed or has been flushed. FPD provides non-speculative exception trap information during the FW stage.

The FGU predicts invalid, overflow, underflow, and unfinished_FPop exceptions.

- Invalid and overflow predictions do not cause an exception trap prediction unless the corresponding FSR.TEM is set.
- An underflow prediction causes an exception trap prediction, dependent on FSR.UFM, FSR.ns, and GSR.im. FMUL and FdTOs are also dependent on gross underflow.
- Instructions capable of generating unfinished_FPop exceptions may cause an exception trap prediction, dependent on FSR.ns and GSR.im. FMUL and FdTOs are also dependent on gross underflow.
- Instructions which can set the inexact exception always cause an exception trap prediction if FSR.NXM is set. FsMULd and FsTOd are special cases.

TABLE 7-14 IEEE Exception Trap Prediction Cases

Instruction	Exception trap prediction case					
	invalid (FSR.NVM=1)	divide by zero (FSR.DZM=1)	overflow (FSR.OFM=1)	underflow	inexact (FSR.NXM=1)	unfinished_FPop (FSR.ns=0 or GSR.im=1)
FABS(s,d)	cannot generate exception trap prediction					
FADD(s,d FSUB(s,d))	Implemented prediction: SNaN or ∞ source More ideal prediction: SNaN source or (∞ - ∞)		Implemented prediction: effective_addition and (Ea= Emax or Eb= Emax)	Implemented prediction: (FSR.ns=0 or GSR.im=1 or FSR.UFM= 1)and effective_subtraction and (~(Ea≥N) and ~(Eb≥ N) and (SC=0 or SC= 1)); where N=54 if DP, N=25 if SP	Implemented prediction: always if FSR.NXM=1 More ideal prediction: qualify with 0,NaN, or ∞ source	Implemented prediction: denorm source denorm result covered by underflow prediction More ideal prediction: qualify denorm source with: one source is denorm and the other source is NaN, ∞, or 0, and the expected result is NaN, ∞, or 0

TABLE 7-14 IEEE Exception Trap Prediction Cases *(Continued)*

FCMP(s,d)	Implemented prediction: SNaN or ∞ source More ideal prediction: SNaN source					
FCMPE(s,d)	Implemented prediction: NaN or ∞ source More ideal prediction: NaN source					
FDIV(s,d)	cannot generate exception trap prediction					
FiTOs FxTO(s,d)					Implemented prediction: always if FSR.NXM=1	
FiTOd	cannot generate exception trap prediction					
FMOV(s,d)	cannot generate exception trap prediction					
FMOV(s,d)cc	cannot generate exception trap prediction					
FMOV(s,d)r	cannot generate exception trap prediction					

TABLE 7-14 IEEE Exception Trap Prediction Cases *(Continued)*

FMUL(s,d)	Implemented prediction: SNaN or ∞ source More ideal prediction: SNaN source or (∞ × 0)		Implemented prediction: (Ea+Eb-bias) ≥ Emax	Implemented prediction: ((FSR.ns=0 or GSR.im=1 or FSR.UFM=1) and ((Ea+Eb-bias) < Emin)) and ~(gross_underflow and FSR.UFM=0) gross_underflow = (((Ea+Eb-bias) -N) and ~(Sr=0 and FSR.rd=2) and ~(Sr=1 and FSR.rd=3)); where N=54 if DP, N=25 if SP	Implemented prediction: always if FSR.NXM=1 More ideal prediction: qualify with 0,NaN, or ∞ source	Implemented prediction: denorm source and ~(gross_underflow and FSR.UFM=0); see underflow prediction for gross_underflow definition denorm result covered by underflow prediction More ideal prediction: qualify denorm source with: one source is denorm and the other source is NaN, ∞, or 0, and the expected result is NaN, ∞, or 0
FNEG(s,d)	cannot generate exception trap prediction					
FsMULd	Implemented prediction: SNaN or ∞ source More ideal prediction: SNaN source or (∞ × 0)					Implemented prediction: denorm source
FSQRT(s,d)	cannot generate exception trap prediction					

TABLE 7-14 IEEE Exception Trap Prediction Cases *(Continued)*

F(s,d)TOi F(s,d)TOx	Implemented prediction: NaN, ∞ , or large source (source >2E or source<-2E) note: FsTOi pessimistically predicts if source= -2Ex1.0 note: FsTOi pessimistically predicts if source= - 2Ex1.[31'b0.21'dont_care} note: F(s,d)TOx pessimistically predicts if source= -2Ex1.0 More ideal prediction: source >2E or source<-(2E+1) Where E=31 (unbiased) if F(s,d)TOi, E=63 (unbiased) if F(s,d)TOx				Implemented prediction: always if FSR.NXM=1 More ideal prediction: qualify with 0,NaN, or ∞ source	Implemented prediction: denorm source

TABLE 7-14 IEEE Exception Trap Prediction Cases *(Continued)*

FsTOd	Implemented prediction: SNaN or ∞ source More ideal prediction: SNaN source				Implemented prediction: always if FSR.NXM=1 and FSR.ns=1 and GSR.im=0 and (denorm or 0 source) More ideal prediction: qualify with 0,NaN, or ∞ source	Implemented prediction: denorm source
FdTOs	Implemented prediction: SNaN or ∞ source More ideal prediction: SNaN source		Implemented prediction: (Eb-896) ≥ Emax source is not NaN or ∞	Implemented prediction: ((FSR.ns=0 or GSR.im=1 or FSR.UFM=1) and ((Eb-896) < Emin)) and ~(gross_underflow and FSR.UFM=0) gross_underflow = (((Eb-896) -25) and ~(Sr=0 and round_mode=2) and ~(Sr=1 and round_mode=3)) Source is not 0 or denorm	Implemented prediction: always if FSR.NXM=1 More ideal prediction: qualify with 0,NaN, or ∞ source	Implemented prediction: denorm source and ~(gross_underflow and FSR.UFM=0); see underflow prediction for gross_underflow definition denorm result covered by underflow prediction

7.10.4.2 Inhibited Results

fp_exception_ieee_754, unfinished_FPop, enabled FRF ECC UE/CE, and IFU/TLU initiated flush must inhibit FGU results. To properly inhibit results, the following actions are taken by the FGU:

- Signal fp_exception_ieee_754, unfinished_FPop, or FRF ECC UE/ CE to the TLU as appropriate
- Disable FRF write (destination register is unchanged)
- FSR condition codes (fcc) are unchanged
- FSR.aexc field is unchanged
- In the case of fp_exception_ieee_754, FSR.cexc field has one bit set corresponding to the IEEE exception. Otherwise, FSR.cexc is unchanged.
- In the case of fp_exception_ieee_754, or unfinished_FPop. FSR.ftt is set to identify the cause of the trap. Otherwise, FSR.ftt is unchanged.

7.10.4.3 Overflow, Underflow, and Gross Underflow

Overflow occurs when the magnitude of what would have been the rounded result (had the exponent range been unbounded) is greater than the magnitude of the largest finite number of the specified precision. FGU supports all overflow conditions.

The underflow exception condition is defined separately for the trap enabled and trap disabled states.

- FSR.UFM=1: underflow occurs when the intermediate result is "tiny"
- FSR.UFM=0: underflow occurs when the intermediate result is "tiny" and there is "loss of accuracy"

A tiny result is detected before rounding, when a nonzero result value computed as though the exponent range were unbounded would be less in magnitude than the smallest normalized number.

Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded (inexact condition).

Because the FGU only supports gross underflow, as described below, if the tiny result condition is met, and an unfinished_FPop trap is not taken, then the loss of accuracy condition must, by definition, also be met. Thus, FSR.UFM has no impact on whether underflow occurs (but, FSR.UFM and FSR.NXM do impact whether underflow is reported via FSR.ufc).

FGU provides only limited support for denormalized operands and results by supporting gross underflow for certain instructions. FGU never pre-normalizes denormalized sources, regardless of the instruction.

SPARC V9 FPop instructions generate an unfinished_FPop trap if either operand is denormalized, or if the unrounded result is denormalized, unless

- FSR.ns=1 and GSR.im=0, or
- the instruction is FABS(s,d), FCMP(s,d), FCMPE(s,d), FiTO(s,d), FMOV(s,d), FMOV(s,d)cc, FMOV(s,d)r, FNEG(s,d), or FxTO(s,d), or
- it is a dual operand instruction where one operand is denormalized and the other operand is NaN, infinity, or zero, and the expected result is NaN, infinity, or zero, or
- the instruction is FSQRT(s,d) and the denormalized operand is negative (invalid operation exception), or
- the instruction is FDIV(s,d) and the rounded result is an overflow (for example, norm ÷ denorm can generate an overflow exception), or
- the instruction is FDIV(s,d), FMUL(s,d) or FdTOs and neither operand is denormalized, and the unrounded result is denormalized, and the rounded result is normalized, or
- the instruction is FMUL(s,d), FDIV(s,d), or FdTOs and the result is a gross underflow as defined below (the FdTOs instruction with a denormalized operand always results in $E_{ur} \leq E_{guf}$, but may not result in gross underflow due to either the sign of the result or round_mode). These instructions handle denormalized unrounded results if the expected rounded result is zero, and not denormalized. This case is defined as gross underflow, and always produces inexact and underflow (satisfying both the tiny and loss of accuracy requirements) conditions, setting FSR.ufc and/or FSR.nxc depending on TEM. FMUL(s,d), FDIV(s,d), and FdTOs never produce a denormalized final result. Gross underflow always delivers a signed zero result.

gross underflow IF

($E_{ur} \leq E_{guf}$)

AND ~(S_r =0 AND round_mode=+∞) /* ensure fraction is not incremented due to rounding */

AND ~(S_r =1 AND round_mode=-∞) /* ensure fraction is not incremented due to rounding */

Where:

- E_{ur} = unnormalized and unrounded biased exponent result

FDIV(s,d): E_{ur}=Ea-Eb+bias-1

FMUL(s,d): E_{ur}=Ea+Eb-bias

FdTOs: E_{ur}=Eb-896

- E_{guf} = gross underflow biased exponent (SP=-25, DP=-54)
- S_r = sign of result
- bias=127 if SP, bias=1023 if DP
- For purposes of detecting gross underflow, if a source is denormalized then the appropriate exponent (Ea and/or Eb) is treated as zero, not Emin.
- For FMUL(s,d), an unnormalized intermediate mantissa result in the format 1X.XX has no effect on gross underflow detection.

7.10.4.4 IEEE Exceptions Handling

TABLE 7-15 IEEE Exception Case

Instruction	IEEE exceptions and OpenSPARC T2 FPX/FPD generated result (FSR.ns=0 OR GSR.im=1) Note: FSR Trap Enable Mask (TEM) is a don't care unless specified otherwise				
	invalid	**divide by zero**	**overflow**	**underflow or denormalized**	**inexact**
FABS(s,d)	cannot generate IEEE exceptions				
FADD(s,d ***FSUB(s,d))***	SNaN ∞ - ∞ result=NaN*,\ FSR.nvc=1 unfinished=0		result=±max or ±∞ FSR.ofc=1d unfinished=0	result=±0 FSR.ufc=unch unfinished=1 if unrounded denorm	result=IEEE\ FSR.nxc=1D unfinished=0
FCMP(s,d)	SNaN result=fcc FSR.nvc=1 unfinished=0				
FCMPE(s,d)	NaN result=fcc FSR.nvc=1 unfinished=0				
FDIV(s,d)	SNaN 0 ÷ 0 ∞ ÷ ∞ result=NaN FSR.nvc=1 unfinished=0	x ÷ 0, for x ≠ 0 or ∞ or NaN result=±∞ FSR.dzc=1 unfinished=0	result=±max or ±∞ FSR.ofc = 1* unfinished=0	gross uf: result=±0 FSR.ufc=1\ unfinished=0 rounded denorm (not gross uf): result=±0 FSR.ufc=unch unfinished=1 rounded norm (not gross uf): result=±min FSR.ufc=1\ unfinished=0	result=IEEE FSR.nxc=1d unfinished=0
FiTOs ***FxTO(s,d)***					result=IEEE FSR.nxc=1 unfinished=0
FiTOd	cannot generate IEEE exceptions				
FMOV(s,d)	cannot generate IEEE exceptions				
FMOV(s,d)cc	cannot generate IEEE exceptions				

TABLE 7-15 IEEE Exception Case *(Continued)*

FMOV(s,d)r	cannot generate IEEE exceptions				
FMUL(s,d)	SNaN ∞ × 0 result=NaN[*] FSR.nvc=1 unfinished=0		result=±max or ±∞ FSR.ofc=1[\] unfinished=0	gross uf: result=±0 FSR.ufc=1 unfinished=0 rounded denorm (not gross uf): result=±0 FSR.ufc=unch unfinished=1 rounded norm (not gross uf): result=±min FSR.ufc=1 unfinished=0	result=IEEE[d] FSR.nxc=1[\] unfinished=0
FNEG(s,d)	cannot generate IEEE exceptions				
FsMULd	SNaN ∞ × 0 result=NaN [*,\] FSR.nvc=1 unfinished=0				
FSQRT(s,d)	SNaN < 0, nor including -0 -∞ result=NaN[*,\] FSR.nvc=1 unfinished=0				result=IEEE[d] FSR.nxc=1 unfinished=0

TABLE 7-15 IEEE Exception Case *(Continued)*

F(s,d)TOi ***F(s,d)TOx***	NaN ∞ large result=max ±integer** FSR.nvc=1 unfinished=0				result=IEEE[d] FSR.nxc=1 unfinished=0
FsTOd	SNaN result=NaN FSR.nvc=1 unfinished=0				
FdTOs	SNaN result=NaN\ FSR.nvc=1 unfinished=0		result=±max or ±∞ FSR.ofc=1 unfinished=0	gross uf: result=±0 FSR.ufc=1[d] unfinished=0 rounded denorm (not gross uf): result=±0 FSR.ufc=unch unfinished=1 rounded norm (not gross uf): result=±min FSR.ufc=1[d] unfinished=0	result=IEEE\ FSR.nxc=1[D] unfinished=0**

* Default response QNaN = $7ff...fff_{16}$

\ SNaN input propagated and transformed to QNaN result

d Clear FSR.ofc (FSR.ufc) if overflow (underflow) exception traps and FSR.OFM (FSR.UFM) is not set and FSR.NXM is set. Set TSR.nxc

\ Rounded or Overflow (underflow) result

D Clear FSR.nxc if an overflow (underflow) exception does trap because FSR.OFM (FSR.UFM) is set, regardless of whether FSR.NXM is set. Set FSR.ofc (FSR.ufc)

** Maximum signed integer ($7ff...fff_{16}$ or $800...000_{16}$)

7.10.4.5 Unfinished_FPop Handling

TABLE 7-16 Unfinished_FPop Trap Cases

Instruction	*unfinished_FPop trap conditions and OpenSPARC T2 FPX/FPD generated result (FSR.ns=0 OR GSR.im=1) Note: FSR Trap Enable Mask (TEM) is a don't care unless specified otherwise*				
	denormalized operands (one or both)*	***invalid result***	***overflow result***	***underflow or denormalized result***	***inexact result***
FABS(s,d)	cannot generate unfinished_FPop trap				
FADD(s,d) ***FSUB(s,d)***	result=±0 FSR.ufc=unch unfinished=1			result=±0 FSR.ufc=unch unfinished=1 if unrounded denorm	
FCMP(s,d)	cannot generate unfinished_FPop trap				
FCMPE(s,d)	cannot generate unfinished_FPop trap				
FDIV(s,d)	result=±0 gross uf: FSR.ufc=1 unfinished=0 not gross uf and not overflow: FSR.ufc=unch unfinished=1			result=±0 gross uf: FSR.ufc=1 unfinished=0 not gross uf: FSR.ufc=unch unfinished=1	
FiTOs ***FxTO(s,d)***	cannot generate unfinished_FPop trap				
FiTOd	cannot generate unfinished_FPop trap				
FMOV(s,d)	cannot generate unfinished_FPop trap				
FMOV(s,d)cc	cannot generate unfinished_FPop trap				
FMOV(s,d)r	cannot generate unfinished_FPop trap				
FMUL(s,d)	result=±0 gross uf: FSR.ufc=1 unfinished=0 not gross uf: FSR.ufc=unch unfinished=1			result=±0 gross uf: FSR.ufc=1 unfinished=0 not gross uf: FSR.ufc=unch unfinished=1	
FNEG(s,d)	cannot generate unfinished_FPop trap				
FsMULd	result=±0 FSR.ufc=unch unfinished=1				

TABLE 7-16 Unfinished_FPop Trap Cases *(Continued)*

FSQRT(s,d)	result=±0 FSR.ufc=unch unfinished=1 (for +denorm operand only)				
F(s,d)TOi ***F(s,d)TOx***	result=±0 FSR.ufc=unch unfinished=1				
FsTOd	result=±0 FSR.ufc=unch unfinished=1				
FdTOs	result=±0 gross uf: FSR.ufc=1 unfinished=0 not gross uf: FSR.ufc=unch unfinished=1			result=±0 gross uf: FSR.ufc=1 unfinished=0 not gross uf: FSR.ufc=unch unfinished=1	

* dual operand instructions do not generate an unfinished '_FPop trap if one operand is denormalized, the other operand is NaN, infinity, or zero, and the expected result is NaN, infonity, or zero.

For exaample:

denorm + QNaN = QNaN

denorm + SNaN = QNaN (with invalid exception)

denorm + ∞ = ∞

denorm × 0 = 0

denorm ×∞ = ∞

CHAPTER 8

Trap Logic Unit

The Trap Logic Unit (TLU) manages exceptions, trap requests, and traps for the SPARC core. Exceptions and trap requests are conditions that may cause a thread to take a trap. A trap is a vectored transfer of control to supervisor software through a trap table (from the SPARC Version 9 Architecture). The TLU maintains processor state related to traps as well as the Program Counter (PC) and Next Program Counter (NPC).

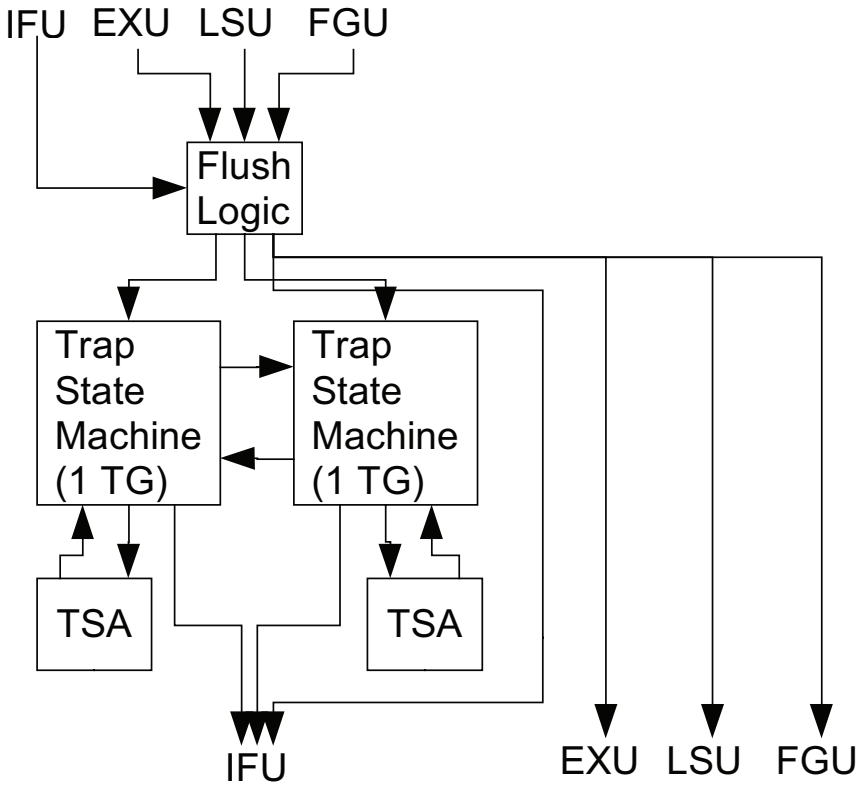
In the event of an exception or trap request, the TLU prevents the update of architectural state for the instruction or instructions after an exception. In many cases, the TLU relies on the execution units and the IFU to assist with the preservation of architectural state.

The TLU preserves the PC and NPC for the instruction with the exception. In some cases, the TLU must create a precise interrupt point for exceptions and interrupt requests not directly related to the instruction stream. In all cases, the TLU maintains the Trap Stack.

8.1 Overview

The TLU supports several logical functions.

FIGURE 8-1 TLU Block Diagram



- The Flush Logic generates flushes in response to exceptions to create precise interrupt points (when possible).
- The Trap Stack Array (TSA) maintains trap state for the eight threads for up to six trap levels per thread.
- The Trap State Machine holds and prioritizes trap requests for the eight threads in two thread groups.

8.2 Architectural Concerns

The TLU supports precise, disrupting, and reset trap categories. While the TLU does support the store_error trap, which is documented as being deferred, its implementation is the same as a disrupting trap.

8.2.1 Precise Traps

A precise trap is caused by a specific instruction. When a precise trap occurs, processor state reflects that all previous instructions have executed and completed, and the excepting instruction and subsequent instructions have not executed.

8.2.2 Disrupting Traps

A disrupting trap is caused by a condition, not an instruction. Once a disrupting trap has been serviced, the program may pick up where it left off. The condition that causes a disrupting trap may or may not be associated with a specific instruction. In some cases, the condition may be or may lead to corruption of state, and therefore a disrupting trap may degenerate into a reset trap.

For example, a trap request from an I/O device is a disrupting trap. In this case, the trap would service the I/O device and then return control to the point at which the trap was taken. However, an uncorrectable ECC error also results in a disrupting trap. In this case, the trap handler may determine that corruption has occurred, and may cause a reset trap.

8.2.3 Reset Traps

A reset trap occurs when hardware or software determines that the hardware must be reset to a known state. Once a reset trap has been serviced, the program does not resume.

On OpenSPARC T2, a POR reset can only occur after a power-on. All other reset traps can only be taken if the thread can make forward progress. A reset trap will not resolve a deadlock.

8.2.4 Deferred Traps

The only deferred trap on OpenSPARC T2 is the store_error trap, and it is implemented as though it were a deferred trap.

8.3 Flushes

The TLU receives exception reports and trap requests from trapping instructions, hardware monitors, steering registers, and the crossbar. When the TLU receives an exception or trap request, it must first flush the relevant thread from the machine, to ensure that the trap handler can proceed without corruption from the thread itself.

Only instructions from the trapping thread are flushed. Instructions for other threads continue executing or remain in instruction buffers. All flushes initiated by the IFU in this section affect the IFU pipe stages as well as the stages shown here.

8.3.1 Excepting Instructions

OpenSPARC T2 has several types of instructions that cause exceptions.

8.3.1.1 Execution Unit and Load Store Unit Exceptions

The EXU generates exceptions for several different instructions and conditions:

Trap on condition code

Out of Range Virtual Addresses (Section 8.3.1.6, "Out of Range Virtual Addresses" on page 8-10)

ECC errors on source operands (Section 8.3.1.8, "Integer Instructions with ECC Errors" on page 8-11)

The LSU generates exceptions for several different instructions and conditions:

Alignment errors (*mem_address_not_aligned)

Data access error (data_access_error)

The EXU and LSU signal exceptions to the TLU in the B stage. (Some exceptions are reported in the M stage, but TLU internally pipes these to the B stage).The TLU determines if the subsequent instruction is from the same thread. If it is, the TLU sends a flush to the Execution Units, the Load Store Unit, and the Floating-point and Graphics Unit in the W stage of the excepting instruction (which is the B or FX2 stage of the subsequent instruction). The TLU also informs the IFU in this cycle that

the processor must flush the relevant thread. The IFU detects and flushes any later instructions from this thread that may be in the machine. TABLE 8-1 shows the flush sequence; shading represents instruction flushes.

TABLE 8-1 Flush Due To Execution Unit or Load Store Unit Exception

D / IRF	ELOp0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5	GenericOp6 Flushed by IFU
E / FRF		ELOp0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5 Flushed by IFU
M / D$ / FX1			ELOp0	GenericOp1	GenericOp2	GenericOp3	GenericOp4 Flushed by IFU
B / FX2				ELOp0 EXU, LSU report exception to TLU	GenericOp1	GenericOp2	GenericOp3 Flushed by IFU
W / FX3					EXU, LSU flush ELOp0 TLU broadcasts flush	GenericOp1 Flushed by TLU	GenericOp2 Flushed by IFU

8.3.1.2 Floating-point and Graphics Exceptions

The FGU predicts floating-point and graphics exceptions in stage FX1, reports the exception prediction status in FX2 (which corresponds to the integer pipe stage B), and reports exceptions to the TLU in stage FB. (The FGU does not predict exceptions on integer or floating-point divides, since they are long latency and therefore have no exception hazard.) The TLU determines if the subsequent instruction is from the same thread. If it is, the TLU sends a flush to the Execution Units, the Load Store Unit, and the Floating-point and Graphics Unit in the FX3 stage of the excepting instruction (which is the B or FX2 stage of the subsequent instruction). The TLU also informs the IFU in this cycle that the processor must flush the relevant thread. The IFU detects and flushes any later instructions from this thread that may be in the machine. The TLU sends the PC and NPC of the instruction after the FGU excepting instruction to the IFU to minimize the mispredict penalty (in the case where an exception did not occur). If the FGU prediction is correct, the TLU reports a flush to the IFU in the FW cycle of the excepting FGU instruction, and the IFU flushes the refetch of the subsequent instructions, and starts fetching the PC of the FGU exception trap handler.

If the FGU prediction is incorrect, the TLU does not flush again and does not send

TABLE 8-2 Flush of the Floating-point and Graphics Unit Due To FGU Exception

P	Op2	Op3	Op4	Op5	Op6 Flushed by IFU	Op7 Flushed by IFU		
D	Op1	Op2	Op3	Op4	Op5	Op6 Flushed by IFU		
FRF / E	FGUOp0	Op1	Op2	Op3	Op4	Op5 Flushed by IFU		
FX1 / M		FGUOp0	Op1	Op2	Op3	Op4 Flushed by IFU		
FX2 / B			FGUOp0 FGU reports exception prediction	Op1	Op2	Op3 Flushed by IFU		
FX3 / W				FGUOp0 TLU broadcasts flush	Op1 Flushed by TLU	Op2 Flushed by IFU		
FX4					FGUOp0			
FX5						FGUOp0		
FB							FGUOp0 FGU reports exception to TLU	
FW								FGU flushes FGUOp0 TLU broadcasts flush

any other PC or NPC to the IFU.

TABLE 8-3 FGU Exception Mispredict

P	Op2	Op3	Op4	Op5	Op6 Flushed by IFU	Op7 Flushed by IFU				Op1 (refetched)
D	Op1	Op2	Op3	Op4	Op5	Op6 Flushed by IFU				
FRF / E	FGUOp0	Op1	Op2	Op3	Op4	Op5 Flushed by IFU				
FX1 / M		FGUOp0	Op1	Op2	Op3	Op4 Flushed by IFU				
FX2 / B			FGUOp0 FGU reports exception prediction	Op1	Op2	Op3 Flushed by IFU				
FX3 / W				FGUOp0 TLU broadcasts flush	Op1 Flushed by TLU	Op2 Flushed by IFU				
FX4					FGUOp0					
FX5						FGUOp0				
FB							FGUOp0 (no exception)			
FW								FGUOp0 (no flush)		

8.3.1.3 Illegal Instructions

The IFU detects Illegal instructions before loading them into the cache or before bypassing the cache. The IFU issues the illegal instruction to the TLU. The TLU determines if the subsequent instruction is from the same thread. If it is, the TLU sends a flush to the Execution Units, Load Store Unit,and Floating-point and Graphics Unit in the W or FX3 stage of the illegal instruction (which is the B or FX2 stage of the subsequent instruction). The TLU also informs the IFU in this cycle that the processor must flush the relevant thread. The timing is the same as EXU exceptions; see TABLE 8-1for more detail.

8.3.1.4 Invalid Instructions

Invalid instructions differ from illegal instructions in that invalid instructions are implemented instructions with correct opcodes but have some unsupported or incorrect field or fields. For example, a load with an unimplemented ASI value is an invalid instruction.

Some instructions have many invalid forms. Some invalid forms are detected at decode and are handled identically to illegal instructions. The execution units (EXU, FGU, LSU) detect the other invalid forms. Execution units may take two approaches to detecting invalid instructions:

- The unit may perform a decode of instruction fields to determine invalid forms. In this case, the unit reports an invalid form exception to the TLU in the B stage; the TLU then flushes the subsequent instruction from the relevant thread and reports the flush to the IFU as with any other EXU exception; see TABLE 8-1 for more detail.
- The unit may attempt execution of the instruction and deduce that the instruction was an invalid form from the result of the execution attempt. For example, invalid addresses for certain ASIs may be detected through forwarding the access over the ASI bus to the target unit. These instructions are long latency instructions, and there are not any subsequent instructions for the relevant thread below pick (when the exception is reported). The execution unit (the LSU in this example) flushes the invalid instruction and reports it to the TLU. The TLU signals flush to the IFU to clear the instruction buffers for the relevant thread.

8.3.1.5 Translation Exceptions

Translation exceptions occur when a virtual or real address (VA or RA) cannot be translated to a physical address (PA) (MMU miss), or when the permissions in the Translation Table Entry (TTE) do not permit the requested access (access exception).

MMU Miss

A MMU miss occurs when L1 TLB cannot find a TTE that matches the virtual page number (VPN) and context of a request and either hardware tablewalk is disabled or hardware tablewalk is also unable to find a matching TTE.

Instruction Access MMU Miss

The IFU accesses the L1 ITLB in parallel with the cache and does not issue the instructions fetched during a translation miss. The IFU inserts an ITLB miss nop into the pipe. This nop serves two purposes. First, it prevents an issue deadlock case if the instruction immediately before the miss is a branch, since a branch cannot

issue until the instruction in its delay slot is available. Second, it ensures that the nop and the miss are non-speculative. Since the fetch unit within the IFU knows this is a miss, it moves the thread to the fetch wait state and does not fetch any more instructions for the relevant thread. The IFU issues the nop to the TLU.

If hardware tablewalk is disabled, the TLU creates a fast_instruction_access_MMU_miss trap and redirects the IFU to the fast_instruction_access_MMU_miss trap vector.

If hardware tablewalk is enabled, the TLU forwards the reload request to the MMU. The MMU accesses the Translation Storage Buffer (TSB) to try to find a matching TTE.

If the MMU does not find a matching TTE, it informs the TLU. The TLU creates an instruction_access_MMU_miss trap and redirects the IFU to the instruction_access_MMU_miss trap vector.

If the MMU finds a matching TTE, the ITLB loads the TTE and no trap is generated.
Data Access MMU Miss

The LSU accesses the L1 DTLB in parallel with the cache and does not execute a load or a store with a TLB miss. The DTLB passes a miss exception to the TLU. The TLU flushes subseqent instructions in the thread in response to the DTLB miss. If hardware tablewalk is enabled, the TLU forwards a DTLB reload request to the MMU in the W stage if it is not flushed; the miss is non-speculative at this point. If the MMU is unable to find a matching TTE, the MMU reports the miss to the TLU. The TLU takes a data_access_MMU_miss and redirects the IFU to the data_access_MMU_miss trap trap vector.

If hardware tablewalk is disabled, the TLU takes a fast_data_access_MMU_miss and redirects the IFU to the fast_data_access_MMU_miss trap trap vector.

If the MMU finds a matching TTE, the DTLB loads the TTE and no trap is generated.

Access Exception

Access exceptions occur when either L1 TLB or the hardware tablewalk finds a TTE with matching VA and context that does not permit the requested access. This includes accessing a privileged address while in user mode, accessing a hypervisor privileged address while not in hypervisor mode, attempting to execute an address that does not permit execution, or attempting to write to an address without write permission.

Instruction Access Exception From ITLB

The ITLB detects access exceptions in parallel with cache access. Instructions fetched with access exceptions are discarded. If the ITLB detects an access exception, the IFU inserts a instruction access exception nop into the pipe . This nop serves two purposes. First, it prevents an issue deadlock case if the instruction immediately before the access exception is a branch, since a branch cannot be picked until the instruction in its delay slot is available. Second, it ensures that the access exception is non-speculative. Since the fetch unit in the IFU knows this is an access exception, it moves the thread to the fetch wait state and does not fetch any more instructions for the relevant thread. The IFU passes the nop down the pipe to the TLU. The TLU waits until the nop reaches the W stage before flushing the thread and taking the instruction_access_exception trap.

Instruction Access Exception From Hardware Tablewalk

If hardware tablewalk is enabled and it detects an access exception on a reload request from the ITLB, the MMU signals the exception to the TLU. The TLU flushes the pipe and takes an instruction_access_exception trap.

Data Access Exception from DTLB

The LSU accesses the DTLB in parallel with the cache and does not execute a load or a store with an exception violation. If it detects an access exception, the LSU signals a LSU synchronization to the IFU to flush all subsequent instructions for the relevant thread. The DTLB signals the access violation to the TLU in the B stage. The TLU flushes the thread and takes a data_access_exception trap.

Data Access Exception from Hardware Tablewalk

If hardware tablewalk is enabled and it detects an access exception on a reload request from the DTLB, the MMU signals the exception to the TLU. The TLU flushes the pipe and takes a data_access_exception trap.

8.3.1.6 Out of Range Virtual Addresses

Since OpenSPARC T2 does not support 64 bit virtual addresses (VAs), the hardware must check that 64 bit quantities that are used as VAs are in one of two valid ranges. Hardware implements a 48 bit VA. The upper 17 bits of any 64 bit value must be equal to all zeroes or all ones for it to be a valid VA range in OpenSPARC T2. The TLU detects out of range VAs for the instruction PC in the E stage. The EXU detects

out of range VAs for branch targets in the E stage. Timing for the flush and trap for out of range VAs is the same as for other EXU exceptions. See TABLE 8-1 for more detail.

The IFU detects out of range VAs in the case instruction fetch reaches the cache line immediately before the VA hole. The IFU creates an out of range VA nop in this case. This nop behaves much like the ITLB miss nop.

The TLU tracks out of range VAs written to the trap stack. The TLU detects an out of range VA exception if a done or retry to an out of range VA exception occurs.

8.3.1.7 Out of Range Real Addresses

OpenSPARC T2 hardware detects Real Addresses (RAs) that do not have the upper 17 bits of the 64 bit address equal to all zeroes or all ones with the same mechanisms used for out of range VA detection. OpenSPARC T2 supports a 40 bit RA, not a 48 bit RA. However, to use RAs, hypervisor software must program a RA to PA Translation Table Entry (TTE) into the Translation Lookaside Buffer (TLB). So, hypervisor software must manage bits 47 to 39 of the RA in the TTE to ensure that no RAs in the RA hole are used.

8.3.1.8 Integer Instructions with ECC Errors

Any instruction that reads the Integer Register File (IRF) can cause a flush due to an ECC error. Hardware does not correct IRF ECC errors. IRF ECC errors are treated (from a flush and trap perspective) the same as any other EXU exception.

8.3.1.9 Floating-point and Graphics Instructions with ECC Errors

Any instruction that reads the Floating-point Register File (FRF) can cause a flush due to an ECC error. The FGU reports a predicted exception in the FX2 stage and reports the ECC error to the TLU in the FB stage. The TLU flushes the subsequent instruction if it is from the same thread, and the IFU flushes later instructions from the same thread. Hardware does not correct FRF ECC errors. FRF ECC errors are treated (from a flush and trap perspective) the same as any other FGU exception.

8.3.1.10 Load Misses with L2 ECC Errors

If a load miss has an L2 ECC error, the L2 reports that error back to the LSU. The LSU reports the ECC error to the TLU. If the ECC error is uncorrectable, then the TLU generates a precise trap. The TLU reports the trap to the IFU, so that the IFU can flush the relevant thread, fetch the trap vector, and transition the thread to the ready state.

If the ECC error was corrected, then the TLU generates a disrupting trap request, and the LSU signals complete to the IFU for the relevant thread.

8.3.1.11 Stores with L2 ECC Errors

The LSU forwards all stores to the L2 cache. Unlike load misses, stores do not cause the thread to wait for completion. Consequently, a store with an uncorrectable L2 ECC error cannot cause a precise trap. The TLU generates a disrupting trap for the ECC error.

8.3.1.12 Instruction Cache Misses with L2 ECC Errors

If an instruction cache miss has an L2 ECC error, the L2 reports that error back to the IFU. If the ECC error was not corrected , the IFU tags the instruction with a L2 ECC exception code (which behaves similarly to the ITLB miss nop) and the TLU generates a precise (for uncorrectable and NotData) or disrupting (for correctable) trap if it reaches the W stage.

8.3.1.13 DONE and RETRY

DONE and RETRY behave similarly to any other EXU exception. The IFU issues DONE and RETRY to the TLU just as it does for the various nops defined for the exceptions that occur in the pipe stages above the EXU. The only difference is that RETRY uses the PC and NPC from the trap stack as the new PC and NPC, and DONE uses the NPC from the trap stack as the new PC.

8.3.1.14 SIR

SIR behaves similarly to any other EXU exception. Since SIR results from an SIR instruction, an SIR trap request is precise.

8.3.2 Trap Requests from Crossbar

Trap requests from the crossbar take three forms:

Software trap requests from other cores

Hardware errors from outside the core

External Interrupt Reset (a.k.a. XIR)

Software must be able to return to the program after the trap, but the relationship of the trap to the instruction stream is not important.

8.3.3 Power On Reset, Warm Reset, DeBug Reset

All of these resets are signalled to the TLU via a zero-to-one transition of the Core Running register. TLU behavior is identical for all three of these resets.

8.4 Traps

The TLU directs the IFU to fetch the correct trap vector (based on exception) at the right time (based on state of the thread).

The TLU has two trap interfaces with the IFU, one per thread group. The TLU multiplexes the trap requests within a thread group to the IFU, favoring longest pipe exceptions.

8.4.1 Precise Traps

The TLU ensures that the thread has completed all instructions prior to and no instruction subsequent to a precise trap exception, so that the trap handler accesses the correct architectural state. An instruction completes if it reaches the W or FW stage and is not flushed by the TLU or IFU. The TLU signals the trap to the IFU at the earliest three cycles after the W or FW stage, assuming no other thread in the thread group has a higher priority trap.

TABLE 8-4 EXU or LSU Exception Trap, Single Thread

BF	Op0	Op1											Trap Vector for IntOp0
F		Op0	Op1										
C			Op0	Op1									
P				Op0	Op1								
D / IRF					Op0	Op1							
E / FRF						Op0	Op1						
M / FX1							Op0	Op1					
B / FX2								Op0 EXU, LSU exception reported to TLU	Op1				
W / FX3									Op0 TLU broadcasts flush	Op1 Flushed by TLU			
T												TLU sends trap vector to IFU	

RETRY and DONE behave as precise traps. For DONE, the TLU forwards the NPC from the trap stack as the PC to the IFU.

For RETRY, the TLU forwards the PC from the trap stack to the IFU. In the event that the TLU believes that the NPC resulting from the RETRY is not sequential to the PC resulting from the retry, the TLU directs the IFU to fetch a single instruction and allow it to execute. Once TLU sees that the single instruction executes with no exception, it will redirect IFU to the NPC.

TABLE 8-5 FGU Exception Trap, Single Thread

BF	FGU Op0	Op1											Op1				Trap Vector for FGU Op0
F		FGU Op0	Op1											Op1			
C			FGU Op0	Op1											Op1		
P				FGU Op0	Op1												
D / IRF					FGU Op0	Op1											
E / FRF						FGU Op0	Op1										
M / FX1							FGU Op0	Op1									
B / FX2								FGU Op0 FGU reports except. pred.	Op1								
W / FX3									FGU Op0 TLU bcasts flush	Op1							
FX4										FGU Op0							
FX5											FGU Op0						

TABLE 8-5 FGU Exception Trap, Single Thread *(Continued)*

FB												FGUOp0 FGU reports except. to TLU					
FW													FGU Op0 TLU b'casts flush				
T												TLU sends Op1 PC to IFU				TLU sends trap vector to IFU	

8.4.2 Disrupting Traps

The TLU services hardware exceptions, and XIR requests with disrupting traps. The TLU uses the PC and NPC of the next instruction that exits the M / FX1 stage for the relevant thread as the PC and NPC to put on the trap stack. The TLU flushes this instruction from the machine. The following table shows the timing for an exception, but all exceptions that create disrupting traps have the same timing.

TABLE 8-6 Disrupting Trap (due to an exception)

Except Request		Exception T0										
BF	T0 Op0	T0 Op1										T0 Trap Vector for exception
F		T0 Op0	T0 Op1									
C			T0 Op0	T0 Op1								
P				T0 Op0	T0 Op1							
D / IRF					T0 Op0	T0 Op1						
E / FRF						T0 Op0	T0 Op1					
M / FX1							T0 Op0	T0 Op1				
B / FX2								T0 Op0	T0 Op1 TLU broadcasts flush			
W / FX3									T0 Op0 TLU steals PC & NPC for exception trap	T0 Op1		
FX4												
FX5												

TABLE 8-6 Disrupting Trap (due to an exception) *(Continued)*

FB													
FW													
T												TLU sends exception trap vector to IFU	

8.4.3 POR, WMR, DBR Traps

POR can only occur at power-on, when the thread is idle. Upon notification of a reset trap request, the TLU updates the trap stack with PC and NPC of all zeroes. The TLU sends the trap vector to the IFU.

8.4.4 Priority of Thread Traps Within A Thread Group

Since only one instruction per thread can enter the W stage per cycle, and since each exception flushes later instructions, each thread can only generate one trap at a time. However, since the threads within a thread group share a trap interface to IFU, only one thread per thread group can trap per cycle. The TLU prioritizes trap requests for the threads within a thread group as follows:

1. Reset trap requests
2. Disrupting trap requests
3. Exceptions on divides
4. Exceptions on load misses and long latency instructions
5. Exceptions on normal pipe FGU instructions
6. Exceptions on normal pipe EXU and LSU instructions
7. Microarchitectural redirects and ITLB reloads

Within a trap request priority level, the TLU uses a static priority from thread 0 to thread 3 to select which request to service.

The following tables document some of the possible concurrent exceptions between threads and the priority of exceptions to take traps within a thread group.

TABLE 8-7 Traps with Concurrent LSU and EXU Exceptions in Different Threads in Same Thread Group

BF	T1 Ex										Trap Vector for T0 Ld	Trap Vector for T1 Ex
F		T1 Ex										Trap Vector for T0 Ld
C			T1 Ex									
P				T1 Ex								
D / IRF					T1 Ex							
E / FRF						T1 Ex						
M / FX1							T1 Ex					
B / FX2								T1 Ex EXU reports exception to TLU				
G	T0 Ld	T0 Ld	T0 Ld	T0 Ld	T0 Ld	T0 Ld	T0 Ld	T0 Ld LSU reports exception to TLU				
W / FX3									T0 Ld TLU broadcasts flush T1 Ex TLU broadcasts flush			
T										TLU sends T0 trap vector to IFU	TLU sends T1 trap vector to IFU	

TABLE 8-8 Traps with Concurrent FGU and EXU Exceptions on Different Threads in Same Thread Group

BF	T0 FP				T1 Ex									Trap Vector for T0 FP	Trap Vector for T1 Ex
F		T0 FP				T1 Ex									Trap Vector for T0 FP
C			T0 FP				T1 Ex								
P				T0 FP				T1 Ex							
D / IRF					T0 FP				T1 Ex						
E / FRF						T0 FP				T1 Ex					
M /FX1							T0 FP				T1 Ex				
B / FX2								T0 FP exc pre d				T1 Ex EXU reports exception to TLU			
W /FX3									T0 FP				T1 Ex TLU broadcasts flush		
FX4										T0 FP					
FX5											T0 FP				
FB												T0 FP FGU reports exception to TLU			
FW													T0 FP TLU broadcasts flush		
T														TLU sends T0 trap vector to IFU	TLU sends T1 trap vector to IFU

TABLE 8-9 Traps with Concurrent FGU, Divide, LSU, and EXU Exceptions on Different Threads in Same Thread Group

BF	T0 FP				T1 Ex									T3 Dv Tr Vector	T2 Ld Tr Vector	T0 FP Tr Vector	T1 Ex Tr Vector
F		T0 FP				T1 Ex									T3 Dv Tr Vector	T2 Ld Tr Vector	T0 FP Tr Vector
C			T0 FP				T1 Ex									T3 Dv Tr Vector	T2 Ld Tr Vector
P				T0 FP				T1 Ex									T3 Dv Tr Vector
D / IRF					T0 FP				T1 Ex								
E / FRF						T0 FP				T1 Ex							
M /FX1							T0 FP				T1 Ex						
B / FX2								T0 FP exc pre d				T1 Ex EXU except					
G	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld	T2 Ld LSU except					
W /FX3									T0 FP				T1 Ex bcast flush T2 Ld bcass flush				
FX4										T0 FP							
FX5											T0 FP						
FD	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div	T3 Div						

TABLE 8-9 Traps with Concurrent FGU, Divide, LSU, and EXU Exceptions on Different Threads in Same Thread Group *(Continued)*

FB												T0 FP FGU except T3 Div FGU except					
FW													T0 FP bcast flush T3 Div bcast flush				
T														TLU sends T3 trap to IFU	TLU sends T2 trap to IFU	TLU sends T0 trap to IFU	TLU sends T1 trap to IFU	

CHAPTER 9

Memory Management Unit

The Memory Management Unit (MMU) reads Translation Storage Buffers (TSBs) for the Translation Lookaside Buffers (TLBs) for the instruction and data caches. The MMU receives reload requests for the TLBs and uses its hardware tablewalk state machine to find valid Translation Table Entries (TTEs) for the requested access. The TLBs use the TTEs to translate Virtual Addresses (VAs) and Real Addresses (RAs) into Physical Addresses (PAs). The TLBs also use the TTEs to validate that a request has the permission to access the requested address.

The MMU maintains several sets of Alternate Space Identifier (ASI) registers associated with memory management. Software uses the scratchpad registers in handling translation misses that the hardware tablewalk cannot satisfy; the MMU maintains these registers. The MMU maintains translation error registers that provide software with the reasons why translation misses occur. Hardware tablewalk configuration registers control how the hardware tablewalk state machine accesses the TSBs. Software reads and writes the TLBs through another set of ASI registers.

The TLBs do not reside in the MMU, but they are documented here to consolidate translation documentation.

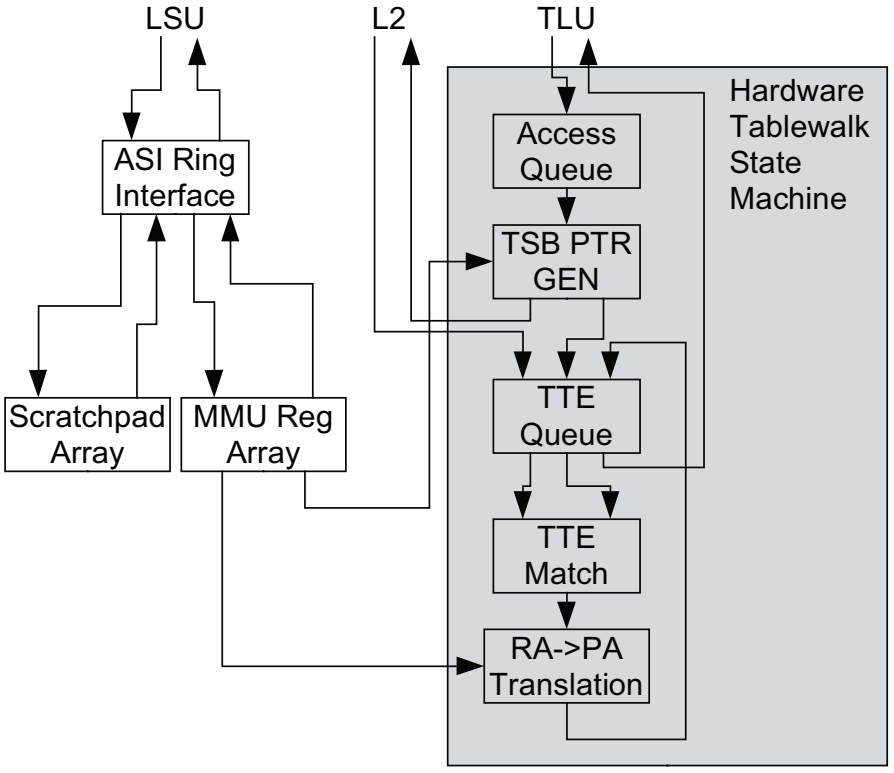
9.1 Overview

The Scratchpad Array stores the scratchpad registers.

The Hardware Tablewalk State Machine services first level TLB misses for the 8 threads.

The MMU Register Array stores the various MMU ASI registers, including the hardware tablewalk control registers.

FIGURE 9-1 MMU Block Diagram



9.2 Translation Lookaside Buffers

The Instruction and Data Translation Lookaside Buffers (TLBs) provide the first level translation for instruction and data accesses. The TLBs are accessed in parallel with the caches and the tags. The ITLB has 64 fully associative entries; the DTLB has 128 fully associative entries.

The threads share the TLBs. A translation loaded for one thread can be used by other threads, if the partition ID, Virtual Page Number (VPN), REAL bit, and context match.

9.2.1 Translation Hit

The TLB compares the partition ID, VPN, REAL bit, and context of each access with each entry of the TLB.

(The TLB hits if either of the contexts of the access matches and the other fields match. If the access is a real access, the context match is ignored.)

If any single entry matches, the TLB generates a Physical Address (PA) by concatenating the Physical Page Number (PPN) stored in the TLB with the lower portion of the virtual address. If no entries match, then the TLB signals a miss. The TLU eventually receives the miss signal and (if hardware tablewalk is enabled) forwards a reload request to the hardware tablewalk state machine within the MMU. If the MMU finds a matching TTE in the TSBs, the TLB loads the matching TTE and the access is retried. If the MMU does not find a matching TTE, the TLU directs the IFU to take the appropriate trap.

The TLB also compares the permission bits with the attempted access. If an access exception occurs, the TLB reports the exception to the TLU, which takes the appropriate trap.

The PA provided by the TLB is compared to the address from the tags to determine cache hit.

9.2.2 ITLB Reload

If hardware tablewalk is disabled or if the MMU does not find a matching TTE, the TLU creates the appropriate trap.

If the MMU finds a matching TTE, it forwards the TTE to the TLU. The TLU forwards the TTE and the PC of the nop to the IFU.

(Note that the trap state machines of the two thread groups must coordinate TLB writes; the TLBs are shared by the thread groups.The IFU writes the PC into the relevant thread's fetch PC.)

The IFU allocates the next two cycles for the ITLB to demap any matching TTEs and to write the new TTE; scheduling the TTE write puts all threads in the wait state for two cycles. The IFU retries the instruction access sometime after the ITLB write, and the ITLB hits. The thread does not trap in this case.

TABLE 9-1 ITLB Reload

BF					All threads in wait	All threads in wait	IFU can pick T0 for fetch	
F						ITLB demap All threads in wait	ITLB write All threads in wait	T0
C								
P								
D / IRF								
E / FRF								
M / FX1								
B / FX2								
W / FX3								
T			TLU arbitrates trap requests, ITLB reloads	TLU prepares TTE packet to IFU	TLU sends TTE to IFU	TLU redirects IFU to PC of miss		
MMU	ITLB_DATA_IN register written	MMU passes TTE to TLU for T0						

9.2.3 DTLB Reload

If the MMU finds a matching TTE, it signals Decode to create two cycles with no LSU instructions in decode. Decode creates the holes and the TLU forwards the TTE to the DTLB so that the TTE demaps and writes when the holes reach the M stage. The LSU instruction with the miss reaches the LSU after the write and hits in the DTLB. In parallel, MMU signals the TLU to redirect the affected thread to the PC of the instruction with the DTLB miss. The thread does not trap in this case.

TABLE 9-2 DTLB Reload

P	T1 Op0			T0 Ld0	T1 Op1				
D / IRF		T1 Op0	Hole created	Hole created	T0 Ld0	T1 Op1			
E / FRF			T1 Op0			T0 Ld0	T1 Op1		
M / FX1				T1 Op0	DTLB demap	DTLB write	T0 Ld0 Hits DTLB	T1 Op1	
B / FX2					T1 Op0			T0 Ld0	T1 Op1
W / FX3						T1 Op0			T0 Ld0
MMU	DTLB_DATA_IN register written	MMU requests LSU hole	MMU requests LSU hole	MMU sends TTE tag and data to DTLB					

Note – The request for the LSU holes only causes LSU instructions to stall at decode. EXU and FGU instructions do not stall at decode due to LSU hole request.

9.2.4 Page Sizes

OpenSPARC T2 supports four page sizes: 8 KB, 64 KB, 4 MB, and 256 MB. The TLBs can hold translations of all four sizes concurrently.

9.2.5 Multiple Contexts

OpenSPARC T2 supports multiple contexts for instruction and data access. For some applications, sharing instruction or data translations can significantly reduce the TLB miss rates. The supervisor can create TTEs that are shared by any number of processes. This mechanism allows software to opportunistically use the shared TTEs (based on the VPN of each access) so that software does not have to specify whether a particular access uses a shared TTE or a private TTE.

The TLBs compare the context in the TTEs to both contexts of the request (PRIMARY_CONTEXT_0 and PRIMARY_CONTEXT_1 or SECONDARY_CONTEXT_0 and SECONDARY_CONTEXT_1). If either context value for the request matches the context in a TTE, and the VPN, page size, and

other checks are satisfied, then the TTE is considered a valid translation. If the contexts of the request match in different TTEs, and the VPN, page size, and other checks are satisfied, the TLB signals a multiple hit error to the TLU.

9.2.6 RA to PA Translation

X-PAR requires the TSB TTEs hold Real Page Numbers (RPNs) since the supervisor controls the TTEs and does not have access to Physical Addresses. Hardware tablewalk converts the RPN specified in the TSB TTE into a PPN; the TLBs store this PPN.

Supervisors may believe that they can access physical addresses directly. The TLBs have a bit in the TTE tag that indicates that the TTE stores a real to physical translation (instead of a virtual to physical translation). The hypervisor uses this bit to virtualize direct physical address access for supervisors.

The X-PAR architecture views ASI_REAL as circumventing the context match, so the TLB ignores the context match if the access has ASI_REAL. Note that the partition ID is always part of the match determination, so two partitions cannot share a TTE, regardless of whether it is a RA to PA TTE or a VA to PA TTE.

Any TTE in the TLB with the REAL bit set to 1 has a RA, not a VA. When the TLB encounters an access with an ASI indicating real addressing, it checks for an entry with a matching real address and a REAL bit equal to 1. If it does not find a matching entry, the TLB misses.

The TLU signals a trap to the hypervisor when the miss becomes non-speculative (the miss reaches the W stage). The hypervisor loads the translation directly into the TLB with the REAL bit set to 1. The hypervisor then issues RETRY.

The TLBs cache RA to PA TTEs just like VA to PA TTEs. RA to PA TTEs have the REAL bit set to 1, so they do not match for accesses with virtual addresses.

9.2.7 Demap

TLB demap provides software a mechanism to remove TTEs cached in the TLBs. It also permits hardware to prevent multiple TTE hits.

Hardware initiates Demap Page on any write to the TLB (using the VA and context of the TTE being written). This prevents multiple TTE matches in the TLB.

Software initiates Demap Page, Demap Context, Demap Real, and Demap All through a write to an ASI register. The ITLB and DTLB have separate ASI registers to control demap.

9.2.7.1 Demap Page

The Demap Page operation invalidates all TTEs that match the specified partition ID, context, VPN, and REAL bit. If the demap real bit is zero, Demap Page invalidates VA to PA TTEs. If the demap real bit is one, Demap Page invalidates RA to PA TTEs, and it ignores the context match.

9.2.7.2 Demap Context

The Demap Context operation invalidates all TTEs that match the specified partition ID and context and with the REAL bit of zero. Demap Context does not invalidate any TTE with the REAL bit of one.

9.2.7.3 Demap Real

The Demap Real operation invalidates all TTEs that match the specified partition ID and have their REAL bits set. Demap Real only invalidates RA to PA TTEs.

9.2.7.4 Demap All

The Demap All operation invalidates all TTEs that match the specified partition ID, regardless of VPN, RPN, REAL bit, or context.

9.2.8 Replacement Algorithm

The TLB has a Used bit replacement algorithm. When an entry matches or is written, the TLB sets the U bit of that entry. When all U bits in the TLB are set, the TLB resets all the U bits the next cycle. Subsequent matches set U bits of used TTEs.

The TLB finds the first entry that does not have neither the U nor the V bits set. When the TLB writes, it replaces this first entry with the TTE being written.

The Used bit algorithm performs better than round robin replacement, but not as well as pseudo-LRU (partitioned LRU).

9.3 Hardware Tablewalk

Hardware Tablewalk (HW TW) services reload requests from the TLBs. It accesses the TSBs to find TTEs that match the VA and one of the contexts of the request. Hardware Tablewalk can access up to four TSBs for each request.

Hardware Tablewalk provides a RPN to PPN translation mechanism. The supervisor controls the TTE, but the supervisor cannot access or control physical memory, so its TTEs have RPNs, not PPNs. The hypervisor programs the RPN to PPN translation within HW TW to permit HW TW to load supervisor-controlled TTEs into the TLBs that can translate VAs into PAs.

Hardware Tablewalk does not translate requests with RAs. In the event that a Real Address misses the TLB, the TLU signals a Real_Address_MMU_Miss trap.(The alternative requires hardware tablewalk to create TTEs for RA to PA translations (including a page size and permissions)).

Hardware Tablewalk is threaded and pipelined; up to four TSB accesses for each of the eight threads can be in the pending at one time. The basic dataflow is pipelined, so that a single instance of the dataflow supports all eight threads.

9.3.1 Translation Storage Buffer Access

Hardware Tablewalk uses the TSB Configuration Registers and the VA of the access to calculate the address of the TTE to examine. The TSB Configuration Register provides the base address of the TSB as well as the number of TTEs in the TSB and the size of the pages translated by the TTEs.

(This implies that all TTEs within a given TSB share a common page size.)

Hardware Tablewalk uses a Nonzero Context TSB Configuration Register if the context of the request is nonzero; otherwise it uses a Zero Context TSB Configuration Register. The context of the request is assumed to be the content of Context Register 0 (in the event of a TLB miss on a Primary or Secondary Context access). Hardware Tablewalk uses the page size from the TSB Configuration Register to calculate the presumed VPN for the given VA. Hardware Tablewalk then uses the number of TTE entries and the presumed VPN to generate an index into the TSB. This index is concatenated with the upper bits of the base address to generate the TTE address.

Hardware Tablewalk forwards the TTE address to the gasket, which forwards the load request to the L2. At some later time, the L2 returns the TTE to the gasket. The gasket forwards the TTE to Hardware Tablewalk.

Hardware Tablewalk compares the VPN and context of the request to that from the TTE. If they match, Hardware Tablewalk translates the RPN into a PPN and forwards the TTE to the TLU. If the VPN and context do not match, Hardware Tablewalk waits for the rest of the enabled TSBs to return TTEs; Hardware Tablewalk supports four TSBs per thread for zero contexts and four for nonzero contexts. In the event that no TSB matches, the MMU signals the TLU to take the appropriate trap.

Hardware Tablewalk supports three TSB search modes:

- Sequential: TSB 0 is searched first. If the access misses TSB 0, then TSB 1 is searched. If the access misses TSB 1, then TSB 2 is searched. If the access misses TSB 2 misses, then TSB 3 is searched.
- Prediction: A history table indicates whether to search TSB 0 or TSB 1 first. If TSB 0 is searched first and misses, then TSB 1 is searched. If TSB 1 is searched first and misses, then TSB 0 is searched. After TSB 0 and TSB 1 are searched and miss, then TSB 2 and TSB 3 are searched in the same manner as in the sequential mode.
- Burst: Requests to load TTEs from all enabled TSBs are sent to the L2 before any TTE is checked. Burst mode creates more traffic to and from the L2 but can reduce hardware tablewalk latency in some situations.

In some configurations, Hardware Tablewalk ignores the context match. OpenSPARC T2 does not support the TSB hash register due to area concerns.

9.3.2 Multiple Contexts

Multiple Primary and Secondary Contexts permit different processes to share TTEs within the TLBs. The Use_Context_0 and Use_Context_1 bits in the TSB Configuration Register disable the context match for Hardware Tablewalk. Hardware Tablewalk ignores the contexts in the TSB TTEs if either of these bits is active for requests with nonzero contexts. If either bit is one and the VPN matches, Hardware Tablewalk signals the TLB to write either context 0 or context 1 (depending on which bit is set) as the context of the TTE when it is loaded (instead of the context in the TTE itself). Hardware tablewalk ignores these bits for requests with zero contexts.

9.3.3 Real Page Number To Physical Page Number Translation

When Hardware Tablewalk fetches a TTE from a TSB, it can translate the Real Page Number in the TTE into a Physical Page Number. The TLBs store this PPN. The TLBs use this PPN to translate VAs into PAs. The hypervisor controls the RPN to PPN translation mechanism.

The RPN to PPN translation mechanism provides both range checking as well as mapping of address ranges from one location to another. The translation mechanism uses the RPN and page size in the TTE and calculates the starting and ending addresses for the specified real page. It then checks that these addresses lie in one of four ranges specified by the Real Range registers. If the real page lies completely inside one of the ranges (and the range is enabled), then the translation mechanism adds the RPN in the TTE to the corresponding field in the Physical Offset Register to create the Physical Page Number. If the real page does not lie completely within either range, then the MMU signals a Real_Address_MMU_Miss exception to the TLU. Each thread has four dedicated ranges with corresponding physical offsets. The RPN to PPN translation does not depend on the context value being zero or nonzero.

9.3.4 Translation Storage Buffer

The Translation Storage Buffer is a region in memory that is managed by the supervisor. The TSB holds the TTEs created by the supervisor to allow the supervisor and user code access to VA to RA translations. Hardware tablewalk accesses TSBs in response to ITLB and DTLB misses.

9.3.4.1 TSB TTE Formats

The TSBs provide software functionality and compatibility with previous microprocessors.

The TTE has a tag and a data section. The tag holds the context and the virtual page number, which the hardware tablewalk state machine compares to the accesses. If the context and page number match, then the hardware tablewalk checks the permission bits and provides the physical page number.

The MMU supports the sun4v TTE format in the TSB.

TABLE 9-3 Sun4v TTE Tag Format

-		*Context*		-		*VA[47:22]*	
63	61	60	48	47	26	25	0

TABLE 9-4 Sun4v TTE Data Format For OpenSPARC T2

V	*NFO*	*L*	*-*		*RA[39:13]*		*IE*	*E*	*CP*	*-*	*E*	*P*	*W*	*-*		*Size[2:0]*
63	60	61	60	40	58	49	12	11	10	9	8	7	6	5	3	

Supervisor code maintains the TSBs, but the supervisor cannot access physical addresses (PAs) directly. The TSB TTE can hold a real address (RA) instead of a PA. RA to PA translation occurs in hardware tablewalk or in the hypervisor code itself.

Since the OpenSPARC T2 core has no virtually addressed caches, the CV bit is reserved.

9.4 ASI Registers

The MMU has many associated control and configuration registers.

9.4.1 TLB Registers

These registers control and configure the TLBs

9.4.1.1 Context Registers

Each thread has two Primary and two Secondary context registers per thread. Threads can also use the Nucleus context, which is hardwired to a value of zero; there is no physical Nucleus context register.

TABLE 9-5 Primary Context Registers 0 and 1

Reserved		*Primary Context*	
63	13	12	0

TABLE 9-6 Secondary Context Registers 0 and 1

Reserved		*Secondary Context*	
63	13	12	0

The ITLB always uses either the Primary or the Nucleus contexts as the context for the reference. The DTLB can use the Primary, Secondary, or Nucleus contexts.

Writes to Primary Context Register 0 also update Primary Context Register 1. Similarly, writes to Secondary Context Register 0 also update Secondary Context Register 1. Through this mechanism, software that is unaware of multiple contexts will still operate as expected without having to set enables for the Primary and Secondary Context Registers 1; writes to Context Registers 0 effectively make the hardware act as though there is only one Primary or Secondary Context Register. (Writes to the Primary and Secondary Context Registers 1 do not affect the contents of the Primary or Secondary Context Registers 0.)

The context of a miss is assumed to be the value of the Context Register 0. The Context Register 0 contents are written to the Tag Access Register on a miss.

9.4.1.2 Partition ID Register

Each thread has a Partition ID Register. The partition ID prevents different logical partitions from sharing TTEs.

TABLE 9-7 Partition ID Register

Reserved		*Partition ID*	
63	3	2	0

9.4.1.3 TLB Maintenance Registers

OpenSPARC T2 implements a set of registers per TLB for each thread that provide software control to the TLB contents. These registers include

DATA_IN Register

DATA_ACCESS Register

TAG_READ Register

TAG_DATA Register

DEMAP Register

9.4.2 Hardware Tablewalk Registers

9.4.2.1 TSB Configuration Registers

OpenSPARC T2 implements a total of eight TSB Configuration Registers per thread. Each thread has a set of four TSB Configuration Registers for zero context accesses, and a set of four TSB Configuration Registers for nonzero context accesses. This permits Hardware Tablewalk to access up to four TSBs per reload request. Hardware Tablewalk is disabled by setting the Enable bits in the TSB Configuration Registers to zero.

The Enable bit controls whether hardware tablewalk searches this TSB.

The Use_Context_0 and Use_Context_1 bits control whether hardware tablewalk checks the context value in the TTE from this TSB and what context value is written into the TTE in the TLB. If both bits are 0, then hardware tablewalk compares the context in the TTE from the TSB to the context of the request and stores that context into the TLB if the TTE matches. If either bit is 1, hardware tablewalk ignores the context of the TTE from the TSB. If Use_Context_0 is 1, hardware tablewalk writes the value of Context Register 0 to the TLB; otherwise if Use_Context_1 is 1, hardware tablewalk writes the value of Context Register 1 to the TLB.

The TSB_Base and TSB_Size describe the location and size of the TSB. The number of entries in the TSB is equal to $2^{TSB\_Size} * 512$. The TSB_Base gives the upper bits of the address of the TSB (which is aligned by hardware to the size of the TSB in bytes, which is $2^{TSB\_Size} * 8$ KB).

The RA_not_PA bit activates RPN to PPN translation in hardware tablewalk. (A given TSB can hold either RAs or PAs but not both.)

The Page_Size is the size of the pages mapped by the TTEs in the TSB. This page size is used to generate the TSB pointer.

9.4.2.2 Real Range Registers

OpenSPARC T2 implements four Real Range Registers per thread. The RPN to PPN translation associates each Real Range Register with its corresponding Physical Offset Register. The RA to PA translation applies to TTEs from TSBs with the RA_not_PA bit set, regardless of zero or nonzero context.

If the Enable field is 0, then this range and offset pair is not used. If all range and offset pairs are disabled, any hit in a TSB with the RA_not_PA bit set results in a Real_Address_MMU_Miss trap.

9.4.2.3 Physical Offset Registers

OpenSPARC T2 implements four Physical Offset Registers per thread. The RPN to PPN translation associates each Physical Offset Register with its corresponding Real Range Register.

9.4.2.4 TSB Pointer Registers

OpenSPARC T2 implements a TSB Pointer Register for each TSB Configuration Register. The TSB Pointer Register holds the address into the TSB, based on the current values of

the TSB Configuration Register

the VA in the Tag Access Register

9.4.3 Scratchpad Registers

Each thread has six privileged scratchpad registers for supervisor use. Normally a processor provides eight scratchpad registers, so accesses to two scratchpad addresses cause a data_access_exception trap; the hypervisor emulates the two missing supervisor scratchpad registers.

Each thread has two hyperprivileged scratchpad registers. Only the hypervisor can access these registers.

CHAPTER 10

Reliability And Serviceability (RAS)

Soft errors fall into one of four classes: reported corrected (RC), reported uncorrected (RU), silent corrected (SC), and silent uncorrected (SU). The OpenSPARC T1 design minimizes silent errors, whether corrected or uncorrected. Most SRAMs and register files have ECC or parity protection. OpenSPARC T2 protects more arrays or increases protection of a given array by adding ECC or parity, or by duplicating arrays to further reduce the silent error rate and the reported uncorrected (e.g., fatal) error rate. OpenSPARC T2 cores do not support lockstep, checkpoint, or retry operations.

This chapter outlines the OpenSPARC T2 core RAS features. The expected FIT rates of OpenSPARC T2 microarchitectural structures drive the RAS features. OpenSPARC T2 has additional chip RAS features which are not described here (related to 3GIO and 10G Ethernet, for example).

The RAS design considers four major types of structures for protection. The first type is 6-device, single-ported SRAM cells optimized for density, such as cache data arrays. These SRAM cells have high Failure In Time (FIT) rates (300-400 FITs per Mb in Epic8c). SRAMs that store critical data have ECC protection. Other SRAMs have parity protection. The second type is register files, which are typically multi-ported. A register file cell has FIT rates on the order of 1/2 or less of a high-density SRAM cell. OpenSPARC T2 protects register files with parity, or with ECC where a single-bit error cannot be tolerated. The third type is flip-flops or latches used in datapath or control blocks. A Flop has a FIT rate of 1/3 or less of a single-ported SRAM cell. In general, OpenSPARC T2 does not protect flops or latches. Flops and latches have parity or ECC protection where they are part of a larger datapath which is so protected. The fourth type is a CAM cell, whose FIT rate may be 1/2 of a standard SRAM cell. CAM cells are difficult to protect. Adding parity to a CAM cell eliminates false CAM hits due to single-bit errors, but cannot detect false misses. OpenSPARC T1 “scrubs” large CAMs. CAM scrubbing is different from traditional DRAM scrubbing. As in DRAM scrubbing, CAM scrubbing reads a CAM location

and checks its parity. Unlike DRAM scrubbing, CAM scrubbing cannot correct single-bit failures in all cases: if parity is bad and hardware cannot innocuously reload the entry, an error results.

The FIT rates for OpenSPARC T2 structures are similar to their OpenSPARC T1 counterparts. To improve FIT rates for the core and L2, OpenSPARC T2 protects structures that are unprotected on OpenSPARC T1 and improves protection on structures already protected on OpenSPARC T1. Alternatively, OpenSPARC T2 may redesign structures with a higher Qcrit to lower the FIT rates. This chapter is preliminary, pending detailed technology and circuit design studies to establish Epic9 FIT rates.

OpenSPARC T2 contains error status registers (ESRs) which describe hardware errors to software. The ESRs isolate the first error. In the case of multiple errors, they also indicate that multiple errors have occurred. Software can read and write the registers via ASI instructions. Software can use a software controlled bit in the register to emulate a parity or ECC error (to allow debug of diagnostic software). In addition, the structures protected by parity or ECC provide mechanisms to inject errors (for test). The complete specification of the error detection, correction, logging, and diagnostic registers is contained in the OpenSPARC T2 PRM.

In this chapter, the term “core” refers to a virtual core, or a specific thread on a physical core (e.g., core 20 refers to thread 4 on physical core 2). Since OpenSPARC T2 has 8 physical cores with 8 threads each, cores are numbered from 0 to 63, inclusively.

Hardware-detected errors can either be attributed directly to a specific core, or not. An example of the former is an instruction cache tag parity error during an instruction fetch. An example of the latter is an uncorrectable error on the write-back of a modified L2 cache line.

An error which can be attributed to a given core can either be precise or imprecise (disrupting). For example, an ITLB parity error is precise. An uncorrectable error on a read of a core's store queue data entry is imprecise. Even though the store instruction is known, the core has updated architectural state past the store by the time the store data is read from the store queue.

OpenSPARC T2 logs errors which are attributable to a given core in an ESR associated with that core. If enabled, these errors result in either precise, disrupting, or deferred traps. OpenSPARC T2 logs errors which are not attributable to a given core in a “global” ESR and, if enabled, directs a disrupting trap to the core identified in the ASI_CMP_ERROR_STEERING register.

Each major structure on the OpenSPARC T2 core with a significant FIT rate has an error detection scheme. The scheme for each structure depends on the way the structure is used and the effect of the scheme on the timing and physical layout. These schemes seek to reduce the numbers of silent errors and of reported uncorrected errors.

The design defines specific hardware behavior for each recorded error. Handling of each error follows a general template. Hardware corrects any correctable errors and retries the operation (either implicitly or explicitly). If a structure does not support ECC or if the structure detects an uncorrectable error, the structure invalidates the corrupted data. After invalidation, the core retries the operation (either implicitly or explicitly). If the data cannot be invalidated or another error occurs as a result of a retry, hardware signals an unrecoverable error and requests a trap for the affected core.

In parallel with error handling within the affected core, OpenSPARC T2 can request traps for arbitrary cores. The ASI_CMP_ERROR_STEERING register controls disrupting trap requests for arbitrary cores in response to corrected and uncorrected errors.

10.1 ITLB

The ITLB is implemented as a CAM and a data array. The 64 entry CAM stores the virtual or real address tag, while the 64 entry data array stores the physical address and page attributes. On OpenSPARC T2, the data array and the CAM are each protected with a parity bit. On a hit to a ITLB entry, the parity of the matching CAM entry and the associated data entry is checked. Parity is not checked for CAM entries that miss.

The ITLB can be accessed by normal instruction fetches or with ASI loads to ASI_ITLB_DATA_ACCESS_REG. Parity is not checked for diagnostic array accesses.

OpenSPARC T2 can be configured to have hardware tablewalks enabled (HWTW). ITLB error handling depends upon the access type and whether or not hardware tablewalks are enabled. All ITLB errors result in an Instruction_Access_MMU_Error trap or a Fast_Instruction_Access_MMU_Miss trap.

10.1.1 MRA or L2 error on an ITLB Miss with HWTW Enabled

- Note that on a “normal” ITLB miss with HWTW enabled, an error can occur from the MRA (The MRA contains pointers and configuration information used for hardware tablewalk.) or the L2 cache.
- For an MRA error, the I-SFSR contains the proper encoding for an ITMC or ITMU error. The index of the failing MRA location is also recorded in the D-SFAR. Hardware takes a precise, non-maskable Instruction_Access_MMU_Error trap.

The VA of the instruction fetch is available in TPC[TL]. To recover from an MRA error, software can read the failing MRA location from the D-SFAR. By using the MRA diagnostic ASI access, software can attempt to perform a correction by reading the failing data and ECC check bits, computing the syndrome, writing corrected data to the MRA, then retrying the original instruction. Even if the MRA has an uncorrectable error, software may be able to recover if it has a "clean" copy of the MRA data elsewhere.

- For an L2 error, one of the I-SFSR ITL2C, ITL2U, or ITL2ND encodings are set. L2 records the physical address where the error occurred and the error type (CE, UE, NotData) in an ESR. The VA of the instruction fetch is available in TPC[TL]. Hardware takes a precise, non-maskable Fast_Instruction_Access_MMU_Miss trap. Software can attempt recovery from an L2 error by correcting the failing L2 data or invalidating the L2 line as appropriate (details will be specified in the L2 RAS document). Then it can retry the instruction.

10.1.2 ITLB CAM Parity Error

There are three access cases which can result in an ITLB CAM parity error. The cases are:

- Accessing a non-locked entry with HWTW enabled
- Accessing a non-locked entry with HWTW disabled
- Accessing a locked entry

In each case, hardware does not invalidate the entry with the error. Hardware logs the error by encoding ITTP in the I-SFSR, and takes a precise Instruction_Access_MMU_Error trap. The VA of the instruction fetch is recorded in TPC[TL]. Software at the trap handler logs the error, and issues a demap_page to the TPC[TL]. Then it issues a retry instruction. Hardware refetches the instruction and reaccesses the ITLB. This time either a hit will occur (since the translation was reloaded by another thread), or a miss will occur. If an ITLB miss occurs, hardware retranslates the address and reloads the ITLB.

This trap is not maskable. Since the trap goes to hypervisor mode, no further ITLB errors will occur for this thread until translation is re-enabled. Multiple error traps can occur at the same time if different threads try to access the same VA.

10.1.3 ITLB CAM Multiple Hit Error, Same or Different Contexts

The OpenSPARC T2 ITLB checks for multiple CAM hits on each access. A multiple CAM hit error has higher priority than a CAM parity error. A multiple hit can occur for the same context (which implies a hardware error, as each ITLB reload by either hardware or software has an implicit auto-demap). A multiple hit can also occur if software maps the same virtual address using different contexts (which is not a hardware error). However, if multiple bits flip in the ITLB CAM entry, it is possible for hardware to detect multiple hits to different contexts even though this was not created by software.

When a multiple CAM hit occurs, hardware logs the error by encoding ITTM in the I-SFSR, and takes a precise Instruction_Access_MMU_Error trap. The VA of the instruction fetch is recorded in TPC[TL]. Software at the trap handler logs the error, and issues a demap_all to the ITLB. Then it issues a retry instruction. Hardware refetches the instruction and reaccesses the ITLB. This time either a hit will occur (since the translation was reloaded by another thread), or a miss will occur. If an ITLB miss occurs, hardware retranslates the address and reloads the ITLB. Whether the new translation will be bypassed to allow forward progress in the presence of a stuck ITLB fault is under discussion.

This trap is not maskable. Since the trap goes to hypervisor mode, no further ITLB errors will occur for this thread until translation is re-enabled. Multiple error traps can occur at the same time if different threads try to access the same VA.

10.1.4 ITLB Data Parity Error

Similar to ITLB CAM parity errors, there are three types of access which can result in a data parity error. The access types are:

- Accessing a non-locked entry with HWTW enabled
- Accessing a non-locked entry with HWTW disabled
- Accessing a locked entry

Hardware handles the cases the same as an ITLB CAM parity error, but logs an ITDP error in the I-SFSR. Hardware takes a precise Instruction_Access_MMU_Error trap. The VA of the instruction fetch is logged in TPC[TL]. Software logs the error and demaps the page. It then issues a retry. Hardware refetches the instruction and reaccesses the ITLB. This time either a hit will occur (since the translation was reloaded by another thread), or a miss will occur. If an ITLB miss occurs, hardware retranslates the address and reloads the ITLB.

This trap is not maskable. Since the trap goes to hypervisor mode, no further ITLB errors will occur for this thread until translation is re-enabled. Multiple error traps can occur at the same time if different threads try to access the same VA.

10.2 Instruction Cache

The L1 instruction cache contains tag, data, and valid bit arrays. Parity protects the data and tag arrays from silent single-bit errors. One parity bit protects each instruction word (4 Bytes), and one parity bit protects each tag entry. The valid bits are duplicated in the valid bit array. Instruction cache arrays are read during an instruction fetch or during a diagnostic load from the arrays. Parity is not checked for diagnostic reads of the Icache arrays. As a result, diagnostic reads to the Icache arrays (valid, tag, or data) can not cause errors and no error resulting from such an access is recorded in the I-SFSR or D-SFAR.

All Icache array parity errors or valid bit mismatch errors result in a disrupting trap if the appropriate CERER and CETER.DHCCE bits are set.

10.2.1 Normal Icache Miss

On an Icache miss, hardware can get a correctable, uncorrectable, or NotData error from L2. The error is reported in the L2 return packet.

If a correctable error occurs, and SETER.DHCCE is set, hardware sends an "error nop" down the pipe to the trap unit. The trap unit records ICL2C in the DESR, and, if enabled, hardware takes a disrupting HW_Corrected_Error trap. If SETER.DHCCE is not set, hardware continues executing.

If the response from L2 indicates an uncorrectable or NotData error, hardware loads the line with bad parity in the Icache data array and sends an "error nop" down the pipe to the trap unit. If SETER.PSCCE is set, hardware records ICL2U or ICL2ND in the I-SFSR and takes a precise Instruction_Access_Error trap. If SETER.PSCCE is not set, hardware continues executing using the invalid data read from L2.

Software can attempt recovery from an L2 error by correcting the failing L2 data or invalidating the L2 line as appropriate, then issuing a retry to reexecute the failing instruction.

10.2.2 Icache Valid Bit Array Mismatch on Instruction Fetch

Hardware invalidates all ways in the cache index which had the mismatch, (The Icache invalidates the line by sending an invalidate request to the L2. L2 returns an "invalidate all ways" request to the Icache.) and sends an "error nop" down the pipe to the trap unit. The trap unit issues a refetch to the failing address, signaling the IFU

to request an invalidation of all ways in the cache from the L2, followed by a refetch of the line. Hardware logs the error by setting ICVP and writing the way and index of the error to the DESR.

If SETER.DHCCE is set, hardware takes a disrupting HW_Corrected_Error trap so software can log the error. OpenSPARC T2 will in fact take the trap at the instruction which received the error, so TPC[TL] will point to the instruction which encountered the error (assuming PSTATE.IE is set or HPSTATE.HPRIV is clear).

If SETER.DHCCE is not set, hardware refetches the instruction. The instruction fetch should miss the cache, and the instruction is bypassed to allow forward progress even if there is a stuck bit in the cache.

10.2.3 Icache Tag Parity Error on Instruction Fetch

Hardware handles a tag parity error on an instruction fetch the same way a valid bit mismatch is handled, except it encodes ICTP and the index and the way with the error in the DESR.

10.2.4 Icache Tag Multiple Hit Error on Instruction Fetch

Hardware handles a tag parity error on an instruction fetch the same way a valid bit mismatch is handled, except it encodes ICTM and the index and one of the ways which hit in the DESR.

10.2.5 Icache Data Parity Error on Instruction Fetch

Hardware handles a tag parity error on an instruction fetch the same way a valid bit mismatch is handled, except it encodes ICDP and the index and the way which hit in the DESR.

10.3 Integer Register File

The IRF can be accessed via normal instructions.

There are two copies of the IRF, one for thread group 0 and one for thread group 1.

Each IRF entry is protected by SEC/DED ECC with 8 check bits. Up to 3 operands can be read from the IRF at a time. Hardware checks each operand's ECC independently. Each read port checks (but does not correct) the ECC of its associated data. Hardware prioritizes operand errors in the order rs1 > rs2 > rs3. This means that an uncorrectable error which occurs on a lower-priority operand (e.g., rs2) simultaneously with a correctable error on a higher priority operand (e.g., rs1) will not be reported (until the correctable error is cleared by software and the instruction is retried).

When an ECC error is detected, the EXU signals a trap request to the TLU and self-flushes the instruction.

If hardware detects either a correctable or uncorrectable error for any valid operand, what happens depends upon the setting of CETER.PSCCE.

If CETER.PSCCE is set, hardware records the error type in the D-SFSR by encoding IRFC or IRFU as appropriate, and records the IRF index and ECC check bits in the D-SFAR. It generates a precise Internal_Processor_Error trap request to the core. Hardware vectors to the trap handler. Software should correct an IRFC error before issuing a retry instruction. Software can correct the error as follows. It turns off the CETER.PSCCE bit, and reads the ECC check bits from the D-SFAR. It decodes the failing address location in the IRF, and reads the data. It then recomputes ECC, generates a new syndrome, and corrects the data. It then writes the corrected data into the IRF (hardware will generate the proper ECC upon the write). In the process of reading the failing location, another error will occur, but will not be logged or trapped. Software should then turn the CETER.PSCCE bit back on. Then software can retry the original failing instruction.

An IRFU error is generally not recoverable.

If the CETER.PSCCE bit is not set, the error is not recorded, and hardware continues executing, using the uncorrected data read from the IRF. This will lead to data corruption.

Since up to three operands can be read for each instruction, software may define an appropriate error threshold for the instruction before considering the IRF broken.

10.4 Floating-Point Register File

OpenSPARC T2 has one FRF. It is a multi-ported register file with 2 read ports and 2 write ports. OpenSPARC T2 protects the FRF with SEC/DEC ECC. Error handling is similar to the IRF. If the FGU detects an error, it self-flushes the instruction and signals a trap request to the TLU. Software must correct the FRF error before retrying the instruction.

The FRF can be accessed via normal instructions.

Each FRF entry is protected by SEC/DED ECC. Up to two operands can be read from the FRF at a time. Hardware checks each operand's ECC independently. Note: PDIST reads 3 operands over 2 cycles, so hardware still prioritizes 3 operands for reporting errors. Hardware prioritizes operand errors in the order rs1 > rs2 > rs3. This means that an uncorrectable error which occurs on a lower-priority operand (e.g., rs2) simultaneously with a correctable error on a higher priority operand (e.g., rs1) will not be reported (until the correctable error is cleared by software and the instruction is retried).

If hardware detects either a correctable or uncorrectable error for any valid operand, what happens depends upon the setting of CETER.PSCCE.

If CETER.PSCCE is set, hardware records the error type by encoding FRFC or FRFU in the D-SFSR as appropriate, and records the failing FRF index and ECC check bits in the D-SFSR. It generates a precise Internal_Processor_Error trap request to the core. Hardware vectors to the trap handler, and software should correct a correctable error before issuing a retry instruction. Handling of an FRFC error is similar to an IRFC error. The additional complication is that each FRF entry contains two ECC words (due to the single-precision FP registers). So two corrections may have to be performed.

Software can correct the error as follows. It turns off the CETER.PSCCE bit, and reads the ECC check bits from the D-SFSR. It decodes the failing address location in the FRF, and reads the data. It then recomputes ECC, generates a new syndrome, and corrects the data. It then writes the corrected data into the FRF (hardware will generate the proper ECC upon the write). In the process of reading the failing location instruction, another error will occur, but will not be logged or trapped. Software should then turn the CETER.PSCCE bit back on. Then software can retry the original failing instruction.

If the CETER.PSCCE bit is not set, the error is not recorded, and hardware continues executing, using the uncorrected data read from the FRF. This will lead to data corruption.

10.5 Data TLB

The DTLB is implemented as a CAM and a data array. The 128 entry CAM stores the virtual or real address tag, while the 128 entry data array stores the physical address and page attributes. On OpenSPARC T2, the data array and the CAM are each protected with a parity bit. On a hit to a DTLB entry, the parity of the matching CAM entry and the associated data entry is checked. Parity is not checked for CAM entries that miss.

The DTLB can be accessed by normal instruction fetches or with ASI loads to ASI_DTLB_DATA_ACCESS_REG. Parity is not checked for diagnostic array accesses.

OpenSPARC T2 can be configured to have hardware tablewalks enabled (HWTW).

DTLB error handling depends upon the access type and whether or not hardware tablewalks are enabled.

All DTLB errors result in either a Data_Access_MMU_Error trap, or a Fast_Data_Access_MMU_Miss trap.

10.5.1 MRA or L2 Error on a DTLB Miss with HWTW Enabled

- Note that on a "normal" DTLB miss with HWTW enabled, an error can occur from the MRA or the L2 cache. In that case, error handling is as follows.
- For an MRA error, the D-SFSR contains the proper encoding for an DTMC or DTMU error. The index of the failing MRA location is also recorded in the D-SFAR. Hardware takes a precise, non-maskable Data_Access_MMU_Error trap. To recover from an MRA error, software can read the failing MRA location from the D-SFAR. By using the MRA diagnostic ASI access, software can attempt to perform a correction by reading the failing data and ECC check bits, computing the syndrome, writing corrected data to the MRA, then retrying the original instruction. Even if the MRA has an uncorrectable error, software may be able to recover if it has a "clean" copy of the MRA data elsewhere.
- For an L2 error, one of the D-SFSR DTL2C, DTL2U, or DTL2ND encodings are set. L2 records the physical address where the error occurred and the error type (CE, UE, NotData) in the DESR or D-SFSR. Hardware takes a precise, non-maskable Fast_Data_Access_MMU_Miss trap. The VA of the data access is available in the D-SFAR. Software can attempt recovery from an L2 error by correcting the failing L2 data or invalidating the L2 line as appropriate (details will be specified in the L2 RAS document). Then it can retry the instruction.

10.5.2 DTLB CAM Parity Error

There are three access cases which can result in an DTLB CAM parity error. The cases are:

- Accessing a non-locked entry with HWTW enabled
- Accessing a non-locked entry with HWTW disabled
- Accessing a locked entry

In each case, hardware does not invalidate the entry with the error. Hardware logs the error by encoding DTTP in the D-SFSR, and takes a precise Data_Access_MMU_Error trap. The VA of the data access is recorded in the D-SFAR. Software at the trap handler logs the error, and issues a demap_page to the VA of the data access. Then it issues a retry instruction. Hardware refetches the instruction and reaccesses the DTLB. This time either a hit will occur (since the translation was reloaded by another thread), or a miss will occur. If a DTLB miss occurs, hardware retranslates the address and reloads the DTLB. Whether the new translation will be bypassed to allow forward progress in the presence of a stuck DTLB fault is under discussion.

This trap is not maskable. Since the trap goes to hypervisor mode, no further DTLB errors will occur for this thread until translation is re-enabled. Multiple error traps can occur at the same time if different threads try to access the same VA.

10.5.3 DTLB CAM Multiple Hit Error, Same or Different Contexts

The OpenSPARC T2 DTLB checks for multiple CAM hits on each access. A multiple CAM hit error has lower priority than a CAM parity error. A multiple hit can occur for the same context (which implies a hardware error, as each DTLB reload by either hardware or software has an implicit auto-demap). A multiple hit can also occur if software maps the same virtual address using different contexts (which is not a hardware error). However, if multiple bits flip in the DTLB CAM entry, it is possible for hardware to detect multiple hits to different contexts even though this was not created by software.

When a multiple CAM hit occurs, hardware logs the error by encoding DTMH in the D-SFSR, and takes a precise Data_Access_MMU_Error trap. The VA of the data access is recorded in the D-SFAR. Software at the trap handler logs the error, and issues a demap_all to the DTLB. Then it issues a retry instruction. Hardware refetches the instruction and reaccesses the DTLB. This time either a hit will occur (since the translation was reloaded by another thread), or a miss will occur. If a DTLB miss occurs, hardware retranslates the address and reloads the DTLB. Whether the new translation will be bypassed to allow forward progress in the presence of a stuck DTLB fault is under discussion.

This trap is not maskable. Since the trap goes to hypervisor mode, no further DTLB errors will occur for this thread until translation is re-enabled. Multiple error traps can occur at the same time if different threads try to access the same VA.

10.5.4 DTLB Data Parity Error

Hardware handles this case the same as a DTLB CAM parity error, but logs a DTDP error in the D-SFSR. Hardware takes a precise Data_Access_MMU_Error trap. The VA of the data access is logged in the D-SFAR. Software logs the error and demaps the page. It then issues a retry. Hardware refetches the instruction and reaccesses the DTLB. This time either a hit will occur (since the translation was reloaded by another thread), or a miss will occur. If a DTLB miss occurs, hardware retranslates the address and reloads the DTLB. Whether the new translation will be bypassed to allow forward progress in the presence of a stuck DTLB fault is under discussion.

This trap is not maskable. Since the trap goes to hypervisor mode, no further DTLB errors will occur for this thread until translation is re-enabled. Multiple error traps can occur at the same time if different threads try to access the same VA.

10.6 Data Cache

The L1 data cache maintains a parity bit for every byte in the data arrays. One parity bit protects the tag portion of the data cache, and the valid array is duplicated. Parity is checked for all memory loads that access the data cache. Parity and valid bit equality is not checked for diagnostic accesses to the data cache.

10.6.1 Data Cache Miss

On a data cache miss, hardware can get a correctable, uncorrectable, or NotData error from L2. L2 records the error in an L2 ESR.

If a correctable error occurs, hardware records the DCL2C in the DESR, and, if SETER.DE is set, takes a hw_corrected_error trap.

If an uncorrectable or NotData error occurs, and SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding one of DCL2U or DCL2ND as appropriate, then takes a precise Data_Access_Error trap.

Software at the trap handler can attempt recovery from an L2 error by correcting the failing L2 data or invalidating the L2 line as appropriate. Then it can issue a retry to reexecute the instruction.

If CETER.PSCCE is not set, hardware uses the (possibly incorrect) data returned by L2. The error is not recorded. Otherwise unpredictable operation and data corruption may result.

10.6.2 Dcache Valid Bit Error

If a load instruction detects a valid bit mismatch, hardware forces a cache miss, and invalidates all ways in the cache index which had the mismatch (via the L2).

Hardware records the error by encoding DCVP and writing the index and way with the error in the DESR. Hardware completes the load by bypassing the data from L2.

If SETER.DHCCE is set, hardware then takes a disrupting HW_Corrected_Error trap so software can log the error.

If SETER.DHCCE is not set, hardware continues executing, and will take a disrupting HW_Corrected_Error trap when software sets the SETER.DHCCE bit.

10.6.3 Dcache Tag Parity Error on Load

A load instruction which detects a data cache tag parity error is handled the same as if a valid bit mismatch was detected. Hardware invalidates all ways in the cache index with the tag parity error.

Hardware records the error by encoding DCTP and writing the index and way with the error in the DESR. Hardware completes the load by bypassing the data from L2.

If SETER.DHCCE is set, hardware then takes a disrupting HW_Corrected_Error trap so software can log the error.

If SETER.DHCCE is not set, hardware continues executing, and will take a disrupting HW_Corrected_Error trap when software sets the SETER.DHCCE bit.

10.6.4 Dcache Tag Multiple Hit Error on Load

Hardware invalidates all ways if multiple tag hits occur on a load.

Hardware records the error by encoding DCTM and writing the index and ways with the error in the DESR. Hardware completes the load by bypassing the data from L2.

If SETER.DHCCE is set, hardware then takes a disrupting HW_Corrected_Error trap so software can log the error.

If SETER.DHCCE is not set, hardware continues executing, and will take a disrupting HW_Corrected_Error trap when software sets the SETER.DHCCE bit.

10.6.5 Dcache Data Parity Error on Load

A load instruction which detects a data cache data parity error is handled the same as if a valid bit mismatch was detected. Hardware invalidates all ways in the data cache index which had the data parity error.

Hardware records the error by encoding DCDP and writing the index and way with the error in the DESR. Hardware completes the load by bypassing the data from L2.

If SETER.DHCCE is set, hardware then takes a disrupting HW_Corrected_Error trap so software can log the error.

If SETER.DHCCE is not set, hardware continues executing, and will take a disrupting HW_Corrected_Error trap when software sets the SETER.DHCCE bit.

10.7 Store Buffer

The STB is organized as a CAM which contains the tag portion of the address and a RAM which contains the data and status bits. The status bits consist of the privilege level of the store. Each CAM entry is protected by a single parity bit. The data bits are protected via 32b SEC/DED ECC. The store buffer is accessed on data loads (to check for RAW (Read-After-Write) hits) and on PCX reads (A PCX read occurs when the store is sent to the L2 cache. PCX stands for Processor to Cache Xbar). and ASI ring stores. (Stores to ASI space which go over the ASI ring internal to the processor are referred to as ASI ring stores.) It can also be accessed with diagnostic reads, but these accesses do not cause parity or ECC errors.

10.7.1 Correctable Data ECC Error on a Load

If a load which results in a full RAW hit in the STB gets a single-bit data error, hardware does not correct the load data.

If SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding SBDLC, and recording store buffer index in the D-SFAR. Hardware presents a precise Internal_Processor_Error trap to the core. In this case software at the trap handler can issue a Membar #Sync to cause the store buffer to drain. Since hardware will correct the data before writing the store data to memory, this error is likely recoverable; software can issue a retry after the Membar to re execute the load.

If SETER.PSCCE is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

10.7.2 Uncorrectable Data ECC Error on a Load

If a load which results in a full RAW hit in the STB gets an uncorrectable data ECC error, the following flow occurs.

If SETER.PSCCE is set, the error is recorded in the D-SFSR by encoding either SBDLU, and writing the store buffer index to the D-SFAR. Hardware presents a precise Internal_Processor_Error trap to the core. Software at the trap handler can issue a Membar #Sync to cause the store buffer to drain. Another uncorrectable error will likely occur when hardware reads the store buffer entry to write the store data to memory.

If SETER.PSCCE is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

10.7.3 STB Address Parity Error on a Load

Load accesses do not check address parity since the contents of the CAM are not read.

10.7.4 Correctable Data ECC Error on a PCX Read to Memory or I/O or Read for an ASI Ring Store

On a PCX read to memory or I/O space or read for an ASI ring store which results in a single bit ECC error, hardware corrects the error before forwarding the data to the crossbar or the ASI ring. Hardware encodes SBDPC and writes the store buffer index to the DESR.

If SETER.DE is set, hardware presents a disrupting HW_Corrected_Error trap to the core.

If SETER.DE is not set, hardware continues executing. Assuming software has not reset DESR.F, a disrupting trap will be presented to the core when software sets SETER.DE.

10.7.5 Uncorrectable Data ECC Error on a PCX Read to Memory

On a PCX read to memory which results in an uncorrectable ECC error, hardware generates NotData before forwarding the data to the crossbar. The error is recorded in the DESR by encoding SBDPU, and writing the store buffer entry index to the DESR.

If CETER.DE is set, hardware presents a disrupting SW_Recoverable_Error trap to the core.

If CETER.DE is not set, hardware continues executing. Note that if DE is not set, hardware has performed a bad ASI store which will not be detected. When software sets CETER.DE, hardware will present a disrupting SW_Recoverable_Error trap to the core.

10.7.6 Uncorrectable Data ECC Error on a PCX Read to I/O Space or Read for an ASI Ring Store

On a PCX read to I/O space which results in an uncorrectable ECC error, hardware suppresses the store, and all subsequent stores then in the store buffer for that strand. It logs the error in the DFESR by setting SBDIOU and also logs the store buffer index and highest privilege level of all the suppressed stores in the DFESR.

Hardware takes a deferred store_error trap. Software can decide what termination action is appropriate. Software at the trap handler should read the contents of the store buffer using diagnostic reads before issuing any stores which will overwrite the store buffer.

10.7.7 Address Bit Parity Error on a PCX Read or Read for an ASI Ring Store

On a PCX read or an ASI ring store read which exposes a parity error on the address bits, hardware suppresses the store, and logs the error in the DFESR by setting SBAPP (address parity error). The store buffer index which had the error is also logged in the DFESR.STBIndex field. Other (younger) stores in the store buffer are also suppressed. The highest privilege level of any suppressed store is also recorded

Hardware takes a deferred store_error trap.This trap is not maskable.

10.8 Scratchpad Array Errors

The Scratchpad array contains the scratchpad registers. It can be accessed only via normal ASI or diagnostic ASI loads and stores. The array is protected via SEC/DED ECC.

ECC is not checked for diagnostic reads of the array, so a diagnostic read can not result in an error.

If a normal ASI read of the array results in a correctable ECC error, hardware corrects neither the returned data nor the error in the array.

If the SETER.PSCCE bit is set, hardware records the error in the D-SFSR by encoding SCAC, and records the array index with the error in the D-SFAR. Hardware signals a precise Internal_Processor_Error to the core. When software takes the trap, it can correct the data in the array. It issues a diagnostic ASI read to read the data and ECC check bits, computes the correct data, and writes the corrected data and syndrome back using a diagnostic ASI write.

If the SETER.PSCCE bit is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

If a normal ASI read of the array results in an uncorrectable ECC error, and SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding SCAU. The array index with the error is stored in the D-SFAR. Hardware signals a precise Internal_Processor_Error to the core.

If the SETER.PSCCE bit is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

10.9 Tick_compare

The Tick_compare arrays are also protected via SEC/DED ECC. They have two access means. The first is via normal or diagnostic ASI loads and stores. The second, compare access, is implicit as hardware cycles through the entries to compare the Tick/Stick register with the Tick_compare registers.

ECC is checked for a normal ASI load. If a correctable error occurs, hardware corrects neither the returned data nor the array location.

If SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding TCCP and the failing array index is stored in the D-SFAR. Hardware generates a precise Internal_Processor_Error trap to the core. For a correctable error, software at the trap

handler can rewrite the array location and retry the failing instruction. It issues a diagnostic ASI read to read the data and syndrome, computes the correct data, and writes the corrected data and syndrome back using a diagnostic ASI write.

If the SETER.PSCCE bit is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

If an uncorrectable error occurs on a normal ASI load, and SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding TCCU and writes the failing index to the D-SFAR. Hardware takes a precise Internal_Processor_Error trap. Software may be able to recover from this error by picking some reasonable value to load the Tick_compare register with, and retrying the ASI load.

If SETER.PSCCE is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

ECC is not checked for a diagnostic ASI load, so no error is recorded and no trap can occur for this access type.

ECC is checked for a compare access. If a correctable or uncorrectable error occurs, hardware does not correct the data in the array, and suppresses any compare operation. Hardware records the error in the DESR, by encoding either TCCD or TCUD, and writing the failing array index.

If SETER.DE is set, hardware presents a disrupting SW_Recoverable_Error trap to the core. For a TCCD error, software can attempt recovery by using diagnostic array ASI accesses to correct the data as described for TCCP processing above. For a TCUD error, software may be able to recover from the error by taking a tick_compare interrupt, and reloading the tick_compare register after processing completes.

If SETER.DE is not set, hardware continues executing without regard to the error.

10.10 TSA Errors

The TSA array is protected via SEC/DED ECC. It contains the Trap Stack Array and the mondo interrupt queue registers. It can be accessed via normal or diagnostic ASI accesses. ASI writes require a read-modify-write operation, so normal ASI stores can generate an ECC error. The TSA is also accessed during Done and Retry instructions.

If hardware detects a correctable error during a normal ASI access, or a Done or Retry instruction, hardware corrects neither the data returned by the read nor the array location. If the access was an ASI store, hardware suppresses the array write.

If SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding TSAC, and writes the failing TSA index in the D-SFAR. Hardware presents the core with a precise Internal_Processor_Error trap. Software can attempt recovery by using TSA diagnostic ASI accesses to read out the failing data and ECC check bits, correct the data, and write the corrected data and ECC to the failing location.

If SETER.PSCCE is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

If hardware detects an uncorrectable error during the read access for a normal ASI load or store, or a Done or Retry instruction, and SETER.PSCCE is set, it records the uncorrectable error to the D-SFSR by encoding TSAU, and writes the failing array index to the D-SFAR. It then presents the core with a precise Internal_Processor_Error trap. It seems that this error is generally unrecoverable unless (somehow) software knows what the value of the TSA entry should be.

If SETER.PSCCE is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

10.11 MRA Errors

The MRA array contains various pointers used by hardware table-walk and the MMU. Each location is protected via parity. The MRA is accessed by normal ASI reads and writes, diagnostic ASI reads and writes, and for hardware tablewalks. It is also read-modify-write for ASI writes.

If hardware detects a correctable error during a normal ASI access, hardware corrects neither the failing array index nor the data returned by an ASI load. If the access was an ASI store, hardware suppresses the array write.

If SETER.PSCCE is set, hardware records the error in the D-SFSR by encoding MRAU, and writes the failing array index to the D-SFAR. Hardware presents a precise Internal_Processor_Error to the core. Software can attempt recovery by reloading the MRA entry using a copy in memory, and retry the ASI access, thereby recovering from an uncorrectable error.

If SETER.PSCCE is not set, hardware continues executing using the uncorrected, and possibly erroneous, data. The error is not recorded.

Parity is not checked for diagnostic ASI reads and writes.

If an MRA location gets a parity error during a hardware tablewalk, the error results in a precise Instruction_Access_MMU_Error or Data_Access_MMU_Error trap to the core (see previous error handling sections). Software can attempt recovery from an error as above for an ASI access.

10.12 MAMEM Parity Error

The MAMEM (Modular Arithmetic MEMory) contains one parity bit for each 32-bit half of each 64-bit entry. Parity is generated whenever the MA unit writes data to the MAMEM (either for an MAMEM load operation or as a result of a MA arithmetic operation). Parity is checked when data is read (either for an MAMEM store operation, or for an MA arithmetic operation).

The MAMEM array is dual-ported. The syndrome information contains the port number, and the index which was being read. If a parity errors occurs at the same cycle during a read of both ports of the MAMEM array, read port 1 is prioritized over read port 2.

If a parity error occurs during an MAMEM operation, hardware terminates the current MA operation, sets a failing status bit in the MA_CTL register, records an MAMU error along with the failing location and port number of the MAMEM array in the DESR, and, if CETER.DE is set, causes a disrupting SW_Recoverable_Error trap to the core whose id is contained in the TID field of the MA_CTL register.

Although the hardware can not correct the error, software can recover from the error by retrying the MA operation from the beginning. An MA operation generally consists of an MA load, one or more arithmetic operations, and an MA store to write the results to memory. In order to retry the complete MA operation, software must preserve a copy of the original MAMEM operands. This implies that the location of the MAMEM store should not overlap the MAMEM load data.

10.13 L2 Errors

The L2 cache is interleaved 8 ways. Each bank operates independently. When an L2 bank detects an error, the error either can or can not be precisely associated with a core access. If the error can be precisely associated with a core access (such as any core read operation except prefetch), then that error is signalled to the thread by one of the error types mentioned above (e.g., ITL2C).

If the error can not be precisely associated with a core read operation, or is a prefetch, L2 generates an error packet and sends it to the core. The thread which receives the trap is either the thread which initiated the operation, or the thread which is specified in the L2 error steering register. Examples of these types of accesses include:

Partial stores from the core

L2 write-backs

L2 scrub operations (data arrays)

DRAM fetches for I/O or core stores

L2 I/O accesses

These errors are logged in the DESR by setting L2C, L2U, or L2ND. Correctable errors are corrected by L2 hardware and, if enabled, result in a HW_Corrected_Error disrupting trap. Uncorrectable or NotData errors require software intervention. When signalled by an L2 bank to the core, and enabled, the core takes an SW_Recoverable_Error disrupting trap.

Like other disrupting traps, if the error is not enabled due to PSTATE.IE or SETER.DE being zero, hardware presents the trap when software sets both PSTATE.IE and SETER.DE.

10.14 Error Registers

OpenSPARC T2's error registers are described in the *OpenSPARC T2 Programmer's Reference Manual*..

CHAPTER 11

ASI/ASR/HPR/PR Access

OpenSPARC T2 conceptually has ASI “rings” to access registers defined in ASI space. These registers are accessed using Load and Store alternate instructions. Access to Ancillary State Registers (ASR), Privileged Registers (PR), and Hyperprivileged Registers (HPR) via RDASR/WRASR, RDPR/WRPR, and RDHPR/WRHPR instructions also occur over the ASI rings. Briefly, there are three logical rings: fast, local, and global.

The width of the ASI ring bus is 65 bits.

This chapter is organized as follows. First, the locations of known registers are described. Then the operation of the ASI ring is described.

11.1 Register Locations

The following tables list ASI registers and their unit locations. (TLU and MMU are on the fast ring, all other units are on the local ring.) If a register is not shown it is either not implemented or does not access an internal register. The 'Synchronizing' column indicates whether a write to this register causes a post-sync, to enable subsequent instructions from that thread to immediately see the effects of the store. The 'Determinate' column indicates whether the access to this register always has a defined latency once it is on the ring.

11.1.1 Ancillary State Registers

TABLE 11-1 OpenSPARC T2 ASR Register Locations

Register Name	*ASR address*	*Unit Location(s)*	*Synchronizing on write*	*Determinate*
Y	0	EXU	Y	Y
CCR	2	EXU	Y	Y
ASI	3	LSU, TLU	Y	Y
TICK	4	TLU	Y	Y
PC	5	TLU	Y	Y
FPRS	6	FGU	Y	Y
(SIR)	15	TLU	Y	Y
PCR	16	PMU	Y	Y
PIC	17	PMU	Y	Y
GSR	19	FGU	Y	Y
SET_SOFTINT	20	TLU	Y	Y
CLEAR_SOFTINT	21	TLU	Y	Y
SOFTINT	22	TLU	Y	Y
TICK_COMPARE	23	TLU	Y	Y
STICK	24	TLU	Y	Y
STICK_COMPARE	25	TLU	Y	Y

11.1.2 Hyperprivileged Registers

TABLE 11-2 Hyperprivileged register locations

Register Name	*HPR address*	*Unit Location(s)*	*Synchronizing on write*	*Determinate*
HPSTATE	0	TLU	Y	Y
HTSTATE	1	TLU	Y	Y
HINTP	3	TLU	Y	Y
HTBA	5	TLU	Y	Y
VER	6	TLU	Y	Y
HSTICK_COMPARE	31	TLU	Y	Y

11.1.3 Privileged Registers

TABLE 11-3 Privileged Register Locations

Register Name	*PR address*	*Unit Location(s)*	*Synchronizing on write*	*Determinate*
TPC	0	TLU	Y	Y
TNPC	1	TLU	Y	Y
TSTATE	2	TLU	Y	Y
TT	3	TLU	Y	Y
TICK	4	TLU	Y	Y
TBA	5	TLU	Y	Y
PSTATE	6	TLU	Y	Y
TL	7	TLU	Y	Y
PIL	8	TLU	Y	Y
CWP	9	EXU	Y	Y for reads/N for writes
CANSAVE	10	EXU	Y	Y
CANRESTORE	11	EXU	Y	Y
CLEANWIN	12	EXU	Y	Y
OTHERWIN	13	EXU	Y	Y
WSTATE	14	EXU	Y	Y
GL	16	TLU	Y	Y

11.1.4 ASI Registers

TABLE 11-4 OpenSPARC T2 ASI Register Locations

Register Name	*ASI/address*	*Unit Location(s)*	*Synchronizing on write*	*Determinate*
ASI_SCRATCHPAD	0x20/*	MMU	N	Y
ASI_PRIMARY_CONTEXT0	0x21/0x08	MMU, IFU, LSU	Y	Y
ASI_SECONDARY_CONTEXT0	0x21/0x10	MMU, LSU	Y	Y
ASI_PRIMARY_CONTEXT1	0x21/0x108	MMU, IFU, LSU	Y	Y
ASI_SECONDARY_CONTEXT1	0x21/0x110	MMU, LSU	Y	Y
ASI_QUEUE	0x25/*	TLU	Y	Y
ASI_CMP	0x41/*	off-core	Y	Y
ASI_INST_MASK_REG	0x42/0x8	EXU	Y	Y

TABLE 11-4 OpenSPARC T2 ASI Register Locations *(Continued)*

ASI_LSU_DIAG_REG	0x42/0x10	LSU	Y	Y
ASI_ERROR_INJECT_REG	0x43	LSU	Y	Y
ASI_LSU_CONTROL_REG	0x45/0x0	LSU	Y	Y
ASI_DECR	0x45/0x08	TLU	Y	Y
ASI_RST_VEC_MASK	0x45/0x18	off-core	Y	Y
ASI_DCACHE_DATA	0x46	LSU	Y	N
ASI_DCACHE_TAG	0x47	LSU	Y	N
ASI_IRF_ECC_REG	0x48	EXU	N/A	N
ASI_FRF_ECC_REG	0x49	FGU	N/A	N
ASI_STB_ACCESS	0x4A	LSU	N/A	N
ASI_(ERROR REGS)	0x4C	TLU	Y	Y
ASI_SPARC_PWR_MGMT	0x4E	LSU	Y	Y
ASI_HYP_SCRATCHPAD	0x4F/*	MMU	N	Y
ASI_ITSB_TAG_TARGET	0x50/0x0	MMU	Y	Y
ASI_ISFSR	0x50/0x18	TLU	Y	Y
ASI_ITLB_TAG_ACCESS	0x50/0x30	MMU	Y	Y
ASI_IMMU_VA_WATCHPOINT	0x50/0x38	TLU	Y	Y
ASI_MRA_ACCESS	0x51	MMU	Y	N
ASI_MMU_REAL_RANGE_0	0x52/0x108	MMU	Y	Y
ASI_MMU_REAL_RANGE_1	0x52/0x110	MMU	Y	Y
ASI_MMU_REAL_RANGE_2	0x52/0x118	MMU	Y	Y
ASI_MMU_REAL_RANGE_3	0x52/0x120	MMU	Y	Y
ASI_MMU_PHYSICAL_OFFSET_0	0x52/0x208	MMU	Y	Y
ASI_MMU_PHYSICAL_OFFSET_1	0x52/0x210	MMU	Y	Y
ASI_MMU_PHYSICAL_OFFSET_2	0x52/0x218	MMU	Y	Y
ASI_MMU_PHYSICAL_OFFSET_3	0x52/0x220	MMU	Y	Y
ASI_ITLB_PROBE	0x53	IFU	N/A	N
ASI_ITLB_DATA_IN_REG	0x54/0x0	MMU	Y	Y
ASI_MMU_*	0x54/0x10-98	MMU	Y	Y
ASI_ITLB_DATA_ACCESS_REG	0x55/0x0-0x1F8	IFU	Y	N
ASI_ITLB_TAG_READ_REG	0x56/0x0-0x1F8	IFU	N/A	N
ASI_IMMU_DEMAP	0x57/0x0	MMU	Y	Y
ASI_DTSB_TAG_TARGET	0x58/0x0	MMU	Y	Y
ASI_DSFSR	0x58/0x18	TLU	Y	Y
ASI_DSFAR	0x58/0x20	TLU	-	Y
ASI_DTLB_TAG_ACCESS	0x58/0x30	MMU	Y	Y
ASI_DMMU_WATCHPOINT	0x58/0x38	LSU	Y	Y
ASI_HWTW_CONFIG	0x58/0x40	MMU	Y	Y

TABLE 11-4 OpenSPARC T2 ASI Register Locations *(Continued)*

ASI_PARTITION_ID	0x58/0x80	MMU, LSU, IFU	Y	Y
ASI_SCRATCHPAD_ACCESS	0x59	MMU	Y	N
ASI_TICK_ACCESS	0x5A	TLU	Y	N
ASI_TSA_ACCESS	0x5B	TLU	Y	N
ASI_DTLB_DATA_IN_REG	0x5C	MMU	Y	Y
ASI_DTLB_DATA_ACCESS_REG	0x5D/0x0-0x1F8	LSU	Y	Y
ASI_DTLB_TAG_READ_REG	0x5E/0x0-0x1F8	LSU	N/A	Y
ASI_DMMU_DEMAP	0x5F/0x0	MMU	Y	Y
CMP_CORE_INTR_ID	0x63/0x0	LSU	N/A	Y
CMP_CORE_ID	0x63/0x10	LSU	N/A	Y
ASI_ICACHE_INSTR	0x66	IFU	Y	N
ASI_ICACHE_TAG	0x67	IFU	Y	N
ASI_INTR_RECEIVE	0x72	TLU	Y	Y
ASI_INTR_W	0x73	off-core	Y	Y
ASIINTR_R	0x74	TLU	N/A	Y

11.2 ASI Accesses

The OpenSPARC T2 core contains two ASI rings, both of which begin and terminate at the LSU. The “fast” ring connects only to the TLU and MMU to keep latency low. The “local” ring connects to all other units in the core (PKU and DEC do not contain any special purpose registers are are therefore not connected to either ring.) Access to ASI registers which are not located in the core (e.g., in the TCU or NCU) is done via the crossbar by mapping the access to I/O space (see below).

All RDASR/WRASR, RDPR/WRPR, RDHPR/WRHPR or Load/Store Alternate instructions discussed in this chapter are generically referred to as ASI load/store accesses. The LSU is the logical root of the ASI ring; all ASI accesses are sent to the LSU. The LSU is the entry and exit point from the ring for all requests. Only the LSU can initiate ring accesses.

There are two cases of ASI accesses: synchronizing and non-synchronizing. Synchronizing ASI accesses require that all subsequent instructions observe any architectural changes generated by the ASI access. The LSU determines whether or not an ASI access is synchronized.

The core treats load ASI operations as load misses because the data return time exceeds the load hit pipe.

(Registers on the fast ring have significantly lower latency than those on the local ring.)

The LSU signals a LSU synchronization event for an ASI load access during the B pipeline stage. A LSU synchronization event flushes the relevant thread, transitions the relevant thread into the WAIT state at pick, and refetches the load's NPC. The LSU decodes the ASI destination address, accesses the appropriate ring, returns data to the core via the W2 port of the IRF, and signals complete to PKU.

ASI stores are treated as normal stores in the sense that they are inserted in the STB and processed in order. The LSU decodes the store's target address and directs the data and address to the proper ring. A non-synchronizing store completes normally once the packet completes it's trip around the ring. A synchronizing store causes a LSU synchronization event when it reaches the B pipeline stage. The LSU synchronization flushes all instructions subsequent to the synchronizing store, causes pick to transition the relevant thread to the WAIT state and refetches the synchronizing store's NPC. When the synchronizing store completes, it signals a trap synchronization to the TLU. A trap synchronization flushes the relevant thread which releases the WAIT state at pick and refetches the synchronizing store's NPC.

11.3 ASI Ring Operation

11.3.1 Fast and local rings

The fast and local rings run within a physical core between the units which have ASI registers. Each unit has a ring node which performs the local access (read or write). The LSU decodes all ASI addresses in order to determine:

- if the ASI access is synchronizing or not
- if the access latency is determinate or not

The local ring latency is about 21 clock cycles. The fast ring latency is 4 clock cycles.

The ring is designed:

- to allow simultaneous determinate ASI accesses by different threads to the local ring
- to allow up to one indeterminate ASI access in parallel with other determinate ASI accesses
- to pipeline non-synchronizing determinate store requests within a thread

At a given ring node, operations either have a predictable latency, or an unpredictable (or long) latency. Some nodes may not have operations with unpredictable latency. The operations with predictable latency may not have the same latency. For example, some ASI registers on a node may be accessed faster than

other registers on that node. The ring protocol requires that each node enforce the maximum of the predictable latencies for all determinate operations. The total number of pipeline stages around the ring, termed the ring latency, is then the sum of the node-to-node transmission latency (typically 1 cycle), and the maximum predictable internal latency of each node. The maximum internal latency of each node is fixed at design time.

Scheduling for the local ring is as follows:

- the LSU generates an ASI request and inserts it at the head of the ring pipeline
- the request travels around the ring in a fixed number of cycles for determinate accesses and an arbitrary number of cycles for indeterminate accesses
- the LSU only allows one indeterminate load and one indeterminate store into the ring at any given time
- each ring node handles an determinate ASI access that is not targeted for the node with a latency equal to the maximum fixed latency of the node
- each ring node handles an determinate ASI access that is targeted for the node with a latency equal to the maximum fixed latency of the node
- each ring node handles an indeterminate ASI access that is targeted for the node in an arbitrary number of cycles. Each node holds the result of a indeterminate access until a hole is found on the ring to insert the result.
- a hole is defined as 2 consecutive idle cycles on the local ring. The LSU ensures a hole on the ring at least once every 16 cycles.
- the access is returned by the local ring to the LSU and processed appropriately

The ring is 65 bits wide. All ASI requests placed on the ring take two consecutive ring cycles. The address and control data are transmitted in the first ring cycle. The data for the access on the second consecutive ring cycle. For loads, second slot eventually holds the load data, although it is initially empty. For stores, this data is the store data to write into the target ASI register.

TABLE 11-5 Format of ASI Ring Control Packet

Bit(s)	*Field*	*Description*
64	Ctl/data	1=control packet, 0=data packet
63	valid	Along with [64] indicates a valid control packet
62	ack	Nodes set ack bit when they respond to a request
61:60	type	00=ASI, 01=ASR, 10=PR, 11=HPR
59	Rd/wrx	1=read/load, 0=write/store
58:56	Thread ID	
55:48	register	8 bit ASI value or ASR/PR/HPR register number
47:0	address	Virtual address for ASI accesses (N/A for ASR/PR/HPR)

When an ASI request arrives at a node, the node checks to see if it owns that request. If the node owns the request, it decodes the address, performs the access, sets the ack bit of the request, and places the appropriate data if required on the ring. If the node does not own the request, it passes the original ASI request through its internal node pipeline and places it unmodified on its node output register.

Indeterminate accesses are performed by the node in an arbitrary number of cycles. For a given ASI access, this latency may be known or unknown. Eventually, the appropriate node completes the indeterminate access. When complete, the node holds the result of the access until a hole can be found on the node's output register. The LSU ensures a hole on the ring at least once every 16 cycles. This ensures that an indeterminate ASI access returns to the LSU via the ring in the presence of determinate ASI accesses.

The LSU dequeues ASI reponses off the return path of the rings. For a load, the thread ID and the data are used to complete the load operation. For a store, the LSU generates a trap synchronization for the thread ID of the request if the internal sync state is set. No action other than dequeuing from the STB is required for a store if the sync bit is not set.

The order of units on the local ring is:

LSU

IFU

EXU0

PMU

FGU

EXU1

Each node contains a 64-bit register to store the incoming packet and another register to store the outgoing packet in addition to any internal pipelining flops.

11.3.2 Off-Core ASI Access

Global ASI accesses are handled via the cache crossbar (ccx) in conjunction with the NCU. Only the SPARC cores can initiate ASI accesses. Off-core ASI accesses are mapped to a physical I/O address and sent to the NCU as a load or store. The NCU will forward requests to other units as necessary. The ASI access is mapped to an I/O address as follows

TABLE 11-6 Format of I/O Mapped ASI Address

PA bits	*Field Description*
39:32	8'h90 (I/O region identifier)
31:29	Cpuid (of initiating core)
28:26	Thread id
25:18	8 bit ASI value
17:0	VA[17:0]

CHAPTER 12

Reset

This chapter describes the OpenSPARC T2 reset philosophy and operation.

Similar to previous SPARC processors, OpenSPARC T2 provides several flavors of resets. Resets can be activated as:

- a side-effect of an internal processor or system error, related either to instruction execution or an external event such as failure of a system component
- a result of explicit instruction execution (e.g., SIR)
- a result of a processor write to an ASI register which generates a reset
- a command over an external bus, such as the system bus or the JTAG interface to the Test Control Unit (TCU)
- a result of activating a pin on the OpenSPARC T2 chip

Some resets are local to a given physical core, or affect only one thread (CMP core). Other resets affect all threads.

(In this section, “core” means CMP core unless otherwise noted. “Physical core” means a SPARC processor which includes all of its threads).

of a given SPARC core, or all physical cores. Each of these capabilities is described below.

A reset is usually raised in response to a catastrophic event. Depending upon the event, it may not be possible for a core or for the entire chip to continue execution without a full reset (POR).

12.1 OpenSPARC T2 Resets

OpenSPARC T2 provides the following resets:

POR (Power-on-reset)

WMR (Warm reset)

XIR (Externally-initiated reset)

WDR (Watchdog reset)

SIR (Software-initiated reset)

The table below summarizes the effects of each OpenSPARC T2 reset.

TABLE 12-1 Effects of OpenSPARC T2 Resets

Reset	*Latches/Flops*	*Register Files*	*Arrays*	*Error Registers*	*Externally Initiated*	*Internally Initiated*	*Affects all threads of a physical core*
POR	Cleared	Cleared	Cleared	Cleared	Yes	No	Yes
WMR	Cleared, with limited updates to architected machine state	Unchanged	Unchanged	Unchanged	Yes	No	Yes
XIR	Trap-cleared	Unchanged	Unchanged	Unchanged	Yes	No	Function of CMP_XIR_STEERING
WDR	Trap-cleared	Unchanged	Unchanged	Unchanged	No	Yes	No
SIR	Trap-cleared	Unchanged	Unchanged	Unchanged	No	Yes	No

Terms in TABLE 12-1 are defined as follows:

cleared means set to 0 or the appropriate logical value as defined by POR. In OpenSPARC T2, clearing is accomplished via a full scan reset.

trap-cleared means only latches and flops which contain speculative pipeline state are cleared. Scan-reset of state is not performed. When the trap is taken, all pipeline state which relates to any speculative instructions is reset. State which relates to instructions which have been completed is not reset. In the case of CR, certain architected state elements may be updated, namely those which were defined to be updated at the next chip reset.

unchanged means no change is made to the pre-reset state, except where necessary to cause the machine to take the reset and to be in an architecturally compliant reset state as a result of taking the reset.

externally initiated means that the reset can be activated by an external source, either via a dedicated pin or bus transaction.

internally initiated means that the reset can be activated by an internal source, either as a direct effect of instruction execution (SIR), or as a side-effect of executing an instruction which resulted in a reset (taking a trap when TL==MAXTL), or as a result of a hyper-privileged instruction writing to a reset ASI register which explicitly causes the reset.

Power-on-reset (POR), also known as "hard" reset, is activated when the chip is first powered-up and power and clocks have stabilized. A hard reset completely erases the current state of the machine and initializes all on-chip flops, latches, register files, and memory arrays such as TLB's and caches to a known good state. Assuming the chip is working properly, a hard reset is guaranteed to put each processor in a consistent state where it can begin to fetch and execute instructions. Although called POR, the clearing of all machine state does not require power cycling. In OpenSPARC T2, the Test Control Unit (TCU) controls the scanning and reset of state elements and initialization of arrays. POR is initiated via an external pin. Following the state initialization process, the TCU instructs the machine (via the Trap Unit) to begin fetching and executing instructions at the RSTVaddr || 0x20. OpenSPARC T2 follows the CMP spec. The default POR state is for all available cores to be enabled and the lowest-numbered available core to be running. These values may be changed by the system controller, if present, during reset. These values take effect upon the deactivation or completion of POR. Caches are disabled following POR.

Warm reset (WMR), also known as "soft" reset, only partially clears OpenSPARC T2 state before branching to the new trap address and executing instructions under the new machine state. A soft reset has been used in previous SPARC processors to synchronize updating of registers which control clock ratios for the bus and memory interfaces. It is also defined as the synchronization point for disabling or enabling CMP cores. Updates to clock ratio and core enable registers do not take effect until after the next chip reset. Chip reset does not reset error status registers or clear on-chip arrays. It performs a limited clear of pipeline flops and state machines (though the vast majority of flops and state machines are cleared, only warm-reset-protected state is not cleared). Architected machine state is only updated in a limited way - for example, integer and floating-point registers are not reset, but the TICK, STICK and associated comparison registers are. As for POR, the default for WMR is that all available cores are enabled and the lowest-numbered available core is running (unparked). These values may be changed by the SC, if present, during chip reset. The new values take effect upon the deactivation or completion of WMR. Depending upon the error state of the chip, it may not be possible for the chip to continue executing instructions.

Externally-initiated reset (XIR) is a SPARC V9-defined trap. An XIR may be generated externally to OpenSPARC T2 via a chip pin. XIR does not reset any machine state, other than internal pipeline state required to cause a OpenSPARC T2 core to take a trap and other than the V9-required side-effects of state updates for an XIR trap. An XIR may be routed to all cores (threads) or a subset of them based upon the contents of the CMP ASI_XIR_STEERING register. Following recognition of an XIR, instruction fetching occurs at RSTVaddr || 0x60.

Watchdog reset (WDR) is a V9-defined trap. WDR can be initiated via an event (such as taking a trap when TL == MAXTL) which causes an entry into the V9 error state - the processor immediately generates a watchdog reset trap to take the core to RED_state. On OpenSPARC T2, a WDR also can result from a fatal error condition detected by on-chip error logic. A WDR only affects the strand which created it. When a WDR is recognized, instruction fetching begins at RSTVaddr || 0x40.

Software-initiated reset (SIR) occurs when privileged software on a thread executes the SIR instruction. SIR only affects the core which executed the SIR instruction. When an SIR is recognized, instruction fetching begins at RSTVaddr || 0x80.

All resets place the processor in RED_state.

12.2 Reset Priority

If multiple resets occur at the same time, resets on OpenSPARC T2 are prioritized in the following order:

1. POR
2. WMR
3. XIR
4. WDR
5. SIR

Only POR and WMR require scan-flushing of latches and flops. Additionally, POR initializes on-chip arrays.

12.3 RED_state

RED_state is entered when any of the above resets occur. RED_state is indicated when PSTATE.RED = 1. However, setting PSTATE.RED=1 via software does not result in a reset.

In RED_state, the I-TLB, the D-TLB and the MMU MSA and MCM are disabled. Address translation is also disabled; addresses are interpreted as physical addresses. Bits 63:47 of the address are ignored. RED_state does not affect the enabling or disabling of the caches.

12.4 Reset Values

See the PRM for the effects of the various resets on architecturally visible registers and machine state.

CHAPTER 13

Debug

This chapter has been superceded by the PRM Debug chapter.

CHAPTER 14

Power Management

OpenSPARC T2's power management support consists of two parts. Hardware power management uses clock gating within functional units to reduce power consumed by flops, latches, and static arrays. Since the OpenSPARC T2 core is static, there is no dynamic logic to be power-managed. Hardware power management can be enabled by software.

At a higher level, power estimation via an external chip agent can provide a means for software or a system agent to limit consumption. If hardware power management is enabled, and estimated power consumption still exceeds a threshold, software can park cores, or identify and stall high power processes to gracefully or selectively reduce power consumption.

This chapter outlines OpenSPARC T2's power management features.

14.1 Clock Distribution

OpenSPARC T2's clock distribution scheme eases local power control of flops and latches, both in datapaths and control blocks. The main PLL clock is fed to L2 clock headers, which are gridded to reduce skew. These outputs feed L1 clock headers, which in turn feed a row of datapath latches/flops or arrays. The L1 clock header outputs are not gridded. Each L1 clock header has an enable signal which controls clock generation to the set of latches/flops it feeds.

14.2 Functional Unit Clock Gating

Within a functional unit, the number of clock gating signals is a function of:

whether the flops are in control or datapath flops

timing restrictions on generating the enable

within a control block, there can be no more than 5 unique clock enables

within a datapath flop, all bits are fed from the same clock (and so require the same enable)

The OpenSPARC T2 core contains approximately 52K flops. Of those about 35K are in DP blocks, and about 85% have clock enables. Of the remaining 17K flops in control blocks, about 62% are gated.

Clock gating is disabled at POR. It can be selectively enabled within various functional units by writing a '1' to associated bit positions of the ASI_SPARC_PWR_MGMT register. See the PRM for details.

CHAPTER 15

Performance Monitors

This chapter describes the OpenSPARC T2 performance monitors. Goals of the performance monitoring capability are:

Enable data collection to develop accurate modeling for OpenSPARC T2 and future highly threaded processors

Enable debug of performance issues

Minimize hardware cost consistent with the above objectives

15.1 Overview

The PMU consists of the PCR and PIC registers for each thread. It takes in events from the rest of the core and, and, based upon the configuration of the PCR registers for each thread, optionally increments a PIC, sets an overflow bit if the PIC is within range, and indicates a trap request to the TLU.

To save area, all threads share two 32-bit adders, one for the PICH and one for the PICL. This means that a given thread only has access to the adders every eighth cycle. In turn, each PIC has an associated 4-bit accumulator, which increments each cycle an event occurred for that PIC. When the thread is selected, each of its two PICs and their corresponding accumulators are summed together in their corresponding 32-bit adder.

To save power, the PMU is clock-gated. It wakes up whenever an ASI read or write is active on the ASI ring, or when at least one counter from any thread is enabled for counting.

The PMU is divided into two parts, pmu_pct_ctl, and pmu_pdp_dp. The former contains the control logic and PCR registers while the latter contains the PIC registers and associated adder and muxing logic, and the PMU ASI ring interface.

See the PRM for details of the events the PMU counts.

15.2 Datapath (pmu_pdp_dp.sv)

The datapath consists of the following components:

A PICH and a PICL for each thread. Each thread's PICH and PICL are stored in one 64-bit register with 2 ports and a clock enable. One port contains the ASI ring input (flopped). The other port is used to update either PIC when it is active (e.g., either the PICH or PICL is configured for incrementing in the PCR), and it is that thread's turn to use the adder. This port is connected to the adder output.

A comparator for each PIC. There are 16 comparators. Each comparator determines whether the associated PIC is “within range” of overflowing. This is defined as being within [-16, -1] of wrapping from all 1's to zero.

A 64-bit, 8-port mux and a 64-bit flop. The muxes select one of the 8 PICs for incrementing. The mux output is flopped. Each 32-bit half of the flop output is connected to one of the input ports of the 2 32-bit adders.

Two 32-bit adders, implemented as a 4-bit adder for the least-significant bits, followed by a 12-bit incrementer, followed by a 16-bit incrementer. This is done to save area relative to a full 32-bit adder, and because there is no 28-bit incrementer.

An ASI ring data-in register. This 64-bit register takes the ASI ring input, except for the ctl/data bit which is flopped in the control block. Certain outputs of this register are fed to the control block for local usage.

An ASI ring output datapath. This consists of an 8:1 64-bit mux to select one of the 8 PICs. In turn this is muxed with PCR data from the control block, and fed to a 3-port 64-bit ASI output flop. The three ports are a) the ASI input flop output (used when bypassing ASI data when there is no PMU-related operation), b) the output of the PIC/PCR mux, selected when there is a non-excepting read of the PIC or PCR, and c) an exception output which is used when a user-level access of the PIC occurs and the PIC is marked privileged, or, a user-level access of the PCR occurs. The output of the flop is buffered, then drives the ASI ring to the next block in the Sparc core.

15.3 Control (pmu_pct_ctl)

The control block has several functions. First, it decodes and responds to any activity on the ASI ring which affects the PMU. This includes updating or reading the PICs in the datapath block as well as the PCRs, and possibly responding with an exception. The control block also registers all event inputs from the rest of the core, selects them appropriately for each counter, and increments the accumulator as required. It also interacts with the TLU to generate traps when a counter overflows, or is within range of overflowing an an event occurs. Finally, it also determines when to power-up and power-down the PMU.

15.3.1 ASI Ring

Timing for the ASI ring is as follows. At the end of a cycle, I, the ASI ring input data is flopped in the datapath and the CTL/NDATA signal is flopped in the control block. During the next cycle, I+1, the ASI operation is decoded. If it does not concern the PMU, the input flop is selected at the input to the output flop, bypassing the ASI input data to the output. If it does concern the PMU, the access type is determined and checked to see if an exception occurred. If so, the exception and ACK flags are selected at the input to the ASI ring flop. Otherwise, the ASI access is ACK'ed in the control packet, and the access type is flopped. The next cycle, I+2, if the access is a read, either the PCR or PIC for the thread is selected at the appropriate muxes and then muxed into the ASI output flop. Similarly, if the access is an ASI write, the write data is currently in the ASI input flop, so it is selected at the input to the appropriate PIC or PCR, and the write data is passed unchanged onto the ASI output flop.

There is one case which is a hazard. It is possible that at the same cycle a given PIC is being selected for updating from the adder, an ASI write is occurring for that PIC. In this case, the PIC is updated as follows. Since the PIC increment pipeline is a 2-cycle operation (described below) there are actually 3 cases:

PIC ASI write and update at the same cycle. Here the ASI value takes precedence (the incremented value is overwritten).

PIC ASI write in cycle I and update in cycle I+1. In this case we also select the ASI value, since there is a 2-cycle PIC update pipeline, and we don't want to overwrite the ASI value with the (incremented) value of the previous PIC.

Update in cycle I and PIC write in cycle I+1. This works fine, we select the increment in cycle I and overwrite it with the ASI value in I+1.

15.3.1.1 Event Types and Event Pipeline

The events the PMU can monitor are divided into event groups. Each group is identified by a particular encoding of the PCR.SL[01] field. Within each event group, there is an 8-bit mask which can further select the type of event(s) to be monitored. The mask field may have multiple bits set; it acts as an AND-OR reduction on the associated event group to select sub-events in that group.

The events that the PMU can count can be divided into two types. Some events (those in event groups 2 and 3) are synchronous. The remaining events are asynchronous. A synchronous event is directly associated with the execution of an individual instruction, and therefore we count it "precisely". If the processor can take a SOFTINT(15) trap, then the tPC will contain the instruction which caused the counter overflow. If the associated instruction is flushed, then the event is not counted. Asynchronous events bear no relation to the execution of a particular instruction, or, can not be counted within the instruction pipeline. Thus there is no way for their counts to be cancelled.

This means that asynchronous events can be fed "directly" into the accumulation logic, while synchronous events must be piped along until they can not be flushed, then they can be counted. The pipeline mimics the actual instruction pipeline. The PMU starts with flops which register all inputs from other blocks (for timing reasons).

15.3.1.2 Synchronous Events

The synchronous events related to instructions are flopped at the end of the D stage, so the PMU contains flops from D:E, E:M, M:B, and B:W. The PMU also flops from W:W1 and W1:W2 for the following reason. Flushes can occur from DEC in E or M (flopped, and applied in M or B, respectively), and from the TLU in B or W (flopped, and applied in W or W1, respectively). If a flush occurs, the corresponding event is masked out. Thus, the last stage a flush can occur is W1, and the update of a given thread's PICs needs to wait at least until W1. In order to avoid a timing path related to selecting a thread's PICs and doing an add in one cycle, the PIC increment pipeline is split into 2 stages (W1 and W2). During W1, the "final" increment signals are generated by gating off any synchronous events with the trap flush signal from W, ORed into the asynchronous events, and incrementing the PIC accumulators. Also, the PICs are muxed between the threads and stored in a flop to be fed to the adder. Similarly, the PIC accumulators are also muxed and stored in a flop. During W2, the selected thread's PICs and accumulators are added together, and stored in the PICs at the end of the cycle. TABLE 15-1 below illustrates the PMU pipeline, where Op0 is an operation being counted by the PMU for a given thread. Op0 proceeds down the pipeline and is committed, so the PIC for the thread (PICH or PICL or both, depending) is incremented. Also note that the cycle when the PIC is updated need not correspond to the cycle immediately after Op0 accumulates; it is updated every 8^{th} cycle for a given thread.

TABLE 15-1 PMU pipeline for synchronous events – instruction Op0 being counted commits, Accumulator and PIC incremented

D / IRF	Op0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5	GenericOp6
E / FRF		Op0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5
M / D$ / FX1			Op0	GenericOp1	GenericOp2	GenericOp3	GenericOp4
B				Op0	GenericOp1	GenericOp2	GenericOp3
W					Op0 commit	GenericOp1	GenericOp2
W1						Accumulator++	GenericOp1
W2							PIC+= Accumulator

TABLE 15-2 below illustrates a case where LSU detects an exception in B on a load (Load1) which is being counted. The load is flushed by TLU in W1, so the accumulator is not incremented. Note that the PIC is still incremented by the value of the accumulator, since it may contain counter updates for prior instructions (e.g., Load0).

TABLE 15-2 PMU pipeline for synchronous events – accumulator not incremented for Load1 since instruction flushed

D / IRF	Load1	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5	GenericOp6 Flushed by IFU
E / FRF	Load0	Load1	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5 Flushed by IFU
M / D$ / FX1		Load0	Load1	GenericOp1	GenericOp2	GenericOp3	GenericOp4 Flushed by IFU
B			Load0	Load1 LSU reports Load1 exception to TLU	GenericOp1	GenericOp2	GenericOp3 Flushed by IFU

TABLE 15-2 PMU pipeline for synchronous events – accumulator not incremented for Load1 since instruction flushed *(Continued)*

W				Load0	EXU, LSU flush Load1 TLU broadcasts flush	GenericOp1 Flushed by TLU	GenericOp2 Flushed by IFU
W1					PMU increments Accumulator for Load0 (Accumulator++)	PMU flushes Load1, Accumulator+=0	
W2							PIC+= Accumulator

15.3.1.3 Asynchronous events

These events include crossbar and "all idle" events. In order to count these events, the PMU first muxes the asynchronous events based upon the event group and the event masks (there are 16 copies of this logic, one for each PIC). In the same cycle, W1, "pipeh_async[7:0]" and "pipel_async[7:0]" are generated, ORed into the synchronous signals, and finally added to the accumulators. TABLE 15-3 Illustrates this case.

TABLE 15-3 PMU pipeline for asynchronous events – PIC incremented, independent of instruction pipeline activity

D / IRF	ELOp0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5	GenericOp6 Flushed by IFU
E / FRF		ELOp0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5 Flushed by IFU
M / D$ / FX1			ELOp0	GenericOp1	GenericOp2	GenericOp3	GenericOp4 Flushed by IFU
B				ELOp0 EXU, LSU report exception to TLU	GenericOp1	GenericOp2	GenericOp3 Flushed by IFU

TABLE 15-3 PMU pipeline for asynchronous events – PIC incremented, independent of instruction pipeline activity *(Continued)*

W					EXU, LSU flush ELOp0 TLU broadcasts flush	GenericOp1 Flushed by TLU	GenericOp2 Flushed by IFU
W1					PMU accepts event, Accumulator++		
W2						PIC+= Accumulator	

15.4 Trap Pipeline

The trap pipeline is similar to the counter increment pipeline, but there are complications. In the normal trap pipeline, a unit (e.g., EXU, LSU) informs the TLU of a trap request in the B stage. The requesting unit self-flushes the instruction in the W stage. The same cycle, W, the TLU broadcasts a flush signal, which all units use to flush the next instruction the next cycle, W1. Also during W, the IFU broadcasts a flush signal to flush any instructions from that thread currently in the B, M, E, or D stages.

However, in the case of the PMU, we want precise interrupts. An instruction which is “within range” and will (logically) cause a counter to overflow should be flushed and the TPC should point to the PC of that instruction. No architectural effects from that instruction should occur. The PMU does not self-flush the instruction, however, as it is used to increment the counter and/or set the PCR.OV bit. If the PMU waits until B to send the exception to the TLU, the TLU will not broadcast the flush signal until W, and the other units (e.g., LSU/EXU) will flush the instruction during W+1, which is too late (the instruction has already updated architectural state in W). Thus the PMU must request the flush one cycle earlier (M). The TLU broadcasts the flush during B, and the units flush the instruction during W, before updating architectural state.

TABLE 15-4 PMU pipeline for synchronous events – instruction commits, but is within range; instruction causes a trap request at M and PIC is incremented at W2

D / IRF	Op0	GenericOp1	GenericOp2	GenericOp3	GenericOp4	GenericOp5 Flushed by IFU	
E / FRF		Op0	GenericOp1	GenericOp2	GenericOp3	GenericOp4 Flushed by IFU	
M / D$ / FX1			Op0 PMU broadcasts flush request to TLU	GenericOp1	GenericOp2	GenericOp3 Flushed by IFU	
B				Op0 TLU broadcasts flush	GenericOp1	GenericOp2 Flushed by IFU	
W					Op0 Flushed by TLU (except for PMU)	GenericOp1 Flushed by IFU	
W1						Accumulator++	
W2							PIC+= Accumulator

There is an additional complication for the data cache group events. The LSU can not report cache/data TLB miss until late in M; this is too late for the PMU to request a flush. Instead, during M, the PMU tells the LSU if a thread's counter is counting data cache/data TLB events and is “within range” of overflowing. If the LSU executes an instruction from that thread which causes a data cache/data TLB miss, these events cause an “lsu_sync” for that thread; all subsequent instructions are flushed. The TLU will flush all units as necessary. The TLU will **not** flush the PMU.

An additional complication arises from the fact that the PMU traps are disrupting. Since the traps are disrupting, the trap request needs to be held until the trap is actually taken. Either the PMU or the TLU can hold the trap request, in principle. Holding the trap request in the TLU proved problematic so the PMU holds the trap request. Thus it is piped from M to B to W to W1 to a “trap_hold” state. Prior to the “trap hold” state, the trap request can be flushed by the TLU (or IFU) if there is a higher priority exception on this instruction, or any exception on a previous

instruction from this thread. Once it gets to the "trap hold" state, it can only be flushed by the TLU actually taking this trap. In this case the request is cleared and the TLU takes the disrupting trap. ***Unfortunately this means that even for the "precise" events, if PIL is set appropriately, or PSTATE.IE is 0, or HPSTATE.HPRIV is set, the TPC captured will not point to an instruction of the class which created the trap request.***

A final complication arises from the lack of area. This leads to imprecision regarding the overflow of a counter and the associated trap. There are two sources of the imprecision. The first is the mass-balance logic required to generate the trap request. Qualifying all threads in the pipeline with a precise counter value requires a lot of logic. Consider the case where there is only one instruction of a given type (which is being counted) from a thread in the pipe. In that case it is easy to qualify the fact that the counter is within 1 of overflowing to generate a trap request. Now assume there are 2 instructions of a given type from the thread in the pipeline. In this case we must consider all the cases where the counter is -2. The first instruction should not take a trap, and the 2^{nd} one should. This qualification has to be done for all pipe stages up between E and W+1 (5 stages). The second source has to do with the fact that the 8 counters are shared among one adder. Thus, looking solely at the counter is not accurate, since the counter can be off by up to 8. Summing the two gives an imprecision of ~13. It is easiest to compare if the counter is within 16 of overflowing.

15.5 Power Management

Within the control block there are two basic groups of flops. The first is related to the ASI interface. The second is related to events and the event pipeline.

Powerup/down of the ASI ring is accomplished via an input signal from the LSU, which is the originator of all local ASI transactions. Specifically, the signal lsu_asi_clken is flopped, then the output pmu_asi_clken is used to control the clock enables for the following groups of flops. Note that the LSU also provides an "enable" signal, lsu_pmu_pmen, to override the lsu_asi_clken power management signal. This signal is flopped and the output is inverted and Ored with all other power management signals.

The flop which registers the priv/hpriv state indications for each thread from the TLU. These 16 flops could be kept running all the time, but it is necessary only to register the current state when 1) any counter is enabled, or 2) an ASI ring operation for the PMU occurs. The former is required since each PIC can be configured to count events in any combination of HPRIV, PRIV, and USER states. The latter is required since reads and writes of the PCR are privileged, and reads and writes of the PICs are optionally privileged (determined via the PCR.PRIV bit).

The group of flops which record whether an ASI read or write of the PCR or PIC has just been decoded and also pipe the CTL/NDATA ASI ring bus signal along.

The ASI input and output registers in the datapath block.

The second group of flops in the control block are power-managed via two signals. The first group, the PCR.OV bits, are power managed by the OR of a write to the PCR (to update the bits from the ASI ring), a read to the PCR (which resets the OV bits), or the PMU being busy. The PMU is busy whenever there is at least one thread whose PCR has at least one of the UT, HT, or ST bits set. This condition also power manages the remainder of the flops in the control block.

There is a corner case regarding the wakeup of the PMU and the updating of a counter, if the counter has just been written to. This bug was exposed in Metrax 85176. The problem arises from the fact that there is a two-cycle pipeline for updating a counter. In this case, the PMU was not enabled. The PIC update pipeline had frozen with the value of PIC0 in the flop feeding the adder (e.g., PIC0 was in the W2 stage). An ASI write to PIC0 then occurred. (Thus the flop contained a stale value). Then the PCR was written to, enabling counting. When the PMU woke up, it wrote the sum of the stale value of PIC0 and the accumulator into PIC0, overwriting the ASI PIC value. The implemented solution is to set a flag if the PIC contained in pic_std_w2 is written to while the PMU is asleep. If so, upon wakeup, the PIC update is disabled.

Each PIC is individually power-managed. The PIC clks are enabled whenever they are being written, or, it is their turn for being incremented, and they are enabled to count in HT, UT, or ST.