

PULSA: PRINCETON ULTRA SCALABLE ARCHITECTURE

YAOSHENG FU, YANQI ZHOU, TRI NGUYEN, MICHAEL McKEOWN, AND DAVID
WENTZLAFF

DISCLAIMER: THIS DOCUMENT IS BASED ON THE OPENSARC T1
MICROARCHITECTURE SPECIFICATION [1]. WE HAVE MODIFIED IT TO REFLECT
OUR DESIGN.

Contents

List of Tables	vi
List of Figures	vii
1 OpenSPARC T1PULSA Overview	1
1.1 Introducing the OpenSPARC T1 PULSA Processor	1
1.2 Functional Description	2
1.3 OpenSPARC T1 PULSA Components	4
1.3.1 SPARC Core	4
1.3.1.1 Instruction Fetch Unit	6
1.3.1.2 Execution Unit	6
1.3.1.3 Load/Store Unit	7
1.3.1.4 Floating-Point Frontend Unit	7
1.3.1.5 Trap Logic Unit	8
1.3.1.6 Stream Processing Unit	8
1.3.1.7 Execution Drafting Synchronization Logic	8
1.3.1.7.1 Stall Thread Synchronization Method	8
1.3.1.7.2 Random Thread Synchronization Method	9
1.3.2 CPU-Cache Crossbar	9
1.3.3 Floating-Point Unit	11
1.3.4 L2-Cache	11
1.3.5 DRAM Controller	13
1.3.6 I/O Bridge	13
1.3.7 J-Bus Interface	13
1.3.8 Serial System Interface	14
1.3.9 Electronic Fuse	14
2 SPARC Core	15
2.1 SPARC Core Overview and Terminology	16
2.2 SPARC Core I/O Signal List	19
2.3 Changes to SPARC Core in PULSA Processor	20
2.4 Instruction Fetch Unit	20
2.4.1 SPARC Core Pipeline	20
2.4.2 Instruction Fetch	23
2.4.3 Instruction Registers and Program Counter Registers	23
2.4.4 Level 1 Instruction Cache	24

2.4.5	I-Cache Fill Path	25
2.4.6	Alternate Space Identifier Accesses, I-Cache Line Invalidations, and Built-In Self-Test Accesses to the I-Cache	26
2.4.7	I-Cache Miss Path	26
2.4.8	Windowed Integer Register File	28
2.4.9	Instruction Table Lookaside Buffer	29
2.4.10	Thread Selection Policy	29
2.4.11	Thread States	30
2.4.12	Thread Scheduling	32
2.4.13	Rollback Mechanism	33
2.4.14	Instruction Decode	34
2.4.15	Instruction Fetch Unit Interrupt Handling	34
2.4.16	Error Checking and Logging	35
2.5	Execution Drafting Synchronization Logic	35
2.5.1	Stall Thread Synchronization Method	38
2.5.2	Random Thread Synchronization Method	38
2.6	Load Store Unit	40
2.6.1	LSU Pipeline	41
2.6.2	Data Flow	41
2.6.3	Level 1 Data Cache (D-Cache)	42
2.6.4	Data Translation Lookaside Buffer	43
2.6.5	Store Buffer	44
2.6.6	Load Miss Queue	44
2.6.7	Processor to Crossbar Interface Arbiter	45
2.6.8	Data Fill Queue	46
2.6.9	ASI Queue and Bypass Queue	46
2.6.10	Alternate Space Identifier Handling in the Load Store Unit . .	47
2.6.11	Support for Atomic Instructions (CAS, SWAP, LDSTUB) . .	47
2.6.12	Support for MEMBAR Instructions	47
2.6.13	Core-to-Core Interrupt Support	48
2.6.14	Flush Instruction Support	48
2.6.15	Prefetch Instruction Support	49
2.6.16	Floating-Point BLK-LD and BLK-ST Instructions Support . .	49
2.6.17	Integer BLK-INIT Loads and Stores Support	49
2.6.18	STRM Load and STRM Store Instruction Support	50
2.6.19	Test Access Port Controller Accesses and Forward Packets Support	50
2.6.20	SPARC Core Pipeline Flush Support	51
2.6.21	LSU Error Handling	51
2.7	L1.5 Data Cache	52
2.7.1	Write-back functionality	52
2.7.2	Way-map table for L1D cache coherence	53
2.7.3	Write-buffer	53
2.7.4	Handling L1I	54
2.7.5	Handling special loads and stores	54

2.7.5.1	Block loads/stores	54
2.7.5.2	Prefetch loads	54
2.7.5.3	Non-cacheable loads/stores	55
2.7.5.4	CAS/SWP/LOADSTUB	55
2.7.5.5	IPI packets	55
2.7.6	Cache coherence	55
2.7.7	Interfaces	55
2.7.8	Testing/debugging support	56
2.7.9	Implementation	56
2.7.9.1	Pipelined implementation	56
2.8	CPX-NOC Transducer	57
2.9	Execution Unit	58
2.10	Floating-Point Frontend Unit	61
2.10.1	Functional Description of the FFU	61
2.10.2	Floating-Point Register File	61
2.10.3	FFU Control (FFU_CTL)	62
2.10.4	FFU Data-Path (FFU_DP)	62
2.10.5	FFU VIS (FFU_DP)	62
2.11	Multiplier Unit	62
2.11.1	Functional Description of the MUL	62
2.12	Stream Processing Unit	62
2.12.1	ASI Registers for the SPU	63
2.12.2	Data Flow of Modular Arithmetic Operations	65
2.12.3	Modular Arithmetic Memory (MA Memory)	65
2.12.4	Modular Arithmetic Operations	67
2.13	Memory Management Unit	69
2.13.1	The Role of MMU in Virtualization	70
2.13.2	Data Flow in MMU	71
2.13.3	Structure of Translation Lookaside Buffer	71
2.13.4	MMU ASI Operations	73
2.13.5	Specifics on TLB Write Access	74
2.13.6	Specifics on TLB Read Access	75
2.13.7	Translation Lookaside Buffer Demap	75
2.13.8	TLB Auto-Demap Specifics	75
2.13.9	TLB Entry Replacement Algorithm	75
2.13.10	TSB Pointer Construction	76
2.14	Trap Logic Unit	76
2.14.1	Architecture Registers in the Trap Logic Unit	78
2.14.2	Trap Types	79
2.14.3	Trap Flow	81
2.14.4	Trap Program Counter Construction	83
2.14.5	Interrupts	83
2.14.6	Interrupt Flow	84
2.14.7	Interrupt Behavior and Interrupt Masking	87
2.14.8	Privilege Levels and States of a Thread	87

2.14.9	Trap Modes Transition	88
2.14.10	Thread States Transition	89
2.14.11	Content Construction for Processor State Registers	90
2.14.12	Trap Stack	91
2.14.13	Trap (Tcc) Instructions	92
2.14.14	Trap Level 0 Trap for Hypervisor	92
2.14.15	Performance Control Register and Performance Instrumentation Counter	92
2.15	Core Debug Features	94
2.15.1	Resetting a Thread	94
2.15.2	Interrupting a Thread	94
2.15.3	Shadow Scan	95
3	SPARC Uncore	99
3.1	CCX Transceiver	100
3.2	L1.5 Cache	100
3.2.1	L1 Design Summary	100
3.2.2	L1.5 Design Summary	101
3.2.3	L1.5 Design Elaboration	102
3.2.3.1	Optimizations elaborations	102
3.2.4	Interactions With Blocks in Uncore	102
3.2.4.1	I/O listing	103
3.2.5	Pipeline Design	105
3.2.5.1	Tenets of designing pipeline	105
3.2.5.2	Pipeline operation detail	110
4	L2 Cache	113
4.1	Overview	113
4.2	Architecture Description	113
4.3	Pipeline Flow	113
4.4	Instruction Operations	113
4.5	Special Accesses to L2	113
4.5.1	Diagnostic access to the data array	114
4.5.2	Diagnostic access to the directory array	115
4.5.3	Diagnostic access to the shared map cache (SMC)	115
4.5.4	Coherence flush on a specific cache line	116
4.5.5	Diagnostic access to the tag array	116
4.5.6	Flush the shared map cache (SMC)	117
4.5.7	Diagnostic access to the state array	117
4.5.8	Access to the coreid register	118
4.5.9	Access to the error status register	118
4.5.10	Access to the L2 control register	118
4.5.11	Access to the L2 access counter	119
4.5.12	Access to the L2 miss counter	119
4.5.13	Displacement line flush on a specific address	119

List of Tables

2.1	SPARC Core Terminology	18
2.2	SPARC Core I/O Signal List	19
2.3	Modular Arithmetic Operations	65
2.4	Error Handling Behavior	68
2.5	Supported OpenSPARC T1 PULSA Trap Types	80
2.6	Privilege Levels and Thread States	87
2.7	Interrupt Data Field - Reset	94
2.8	Interrupt Data Field - HW Int	94
2.9	SPARC Physical Core Shadow Scan Chain	97

List of Figures

1.1	OpenSPARC T1 PULSA Processor Block Diagram	3
1.2	SPARC Core Pipeline	5
1.3	CCX Block Diagram	10
2.1	SPARC Core Block Diagram	16
2.2	Physical Location of Functional Units on an OpenSPARC T1 SPARC Core	17
2.3	Virtualization of Software Layers	18
2.4	SPARC Core Pipeline and Support Structures	22
2.5	Frontend of the SPARC Core Pipeline	23
2.6	I-Cache Fill Path	25
2.7	I-Cache Miss Path	27
2.8	IARF and IWRF File Structure	28
2.9	Basic Transition of Non-Active States	30
2.10	Thread State Transition of an Active Thread	31
2.11	State Transition for a Thread in Speculative States	32
2.12	Rollback Mechanism Pipeline Graph	33
2.13	ESL Additions and Interactions with SPARC Core Front-End	36
2.14	Internal Structure of the ESL	37
2.15	RTSM LFSR	39
2.16	LSU Pipeline Graph	41
2.17	LSU Data Flow Concept	42
2.18	Execution Unit Diagram	58
2.19	Shifter Block Diagram	59
2.20	ALU Block Diagram	59
2.21	IDIV Block Diagram	60
2.22	Top-Level FFU Block Diagram	61
2.23	Multiplier (MUL) Block Diagram	63
2.24	Layout of MA ADDR Register Bit Fields	64
2.25	Data Flow of Modular Arithmetic Operations	65
2.26	State Transition Diagram Illustrating MA Operations	67
2.27	Multiply Function Result Generation Sequence Pipeline Diagram	69
2.28	MMU and TLBs Relationship	70
2.29	Virtualization Diagram	70

2.30	Translation Lookaside Buffer Structure	72
2.31	TLU Role With Respect to All Other Backlogs in a SPARC Core . .	77
2.32	Trap Flow Sequence	81
2.33	Trap Flow With Respect to the Hardware Blocks	82
2.34	Flow of Hardware and Vector Interrupts	84
2.35	Flow of Reset, Idle, or Resume Interrupts	85
2.36	Flow of Software and Timer Interrupts	86
2.37	Trap Modes Transition	88
2.38	Thread State Transition	89
2.39	PCR and PIC Layout	93
2.40	SPARC Shadow Scan Chain	95
2.41	Shadow Scan Snap Timing Diagram	96
3.1	Architecture of the Uncore	99
3.2	Pipeline diagram of the L1.5	110

Chapter 1

~~OpenSPARC T1~~**PULSA** Overview

This chapter contains the following topics:

- Section 1.1, Introducing the ~~OpenSPARC T1~~**PULSA** Processor, on page 1
- Section 1.2, Functional Description, on page 2
- Section 1.3, ~~OpenSPARC T1~~**PULSA** Components, on page 4

1.1 Introducing the ~~OpenSPARC T1~~**PULSA** Processor

~~The OpenSPARC T1 processor is the first chip multiprocessor that fully implements the Sun Throughput Computing Initiative. The OpenSPARC T1 processor is a highly integrated processor that implements the 64-bit SPARC V9 architecture. The PULSA processor is a scalable manycore processor that is extensible to multiple chips, creating a multichip manycore fabric, and implements the 64-bit SPARC V9 architecture. This processor targets commercial applications such as application servers and database servers.~~

~~The ~~OpenSPARC T1~~**PULSA** processor contains eight~~**40** SPARC® processor cores, which each have full hardware support for ~~four~~**two** threads. Each SPARC core has an instruction cache, a data cache, and a fully associative instruction and data translation lookaside buffers (TLB). The ~~eight~~**40** SPARC cores are connected through a ~~crossbar~~
~~to an on-chip unified level 2 cache (L2-cache)~~**2D mesh network-on-chip (NOC)** to a ~~distributed on-chip unified level 2 cache (L2-cache)~~ and 40 FPUs.

~~The four on-chip dynamic random access memory (DRAM) controllers directly interface to the double data rate-synchronous DRAM (DDR2 SDRAM). Additionally,~~

~~there is an on-chip J-Bus controller that provides an interconnect between the OpenSPARC T1 processor and the I/O subsystem.~~Dynamic random access memory (DRAM) controllers and the I/O subsystem are implemented in FPGA which communicates to the PULSA chip through a single off-chip interface (OCI).

The PULSA chip integrates several research ideas from the Princeton-Parallel research group. Each SPARC core implements Execution Drafting mechanisms, which deduplicates computation to improve energy efficiency. TODO: ADD RESEARCH IDEAS HERE. These are discussed in more detail in later sections.

1.2 Functional Description

The features of the ~~OpenSPARC T1~~PULSA processor include:

- ~~840~~ SPARC V9 CPU cores, with ~~42~~ threads per core, for a total of ~~3280~~ threads
- ~~132 Gbytes/sec crossbar~~6 64-bit NOCs interconnect for on-chip communication
- 16 Kbytes of primary (Level 1) instruction cache per CPU core
- 8 Kbytes of primary (Level 1) data cache per CPU core
- 8 Kbytes of intermediate (Level 1.5) writeback data cache per CPU core
- ~~3 Mbytes of secondary (Level 2) cache 4 way banked, 12 way associative shared by all CPU cores~~2.5 Mbytes of distributed secondary (Level 2) cache (64 Kbytes per core), 8-way(TODO) set associative - shared by all cores
- ~~4 DDR-II DRAM controllers 144-bit interface per channel, 25 GBytes/sec peak total bandwidth~~
- IEEE 754 compliant floating-point unit (FPU), ~~shared by all CPU cores~~one per core, although any core can use any FPU
- Execution Drafting mechanisms in each core for energy efficiency
- TODO: Other research ideas
- External interfaces:
 - ~~J-Bus interface (JBI) for I/O 2.56 Gbytes/sec peak bandwidth, 128-bit multiplexed address/data bus~~
 - ~~Serial system interface (SSI) for boot PROM~~
 - One bidirectional 64-bit off-chip interface to FPGA which handles DRAM controllers and I/O

Figure 1.1 shows a block diagram of the ~~OpenSPARC T1~~PULSA processor illustrating the various interfaces and integrated components of the chip.

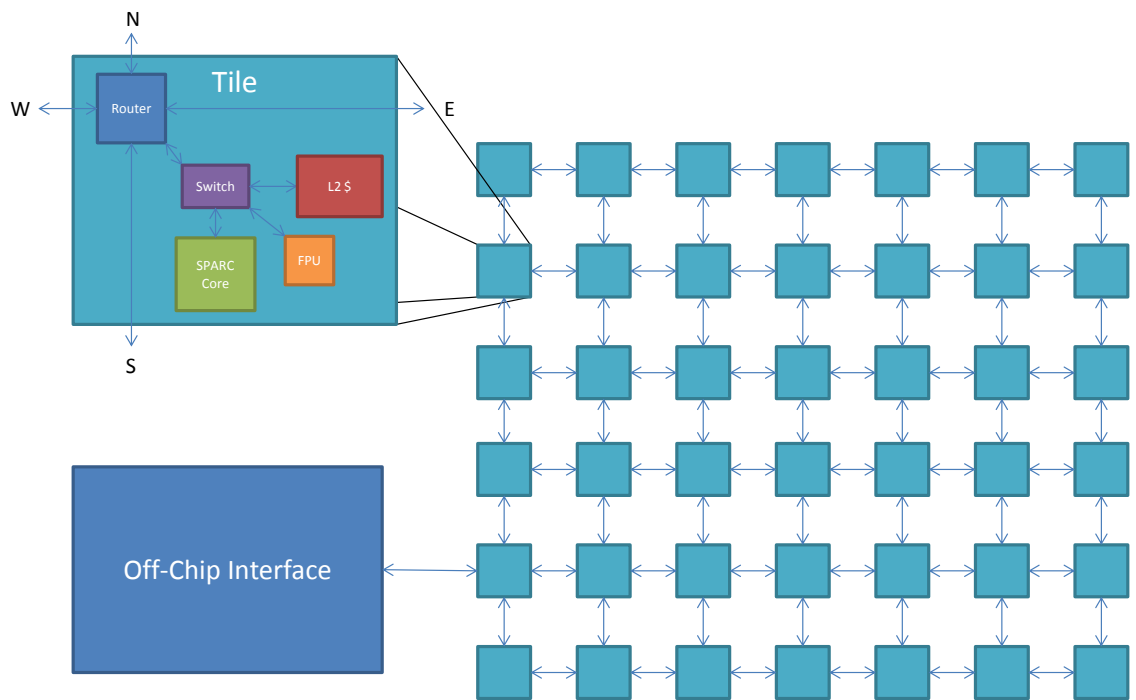


Figure 1.1: ~~OpenSPARC T1~~PULSA Processor Block Diagram

1.3 ~~OpenSPARC T1~~PULSA Components

This section provides further details about the ~~OpenSPARC T1~~PULSA components.

1.3.1 SPARC Core

Each SPARC core has hardware support for ~~four~~two threads. This support consists of a full register file (with eight register windows) per thread, with most of the address space identifiers (ASI), ancillary state registers (ASR), and privileged registers replicated per thread. The ~~four~~two threads share the instruction, the data caches, and the TLBs. Each instruction cache is 16 Kbytes with a 32-byte line size. The data caches are write through, 8 Kbytes, and have a 16-byte line size. The TLBs include an autodemap feature which enables the multiple threads to update the TLB without locking. ~~There is also a writeback data cache that is 8 Kbytes and has a 16-byte line size, which acts as a L1.5 cache.~~

Each SPARC core has single issue, six stage pipeline. These six stages are:

1. Fetch
2. Thread Selection
3. Decode
4. Execute
5. Memory
6. Write Back

Figure 1.2 shows the SPARC core pipeline used in the ~~OpenSPARC T1~~PULSA Processor.

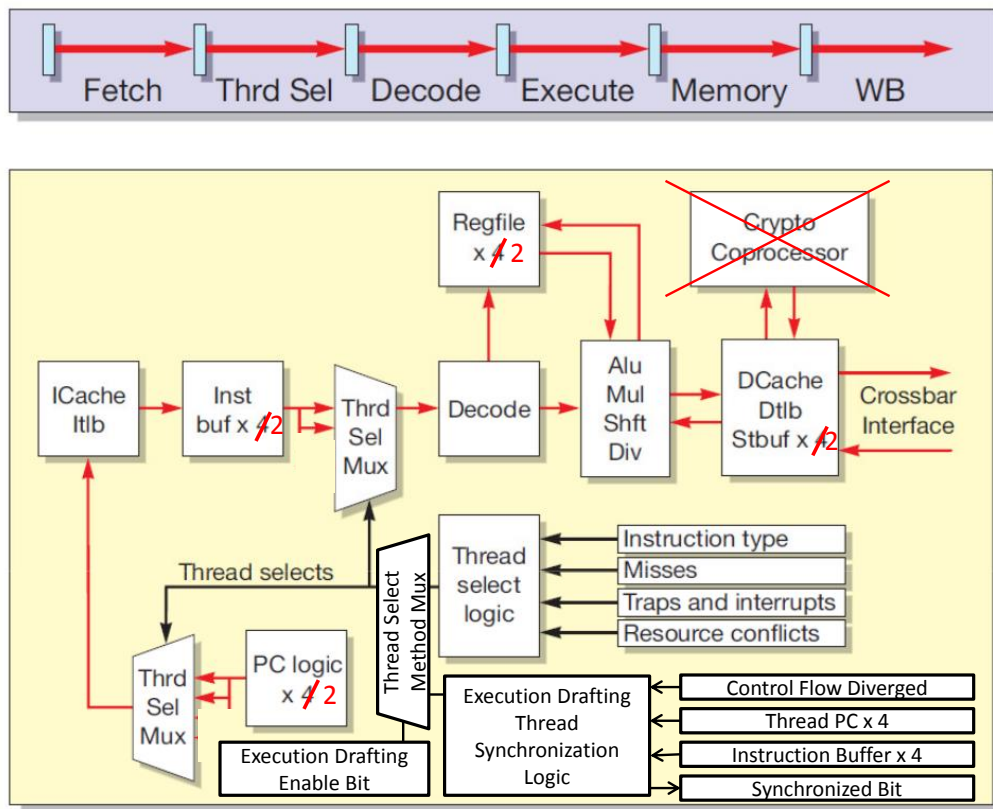


Figure 1.2: SPARC Core Pipeline

Each SPARC core has the following units:

1. Instruction fetch unit (IFU) includes the following pipeline stages: fetch, thread selection, and decode. The IFU also includes an instruction cache complex.
2. Execution unit (EXU) includes the execute stage of the pipeline.
3. Load/store unit (LSU) includes memory and writeback stages, and a data cache complex.
4. Trap logic unit (TLU) includes trap logic and trap program counters.
5. ~~Stream processing unit (SPU) is used for modular arithmetic functions for crypto.~~
6. Memory management unit (MMU).
7. Floating-point frontend unit (FFU) interfaces to the FPU.
8. Execution Drafting synchronization logic (ESL) includes thread synchronization logic for Execution Drafting

1.3.1.1 Instruction Fetch Unit

The thread selection policy is as follows - a switch between the available threads every cycle giving priority to the least recently executed thread. The threads become unavailable due to the long latency operations like loads, branch, MUL, and DIV, as well as to the pipeline stalls like cache misses, traps, and resource conflicts. The loads are speculated as cache hits, and the thread is switched-in with lower priority. **This scheme is only followed when not in Execution Drafting mode (determined by a configuration bit). When Execution Drafting mode is enabled, thread selection follows policies in order to synchronize threads. These are discussed in more detail later.**

Instruction cache complex has a 16-Kbyte data, 4-way, 32-byte line size with a single ported instruction tag. It also has dual ported (1R/1W) valid bit array to hold cache line state of valid/invalid. Invalidates access the V-bit array, not the instruction tag. A pseudo-random replacement algorithm is used to replace the cache line.

There is a fully associative instruction TLB with **648 or 16 (TODO)** entries. The buffer supports the following page sizes: 8 Kbytes, 64 Kbytes, 4 Mbytes, and 256 Mbytes. The TLB uses a pseudo least recently used (LRU) algorithm for replacement. Multiple hits in the TLB are prevented by doing an autodemap on a fill.

Two instructions are fetched each cycle, though only one instruction is issued per clock, which reduces the instruction cache activity and allows for an opportunistic line fill. There is only one outstanding miss per thread, and only four per core. Duplicate misses do not issue requests to the L2-cache.

The integer register file (IRF) of the SPARC core has **52.5** Kbytes with 3 read/2 write/1 transport ports. There are **640320** 64-bit registers with error correction code (ECC). Only 32 registers from the current window are visible to the thread. Window changing in background occurs under the thread switch. Other threads continue to access the IRF (the IRF provides a single-cycle read/write access).

1.3.1.2 Execution Unit

The execution unit (EXU) has a single arithmetic logic unit (ALU) and shifter. The ALU is reused for branch address and virtual address calculation. The integer multiplier has a 5 clock latency, and a throughput of half-per-cycle for area saving. One integer multiplication is allowed outstanding per core. ~~The integer multiplier is shared between the core pipe (EXU) and the modular arithmetic (SPU) unit on a round-robin basis.~~ There is a simple non-restoring divider, which allows for one divide outstanding per SPARC core. Thread issuing a MUL/DIV will be rolled back and switched out if another thread is occupying the MUL/DIV units.

1.3.1.3 Load/Store Unit

The data cache complex has an 8-Kbyte data, 4-way, 16-byte line size. It also has single ported data tag. There is a dual-ported (1R/1W) valid bit array to hold cache line state of valid or invalid. Invalidates access the V-bit array but not the data tag. A pseudo-random replacement algorithm is used to replace the data cache line. The loads are allocating, and the stores are non-allocating. The data TLB operates similarly to the instruction TLB.

The load/store unit (LSU) has an 8 entry store buffer per thread, which is unified into a single ~~32~~16 entry array, with RAW bypassing. Only a single load per thread outstanding is allowed. Duplicate requests for the same line are not sent to the L2-cache. The LSU has interface logic to interface to the CPU-cache crossbar (CCX). This interface performs the following operations:

- Prioritizes the requests to the crossbar for floating-point operation (Fpops), ~~streaming operations~~, I\$ and D\$ misses, stores and interrupts, and so on.
- Request priority: imiss>ldmiss>stores,{fpu,~~stream~~,interrupt}.
- Assembles packets for the processor-cache crossbar (PCX).

The LSU handles returns from the CPX crossbar and maintains the order for cache updates and invalidates.

The CCX interface is extended with a transducer to convert between OpenSPARC T1 CCX packets and NOC packets in the PULSA processor.

1.3.1.4 Floating-Point Frontend Unit

The floating-point frontend unit (FFU) decodes floating-point instructions and it also includes the floating-point register file (FRF). Some of the floating-point instructions like move, absolute value, and negate are implemented in the FFU, while the others are implemented in the FPU. The following steps are taken when the FFU detects a floating-point operation (Fpop):

- The thread switches out.
- The Fpop is further decoded and the FRF is read.
- Fpops with operands are packetized and shipped over the ~~crossbar~~NOC to the FPU.
- The computation is done in the FPU and the results are returned by way of the ~~crossbar~~NOC.
- Writeback completed to the FRF and the thread restarts.

1.3.1.5 Trap Logic Unit

The trap logic unit (TLU) has support for six trap levels. Traps cause pipeline flush and thread switch until trap program counter (PC) becomes available. The TLU also has support for up to 64 pending interrupts per thread.

1.3.1.6 Stream Processing Unit

~~The stream processing unit (SPU) includes a modular arithmetic unit (MAU) for crypto (one per core), and it supports asymmetric crypto (public key RSA) for up to a 2048-byte size key. It shares an integer multiplier for modular arithmetic operations. MAU can be used by one thread at a time. The MAU operation is set up by the store to control register, and the thread returns to normal processing. The MAU unit initiates streaming load/store operations to the L2-cache through the crossbar, and compute operations to the multiplier. Completion of the MAU can be checked by polling or issuing an interrupt.~~

1.3.1.7 Execution Drafting Synchronization Logic

The Execution Drafting Synchronization Logic (ESL) includes two synchronization methods to synchronize threads for energy efficiency, the Stall Thread Synchronization Method (STSM) and the Random Thread Synchronization Method (RTSM). It acts as an alternate to the SPARC core thread selection logic and only takes affect when the Execution Drafting enable bit is set. There is a configuration bit to set which thread synchronization method to use.

1.3.1.7.1 Stall Thread Synchronization Method The stall thread synchronization method (STSM), relies on thread PCs for synchronization. When STSM encounters a divergence in the control flow of threads, it compares the PCs of the threads. It selects the thread with the lowest PC to execute, as this indicates it is earliest in program order. Selecting the thread with the lowest PC is an attempt to accelerate the thread to synchronize with the others.

This method works well on if-else structures and loops. If two threads take different paths down an if-else structure, the thread that takes the path earlier in PC order, will exit the if-else structure first. When it exits the if-else structure, it's PC will be larger than that of the other thread. Thus, STSM will execute the other thread until it also exits the if-else structure. Both threads will end up at the end of the if-else structure, synchronized. Similar, logic explains why STSM also performs well on loops. If two threads are executing different numbers of iterations of a loop, lets say 5 iterations and 10 iterations, the threads should stay synchronized for the first 5 iterations, after which one thread will exit the loop. This thread will now have a PC

larger than the other thread, and STSM will cause it to wait until the other thread finishes its iterations and the threads are synchronized again.

There is one caveat with STSM and function calls. Consider the case that one thread takes a function call and the other does not (the function call is within a conditional code block). Assume the function code has a larger PC than any other threads' PC, i.e. it is a shared library or is placed at higher memory addresses. The thread that takes the function call will have a larger PC than the other thread. Thus, it will stall the thread in the function call until the other thread reaches a PC larger than it. This may never occur, and the thread may wait in the function until the other thread completes. Ideally, we would like to execute the larger PC in this case, however this is at odds with STSM. Thus, in order to handle this case, we set a threshold value for STSM. If the difference between PCs is outside this threshold, STSM resorts to fine-grain multithreading. Only when the PCs are inside this threshold does STSM operate as described previously. Thus, in the case of the function call, STSM will alternate executing instructions from both threads, until the thread that took the function call returns, in which the PCs should be within the threshold again. Choosing the value for this threshold is somewhat ad hoc, but empirically 100 has proved to be a reasonable value and is used in the PULSA processor.

Another problem STSM has is related to drafting different programs or different versions of the same program. In these cases, if two threads have the same PCs, they do not necessarily point to the same instruction. This means STSM is actively working against drafting instructions and does not synchronize the threads. This reliance on PCs for synchronization is a shortcoming of STSM. We verify that instructions are the same in our results section before counting them as drafted. The PULSA processor contains a configuration bit to indicate if the threads have identical code.

1.3.1.7.2 Random Thread Synchronization Method The random thread synchronization method (RTSM) is a simple mechanism. When RTSM encounters a divergence in the control flow of threads, it chooses a thread to execute at random until threads are synchronized again. RTSM fills the shortcoming that STSM has with drafting different programs or the same program with different versions and works well in practice. A configuration bit is present in RTSM to indicate whether the threads have identical code as well.

Pseudo-randomness in RTSM is introduced using a linear feedback shift register. This is discussed in more detail later.

1.3.2 CPU-Cache Crossbar

The ~~eight~~**forty** SPARC cores, the ~~four~~ L2-cache banks, the ~~I/O Bridge~~, and the FPU all interface with the ~~crossbar~~**NOC**. However, the core interface to the NOC is simply translated from the crossbar interface to the NOC interface, minimizing the SPARC

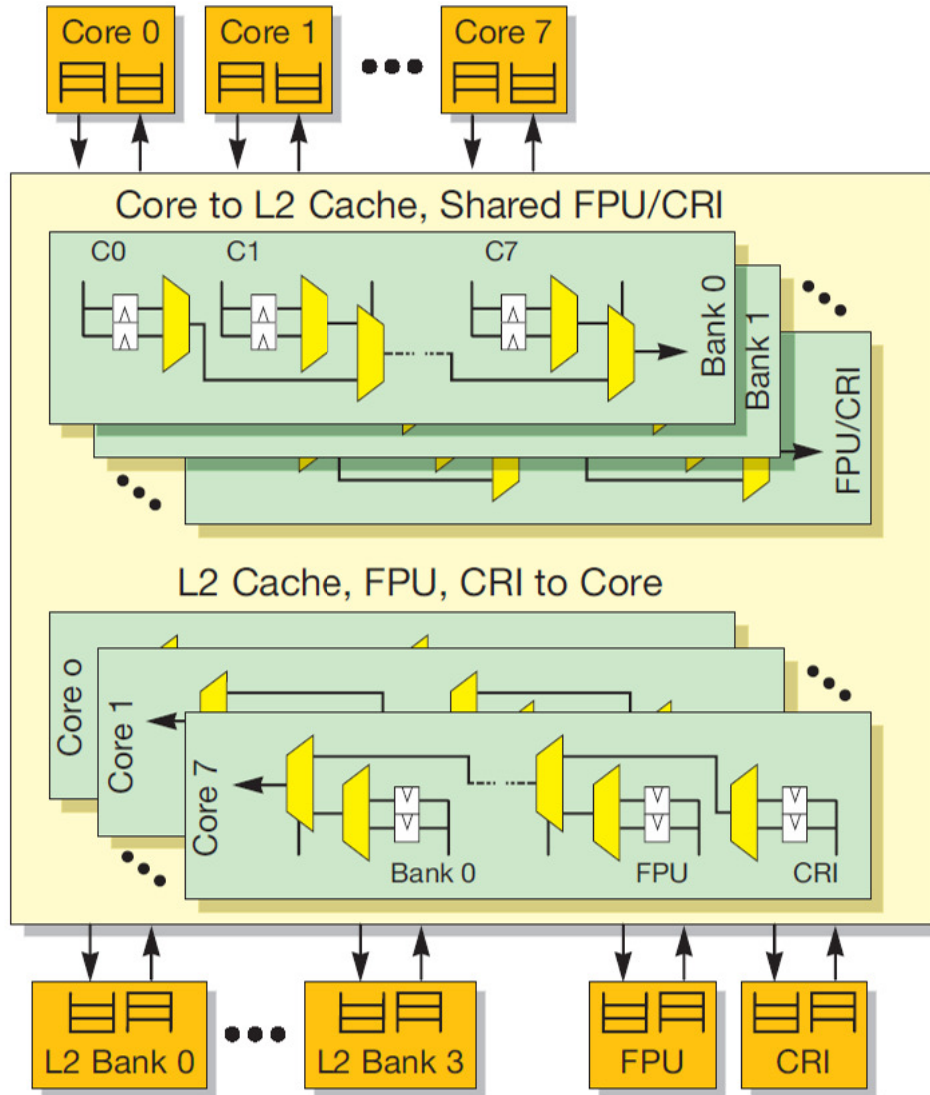


Figure 1.3: CCX Block Diagram

core changes. Thus, the documentation regarding the crossbar is left in place. Figure 1.3 displays the crossbar block diagram. The CPUcache crossbar (CCX) features include:

- Each requester queues up to two packets per destination.
- Three stage pipeline request, arbitrate, and transmit.
- Centralized arbitration with oldest requester getting priority.
- Core-to-cache bus optimized for address plus doubleword store.
- Cache-to-core bus optimized for 16-byte line fill. 32-byte I\$ line fill delivered in two back-to-back clocks.

1.3.3 Floating-Point Unit

A single floating-point unit (FPU) is shared by all eighty SPARC cores. The shared floating-point unit is sufficient for most commercial applications in which typically less than one percent of the instructions are floating-point operations.

1.3.4 L2-Cache

TODO

The L2-cache is banked four ways, with the bank selection based on the physical address bits 7:6. The cache is 3-Mbyte, 12-way set-associative with pseudo-least recently used (LRU) replacement (the replacement is based on a used bit scheme). The line size is 64 bytes. Unloaded access time is 23 cycles for an L1 data cache miss and 22 cycles for an L1 instruction cache miss.

L2-cache has a 64-byte line size, with 64 bytes interleaved between banks. Pipeline latency in the L2-cache is 8 clocks for a load, 9 clocks for an I-miss, with the critical chunk returned first. 16 outstanding misses per bank are supported for a 64 total misses. Coherence is maintained by shadowing the L1 tags in an L2-cache directory structure (the L2-cache is a point of global visibility). DMA from the I/O is serialized with respect to the traffic from the cores in the L2-cache.

The L2-cache directory shadows the L1 tags. The L1 set index and the L2-cache bank interleaving is such that one forth of the L1 entries come from an L2-cache bank. On an L1 miss, the L1 replacement way and set index identifies the physical location of the tag which will be updated by the miss address. On a store, the directory will be cammed. The directory entries are collated by set, so only 64 entries need to be cammed. This scheme is quite power efficient. Invalidates are a pointer to the physical location in the L1-cache, eliminating the need for a tag lookup in the L1-cache.

Coherency and ordering in the L2-cache are described as:

- Loads update directory and fill the L1-cache on return
- Stores are non-allocating in the L1-cache
 - There are two flavors of stores: total store order (TSO) and read memory order (RMO).
Only one outstanding TSO store to the L2-cache per thread is permitted in order to preserve the store ordering. There is no such limitation on RMO stores.
 - No tag check is done at a store buffer insert
 - Stores check directory and determines an L1-cache hit

- ~~– Directory sends store acknowledgements or invalidates to the SPARC core~~
 - ~~– Store updates happens to D\$ on a store acknowledge~~
- ~~Crossbar orders the responses across cache banks.~~

1.3.5 DRAM Controller

DRAM controllers are located in the off-chip FPGA on the PULSA processor. All memory transactions are directed to the off-chip interface to be handled by the DRAM controller in the FPGA.

The OpenSPARC T1 processor DRAM controller is banked four ways, with each L2 bank interacting with exactly one DRAM controller bank (a two-bank option is available for cost-constrained minimal memory configurations). The DRAM controller is interleaved based on physical address bits 7:6, so each DRAM controller bank must have identical dual in-line memory modules (DIMM) installed and enabled.

The OpenSPARC T1 processor uses DDR2 DIMMs and can support one or two ranks of stacked or unstacked DIMMs. Each DRAM bank/port is two DIMMs wide (128-bit + 16-bit ECC). All installed DIMMs must be identical, and the same number of DIMMs (that is, ranks) must be installed on each DRAM controller port. The DRAM controller frequency is an exact ratio of the core frequency, where the core frequency must be at least three times the DRAM controller frequency. The double data rate (DDR) data buses transfer data at twice the frequency of the DRAM controller frequency.

The OpenSPARC T1 processor can support memory sizes of up to 128 Gbytes with a 25 Gbytes/sec peak bandwidth limit. Memory access is scheduled across 8 reads plus 8 writes, and the processor can be programmed into a two-channel mode for a reduced configuration. Each DRAM channel has 128 bits of data and 16 bytes of ECC interface, with chipkill support, nibble error correction, and byte error detection.

1.3.6 I/O Bridge

The I/O subsystem is located in the off-chip FPGA on the PULSA processor. All I/O transactions are directed to the off-chip interface to be handled by the FPGA. I/O is all memory mapped on the PULSA processor, so address snooping and such will be performed in FPGA.

The I/O bridge (IOB) performs an address decode on I/O-addressable transactions and directs them to the appropriate internal block or to the appropriate external interface (J-Bus or the serial system interface). Additionally, the IOB maintains the register status for external interrupts.

1.3.7 J-Bus Interface

The J-Bus interface (JBI) is the interconnect between the OpenSPARC T1 processor and the I/O subsystem. The J-Bus is a 200 MHz, 128-bit wide, multiplexed address

or data bus, used predominantly for direct memory access (DMA) traffic, plus the programmable input/output (PIO) traffic used to control it.

The J-Bus interface is the functional block that interfaces to the J-Bus, receiving and responding to DMA requests, routing them to the appropriate L2 banks, and also issuing PIO transactions on behalf of the processor threads and forwarding responses back.

1.3.8 Serial System Interface

The OpenSPARC T1 processor has a 50 Mbyte/sec serial system interface (SSI) that connects to an external application-specific integrated circuit (ASIC), which in turn interfaces to the boot read-only memory (ROM). In addition, the SSI supports PIO accesses across the SSI, thus supporting optional control status registers (CSR) or other interfaces within the ASIC.

1.3.9 Electronic Fuse

TODO: Do we need this?

The electronic fuse (e-Fuse) block contains configuration information that is electronically burned-in as part of manufacturing, including part serial number and core available information.

Chapter 2

SPARC Core

An ~~OpenSPARC T1~~**PULSA** processor contains ~~eight~~**forty** SPARC cores, and each SPARC core has several function units. These SPARC core units are described in the following sections:

- Section 2.1, SPARC Core Overview and Terminology, on page 16
- Section 2.2, SPARC Core I/O Signal List, on page 19
- Section 2.3, **Changes to SPARC Core in PULSA Processor**, on page 20
- Section 2.4, Instruction Fetch Unit, on page 20
- Section 2.5, **Execution Drafting Synchronization Logic**, on page 35
- Section 2.6, Load Store Unit, on page 40
- Section 2.7, L1.5 Data Cache, on page 52
- Section 2.8, **CPX-NOC Transducer**, on page 57
- Section 2.9, Execution Unit, on page 58
- Section 2.10, Floating-Point Frontend Unit, on page 61
- Section 2.11, Multiplier Unit, on page 62
- Section 2.12, ~~Stream Processing Unit~~, on page 62
- Section 2.13, Memory Management Unit, on page 69
- Section 2.14, Trap Logic Unit, on page 76
- Section 2.15, Core Debug Features, on page 94

2.1 SPARC Core Overview and Terminology

Figure 2.1 presents a high-level block diagram of a SPARC core, and Figure 2.2 shows the general physical location of these units on an example core.

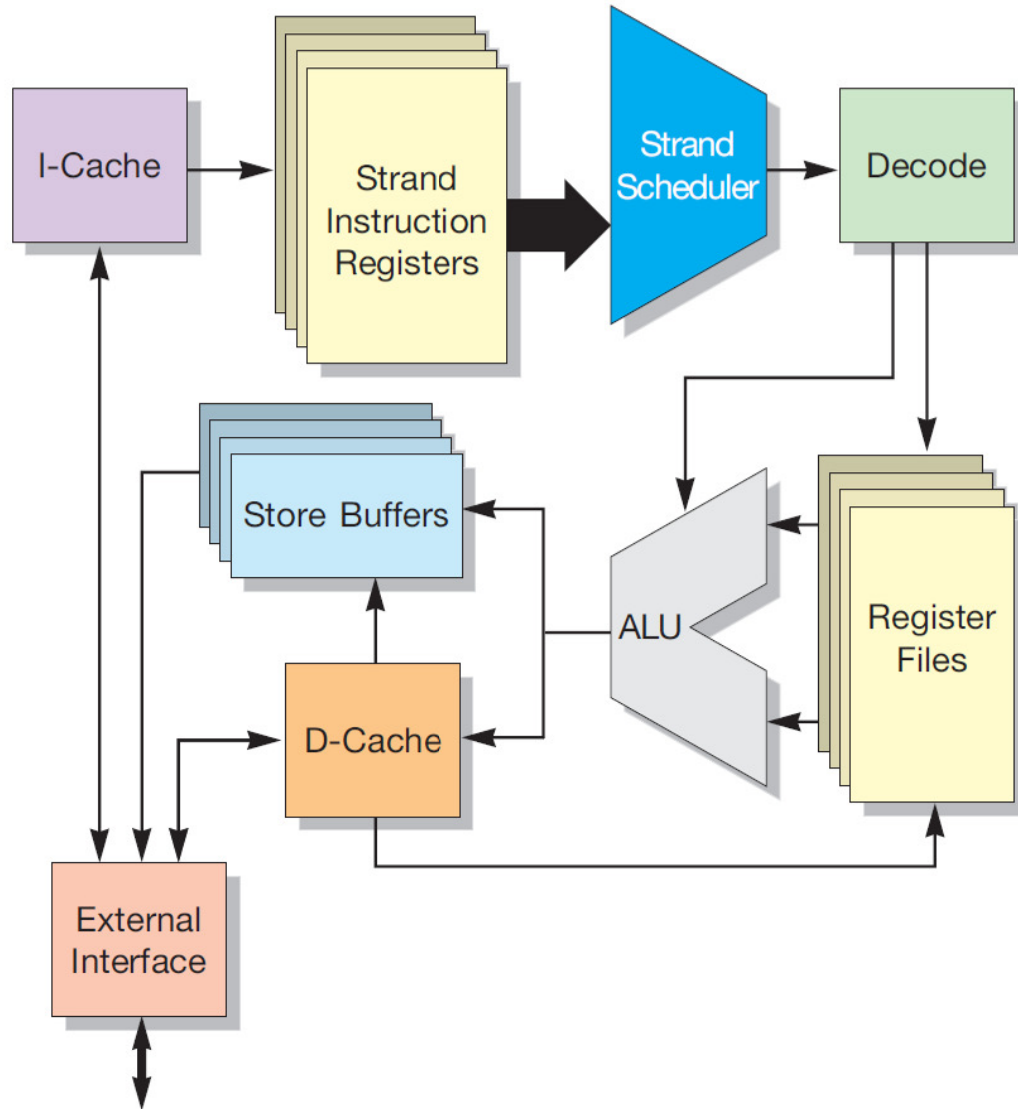


Figure 2.1: SPARC Core Block Diagram

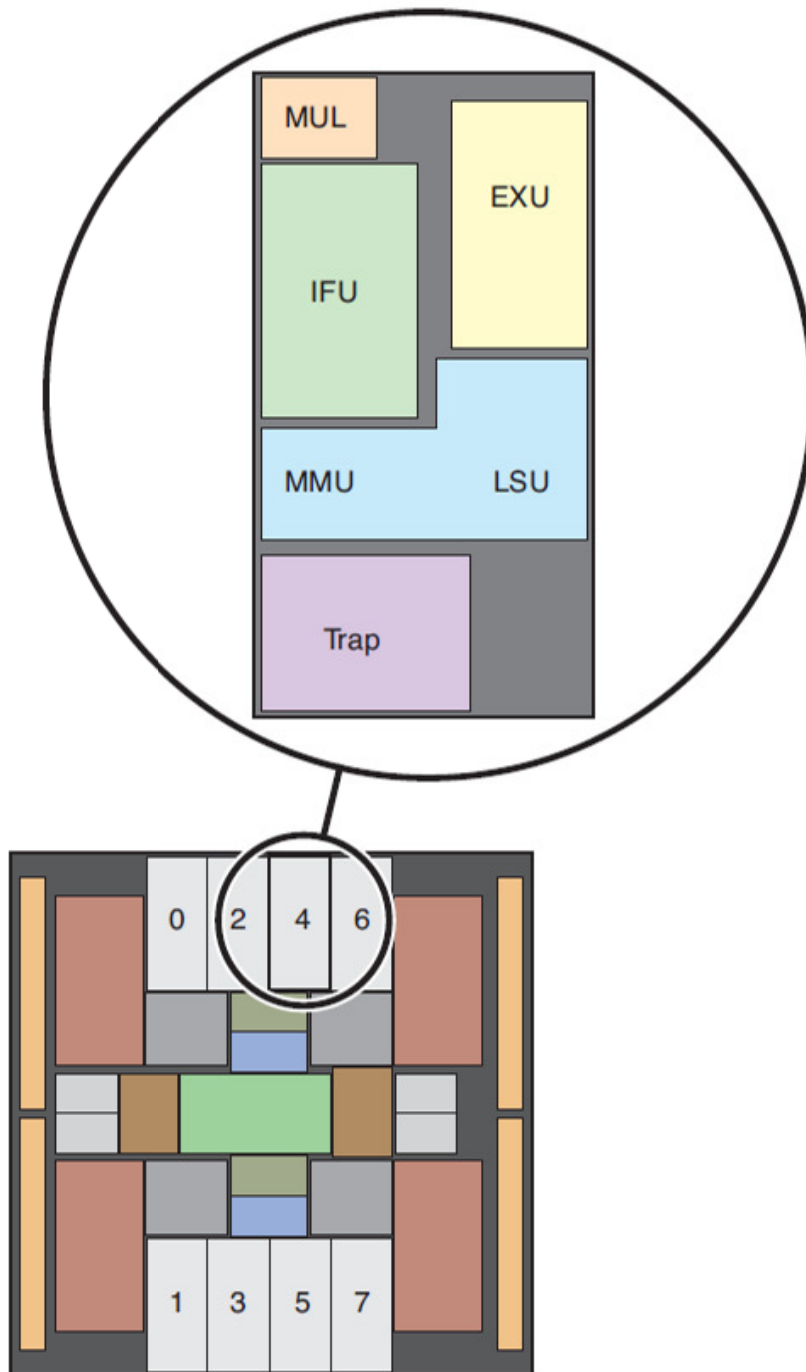


Figure 2.2: Physical Location of Functional Units on an OpenSPARC T1 SPARC Core

Table 2.1 defines acronyms and terms that are used throughout this chapter.

Table 2.1: SPARC Core Terminology

Term	Description
Thread	<p>A thread is a hardware strand (thread and strand will be used interchangeably in this chapter). Each thread, or strand, enjoys a unique set of resources in support of its execution while multiple threads, or strands, within the same SPARC core will share a set of common resources in support of their execution.</p> <p>The per-thread resources include registers, a portion of I-fetch data-path, store buffer, and miss buffer. The shared resources include the pipeline registers and data-path, caches, translation lookaside buffers (TLB), and execution unit of the SPARC Core pipeline.</p>
ST	Single threaded.
MT	Multi-threaded.
Hypervisor (HV)	The hypervisor is the layer of system software that interfaces with the hardware.
Supervisor (SV)	The supervisor is the layer of system software such as operation system (OS) that executes with privilege.
Long latency instruction (LLI)	LLI represents an instruction that would take more than one SPARC core clock cycle to make its results visible to the next instruction.

Figure 2.3 shows the view from virtualization, which illustrates the relative privileges of the various software layers.

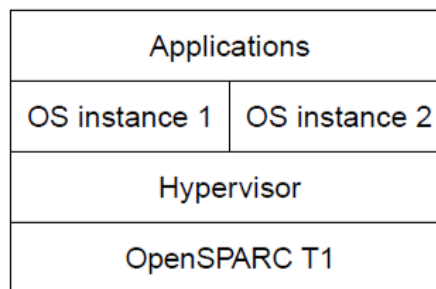


Figure 2.3: Virtualization of Software Layers

2.2 SPARC Core I/O Signal List

Table 2.2 lists and describes the SPARC Core I/O signals.

Table 2.2: SPARC Core I/O Signal List

Signal Name	I/O	Source/Destination	Description
pcx_spc_grant_px[4:0]	In	CCX:PCX	PCX to processor grant info
cpx_spc_data_rdy_cx2	In	CCX:CPX	CPX data in-flight to SPARC
cpx_spc_data_cx2[144:0]	In	CCX:CPX	CPX to SPARC data packet
const_cpuid[3:0]	In	Hard wired	CPU ID
const_maskid[7:0]	In	CTU	Mask ID
ctu_tck	In	CTU	To IFU of sparc_ifu.v
ctu_sscan_se	In	CTU	To IFU of sparc_ifu.v
ctu_sscan_snap	In	CTU	To IFU of sparc_ifu.v
ctu_sscan_tid[3:0]	In	CTU	To IFU of sparc_ifu.v
ctu_tst_mbist_enable	In	CTU	To test_stub of test_stub_bist.v
efc_spc_fuse_clk1	In	EFC	
efc_spc_fuse_clk2	In	EFC	
efc_spc_ifuse_ashift	In	EFC	
efc_spc_ifuse_dshift	In	EFC	
efc_spc_ifuse_data	In	EFC	
efc_spc_dfuse_ashift	In	EFC	
efc_spc_dfuse_dshift	In	EFC	
efc_spc_dfuse_data	In	EFC	
ctu_tst_macrotest	In	CTU	To test_stub of test_stub_bist.v
ctu_tst_scan_disable	In	CTU	To test_stub of test_stub_bist.v
ctu_tst_short_chain	In	CTU	To test_stub of test_stub_bist.v
global_shift_enable	In	CTU	To test_stub of test_stub_two_bist.v
ctu_tst_scanmode	In	CTU	To test_stub of test_stub_two_bist.v
spc_scanin0	In	DFT	Scan in
spc_scanin1	In	DFT	Scan in
cluster_cken	In	CTU	To spc_hdr of cluster_header.v
gclk	In	CTU	To spc_hdr of cluster_header.v
cmp_grst_l	In	CTU	Synchronous reset
cmp_arst_l	In	CTU	Asynchronous reset
ctu_tst_pre_grst_l	In	CTU	To test_stub of test_stub_bist.v
adbgininit_l	In	CTU	Asynchronous reset
gdbgininit_l	In	CTU	Synchronous reset
spc_pcx_req_pq[4:0]	Out	CCX:PCX	Processor to pcx request
spc_pcx_atom_pq	Out	CCX:PCX	Processor to pcx atomic request
spc_pcx_data_pa[123:0]	Out	CCX:PCX	Processor to pcx packet
spc_sscan_so	Out	DFT	Shadow scan out

spc_scanout0	Out	DFT	Scan out
spc_scanout1	Out	DFT	Scan out
tst_ctu_mbist_done	Out	CTU	MBIST done
tst_ctu_mbist_fail	Out	CTU	MBIST fail
spc_efc_ifuse_data	Out	EFC	From IFU of sparc_ifu.v
spc_efc_dfuse_data	Out	EFC	From IFU of sparc_ifu.v

2.3 Changes to SPARC Core in PULSA Processor

The high-level changes that were made to the SPARC core for the PULSA processor are as follows:

- Reduction from four threads to two threads per core
- Reduction from 64 to 8 or 16 TLB entries (TODO!!)
- Removal of SPU
- Addition of Execution Drafting support (Section 2.5)
- Addition of L1.5 cache for writeback capabilities (Section 2.7)
- Addition of CPX-NOC transducer to translate between CPX packets and NOC packets (Section 2.8)

2.4 Instruction Fetch Unit

The instruction fetch unit (IFU) is responsible for maintaining the program counters (PC) of different threads and fetching the corresponding instructions. The IFU also manages the level 1 I-cache (L1I) and the instruction translation lookaside buffer (ITLB), as well as managing and scheduling the ~~four~~**two** threads in a SPARC core. The SPARC core pipeline resides in the IFU, which controls instruction issue and instruction flow in the pipeline. The IFU decodes the instructions flowing through the pipeline, schedules interrupts, and it implements the idle/resume states of the pipeline. The IFU also logs the errors and manages the error registers.

2.4.1 SPARC Core Pipeline

There are six stages in a SPARC core pipeline:

- Fetch F-stage
- Thread selection S-stage

- Decode D-stage
- Execute E-stage
- Memory M-stage
- Writeback W-stage

The I-cache access and the ITLB access take place in fetch stage. A selected thread (hardware strand) will be picked in the thread selection stage. The instruction decoding and register file access occur in the decode stage. The branch evaluation takes place in the execution stage. The access to memory and the actual writeback will be done in the memory and writeback stages. Figure 2.4 illustrates the SPARC core pipeline and support structures.

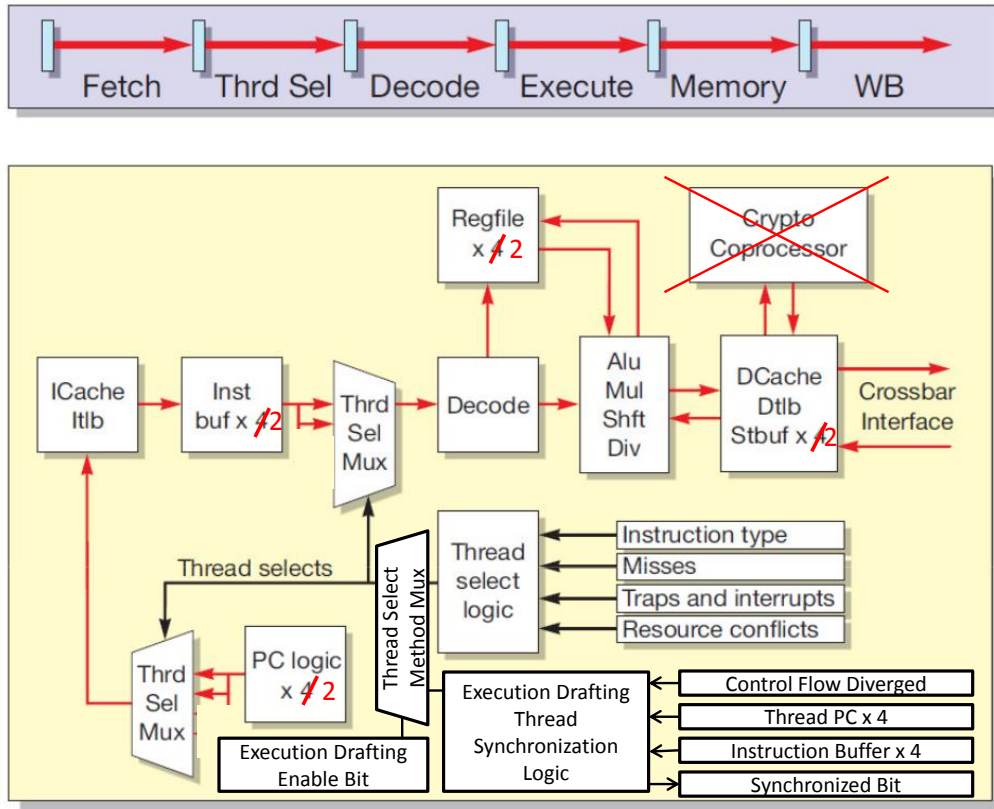


Figure 2.4: SPARC Core Pipeline and Support Structures

The instruction fill queue (IFQ) feeds into the I-cache. The missed instruction list (MIL) stores the addresses that missed the I-cache and the ITLB, and the MIL feeds into the load store unit (LSU) for further processing. The instruction buffer is two levels deep, and it includes the thread instruction (TIR) and next instruction (NIR) unit. Thread selection and scheduler (S-stage) resolves the arbitration among the TIR, NIR, branch-PC, and trap-PC to pick one thread send it to the decode stage (D-stage). Figure 2.5 shows the support structure for this portion of the thread pipeline.

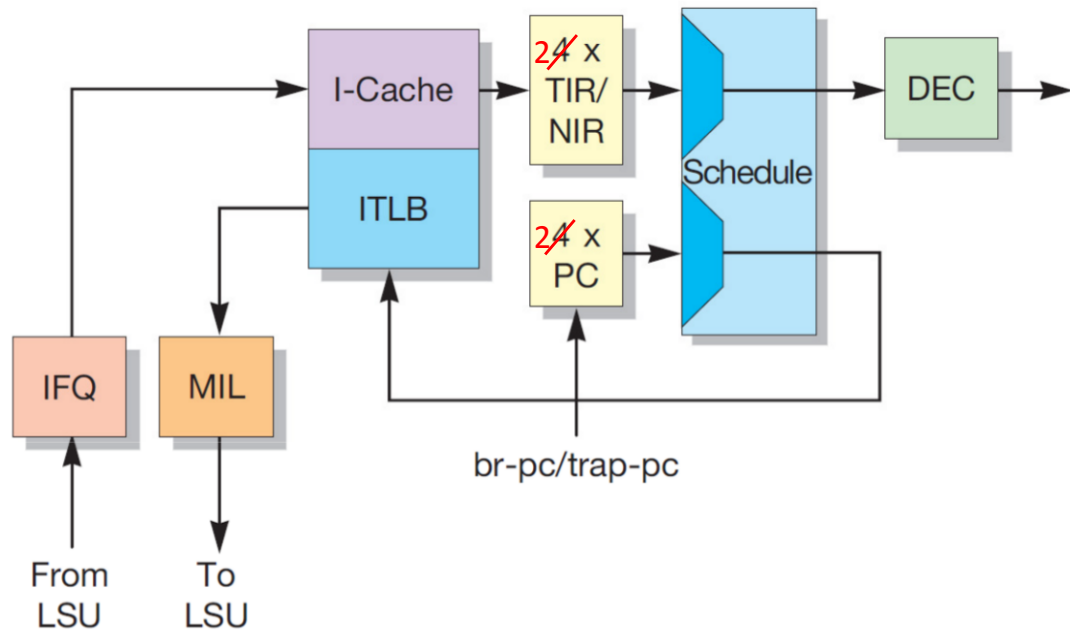


Figure 2.5: Frontend of the SPARC Core Pipeline

2.4.2 Instruction Fetch

The instruction fetch unit (IFU) maintains the program counters (PC) and the next program counters (NPC) of all live instructions executed on the ~~OpenSPARC~~ ~~T1PULSA~~ processor. For every SPARC core clock cycle, two instructions are fetched for every instruction issued. This two fetches per one issue relationship is intended to reduce the I-cache access in order to allow the opportunistic I-cache line fill. Each thread is allowed to have one outstanding I-cache miss, and the SPARC core allows a total of ~~four~~ ~~two~~ I-cache misses. Duplicated I-cache misses do not induce the redundant fill request to the level 2 cache (L2-cache).

2.4.3 Instruction Registers and Program Counter Registers

In the instruction buffer, there are two instruction registers per thread the thread instruction register (TIR) and the next instruction register (NIR). The TIR contains the current thread instruction in the thread selection stage (S-stage), and the NIR contains the next instruction. An I-cache miss fill bypasses the I-cache and writes directly to the TIR, but it never writes to the NIR.

The thread scheduler selects a valid instruction from the TIR. After selecting the instruction, the valid instruction will be moved from the NIR to the TIR. If no valid instruction exists in the TIR, a no operation (NOP) instruction will be inserted.

There is one program counter (PC) register per thread. The next-program counter (NPC) could come from one of these sources:

1. Branch
2. TrapPC
3. Trap NPC
4. Rollback (a thread rolled back due to a load miss)
5. $PC + 4$

The IFU tracks the PC and NPC through W-stage. The last retired PC will be saved in the trap logic unit (TLU), and, if a trap occurs, it will also be saved in the trap stack.

2.4.4 Level 1 Instruction Cache

The instruction cache is commonly referred to as the level 1 instruction cache (L1I). The L1I is physically indexed and tagged and is 4-way set associative with 16 Kbytes of data. The cache-line size is 32 bytes. The L1I data array has a single port, and the I-cache fill size is 16 bytes per access. The characteristics of cached data include 32-bit instructions, 1-bit parity, and 1-bit predecode. The tag array also has a single port.

There is a separate array for valid bit (V-bit). This V-bit array holds the cache line state of either valid or invalid, and the array has one read port and one write port (1R1W). The cache line invalidation only accesses the V-bit array, and the cache line replacement policy is pseudo-random.

The read access to the I-cache has a higher priority over the write access. The ASI read and write accesses to the I-cache are set to lower priorities. The completion of the ASI accesses are opportunistic, and there is fairness mechanism built in to prevent the starvation of service to ASI accesses.

The maximum wait period for a write access to the I-cache is 25 SPARC core clock cycles. A wait longer than 25 clock cycles will stall the SPARC core pipeline in order to allow the I-cache write access completion.

2.4.5 I-Cache Fill Path

I-cache fill packets come from the level 2 cache ~~across the NOC~~ to processor interface (CPX) by way of the load store unit (LSU). Parity and predecode bits will be calculated before the I-cache fills up. CPX packets include invalidations (invalidation packets are nonblocking), test access point (TAP) reads and writes, and error notifications. The valid bit array in the I-cache has a dedicated port for servicing the invalidation packets. Figure 2.6 illustrates the I-cache fill path.

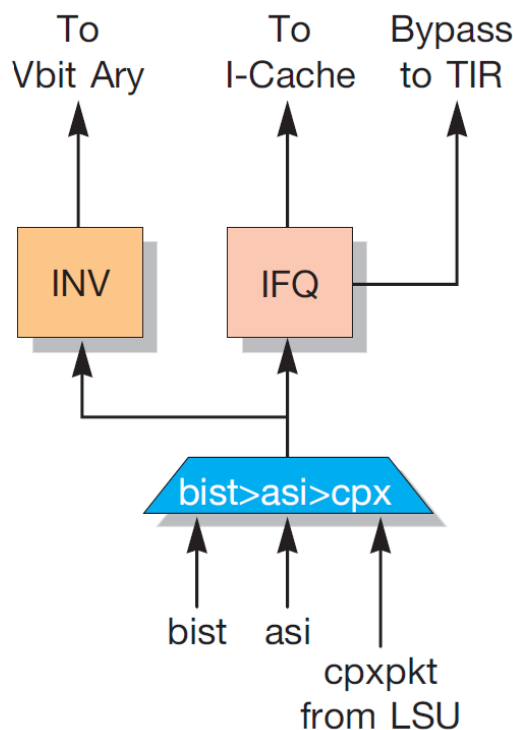


Figure 2.6: I-Cache Fill Path

The I-cache line size is 32 bytes, and a normal I-cache fill takes two CPX packets of 16 bytes each. The instruction fill queue (IFQ) has a depth of two. An I-cache line will be invalidated when the first CPX packet is delivered and filled in the I-cache. That cache line will be marked as valid when the second CPX packet is delivered and filled. I-cache control guarantees the atomicity of the I-cache line fill action between the two halves of the cache line being filled.

~~An instruction fetch from the boot PROM, by way of the system serial interface (SSI), is a very slow transaction. The boot prom is a part of the I/O address space. All instruction fetches from the I/O space are non-cacheable. The boot PROM fetches only one 4-byte instruction at a time. This 4-byte instruction is replicated four times during the formation of the CPX packet. Only one CPX packet of non-cacheable instructions will be forwarded to the IFQ. The non-cacheable instructions fetched from the boot PROM will not be filled in the I-cache. They will be sent to (or,~~

bypassed to) the thread instruction register (TIR) directly. **TODO: How will boot work?**

2.4.6 Alternate Space Identifier Accesses, I-Cache Line Invalidations, and Built-In Self-Test Accesses to the I-Cache

Alternate space identifiers (ASI) accesses to the I-cache, and the built-in self-test (BIST) accesses to the I-cache, go through the IFQ data-path to the I-cache. All ASI accesses and BIST accesses will cause the SPARC core pipeline to stall, so these accesses are serviced almost immediately.

The load store unit (LSU) initiates all ASI accesses. The LSU serializes all ASI accesses so that the second access will not be launched until the first access has been acknowledged. ASI accesses tend to be slow, and data for an ASI read will be sent back later.

A BIST operation requires atomicity, and it assumes and accommodates no interruptions until it completes.

Level 2 cache invalidations will always undergo a CPU-ID check in order to ensure that this invalidation packet is indeed meant for the specified SPARC core. In the following cases, an invalidation could be addressing anyone:

- A single I-cache line invalidation due to store acknowledgements, or due to a load exclusivity requiring that the invalidation of the other level 1 I-caches resulted from the self-modifying code.
- Invalidating two I-cache lines because of a cache-line eviction in the level 2 cache (L2-cache).
- Invalidating all ways in a given set due to error conditions, such as encountering a tag ECC error in a level 2 cache line.

2.4.7 I-Cache Miss Path

A missed instruction list (MIL) is responsible for sending the I-cache miss request to the level 2 cache (L2-cache) in order to get an I-cache fill. The MIL has one entry per thread, which supports a total of **fourtwo** outstanding I-cache misses for all **fourtwo** threads in the same SPARC core at the same time. Each entry in the MIL contains the physical address (PA) of an instruction that missed the I-cache, the replacement way information, the MIL state information, the cacheability, the error information, and so on. The PA tracks the I-fetch progress from the indication of an I-cache miss until the I-cache has been filled. The dispatch of I-cache miss requests from different threads follow a fairness mechanism based on a round-robin algorithm.

Figure 2.7 illustrates the I-cache miss path.

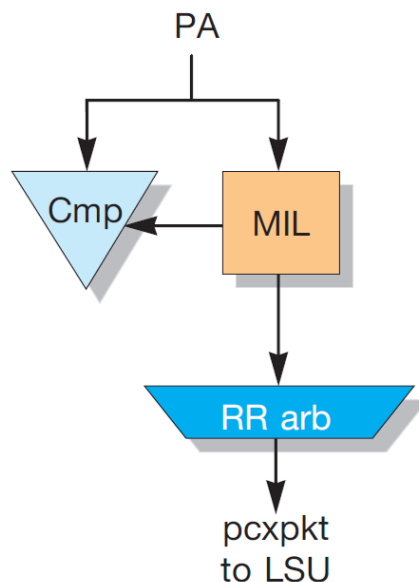


Figure 2.7: I-Cache Miss Path

The MIL keeps track of the physical address (PA) of an instruction that missed the I-cache. A second PA that matches the PA of an already pending I-cache miss will cause the second request to be put on hold and marked as a child of the pending I-cache miss request. The child request will be serviced when the pending I-cache miss receives its response. The MIL uses a linked list to track and service the duplicated I-cache miss request. The depth for such a linked list is four.

The MIL cycles through the following states:

1. Make request.
2. Wait for an I-cache fill.
3. Fill the first 16 bytes of data. The MIL sends a speculative completion notification to the thread scheduler at the completion of filling the first 16 bytes.
4. Fill the second 16 bytes of data. The MIL sends a completion notification to the thread scheduler at the completion of filling the second 16 bytes.
5. Done.

An I-cache miss request could be canceled because of, for example, a trap. The MIL still goes through the motions of filling a cache line but it does not bypass it to the thread instruction register (TIR). A pending child request must be serviced even if the original parent I-cache miss request was cancelled.

When a child I-cache miss request crosses with a parent I-cache miss request, the child request might not be serviced before the I-cache fill for the parent request occurs. The

child instruction fetch shall be retired (rolled back) to the F-stage to allow it to access the I-cache. This kind of case is referred to as miss-fill crossover.

2.4.8 Windowed Integer Register File

The integer register file (IRF) contains 52.5 Kbytes of storage, and has three read ports, 2 write ports, and one transfer port (3R/2W/1T). The IRF houses 640320 64-bit registers that are protected by error correcting code (ECC). All read or write accesses can be completed in one SPARC core clock cycle.

Figure 2.8 illustrates the structure of an integer architectural register file (IARF) and an integer working register file (IWRF).

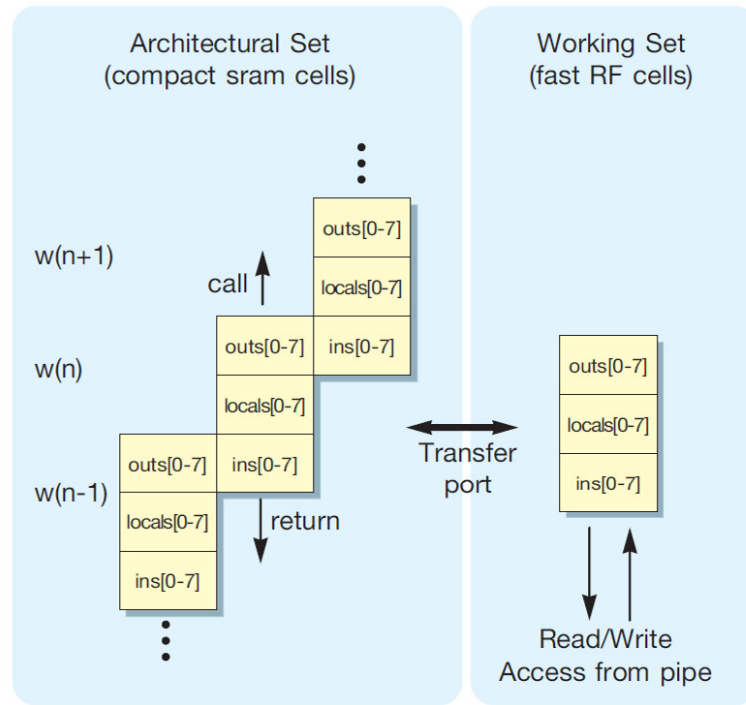


Figure 2.8: IARF and IWRF File Structure

Each thread requires 128 registers for the eight windows (with 16 registers per window), and four sets of global registers with eight global registers per set. There are 160 registers per thread, and there are fourtwo threads per SPARC core. There are a total of 640320 registers per SPARC core.

Only 32 registers from the current window are visible to the thread. A window change occurs in the background under thread switching while the other threads continue to access integer register file.

Please refer to OpenSPARC T1 Processor Megacell Specification for additional details on the IRF.

2.4.9 Instruction Table Lookaside Buffer

The instruction table lookaside buffer (ITLB) is responsible for address translation and tag comparison. The ITLB is always turned-on for non-hypervisor mode operations, and the ITLB is always turned-off for hypervisor mode operations.

The ITLB contains 648 or 16 (TODO) entries. The replacement policy is a pseudo least recently used (pseudo-LRU) policy, which is the same policy as that for the I-cache.

The ITLB supports page sizes of 8 Kbytes, 64 Kbytes, 4 Mbytes, and 256 Mbytes. Multiple hits in the ITLB are prevented by the autodemap feature in an ITLB fill.

2.4.10 Thread Selection Policy

Thread selection is dependent on whether Execution Drafting is enabled in the core. A configuration bit is used to select the thread selection decision from the Execution Drafting synchronization logic or from the SPARC core default thread selection policy. If Execution Drafting is enabled, thread selection follows the algorithms described in Section 2.5. However, if Execution Drafting is not enabled the default SPARC core thread selection policy is used and operates as follows.

Thread switching takes place during every SPARC core clock cycle. At the time of a thread selection, the priority is given to the least recently executed yet available thread. Load instructions will be speculated as cache hits and the thread executing a load instruction will be deemed as available and allowed to be switched-in with a low priority.

A thread could become unavailable due to one of these reasons:

1. The thread is executing one of the long latency instructions, such as load, branch, multiplication, division, and so on.
2. The SPARC core pipeline has been stalled due to one of the long latency operations, such as encountering a cache miss, taking a trap, or experiencing a resource conflict.

2.4.11 Thread States

A thread cycles through these three different states idle, active, and halt. Figure 2.9 illustrates the basic transition of non-active states.

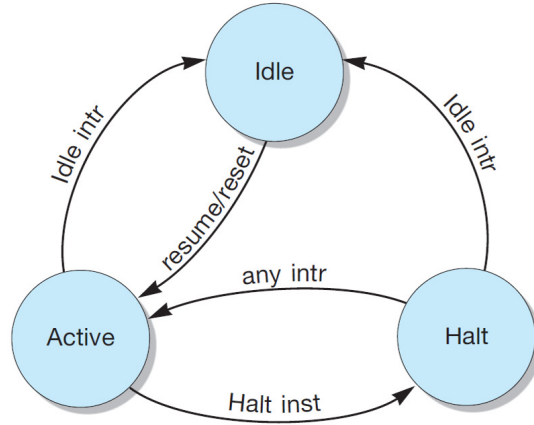


Figure 2.9: Basic Transition of Non-Active States

A thread is in an idle state at power-on. An active thread will only be transitioned to an idle state after a wait mask for an I-cache fill has been cleared.

A thread in the idle state should not receive the resume command without a previous reset. When a thread is violated, the integrity of the hardware behavior cannot be guaranteed.

Figure 2.10 illustrates the thread state transition of an active thread.

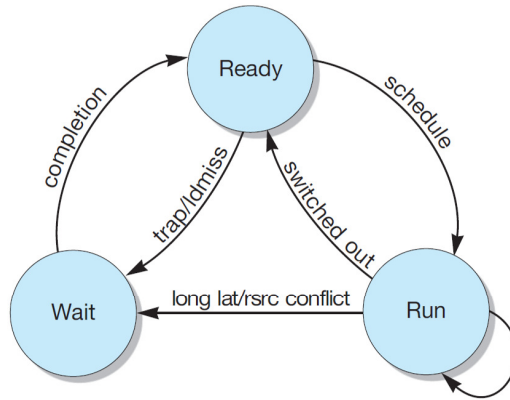


Figure 2.10: Thread State Transition of an Active Thread

An active thread could be placed in the wait state because of any of the following reasons:

1. Wait for an I-cache fill.
2. Wait due to store buffer full.
3. Wait due to long latency, or a resource conflict where all resource conflicts arise because of long latency.
4. Wait due to any combination of the preceding reasons.

The current wait state is tracked in the IFU wait masks.

Figure 2.11 illustrates the state transition for a thread in speculative states.

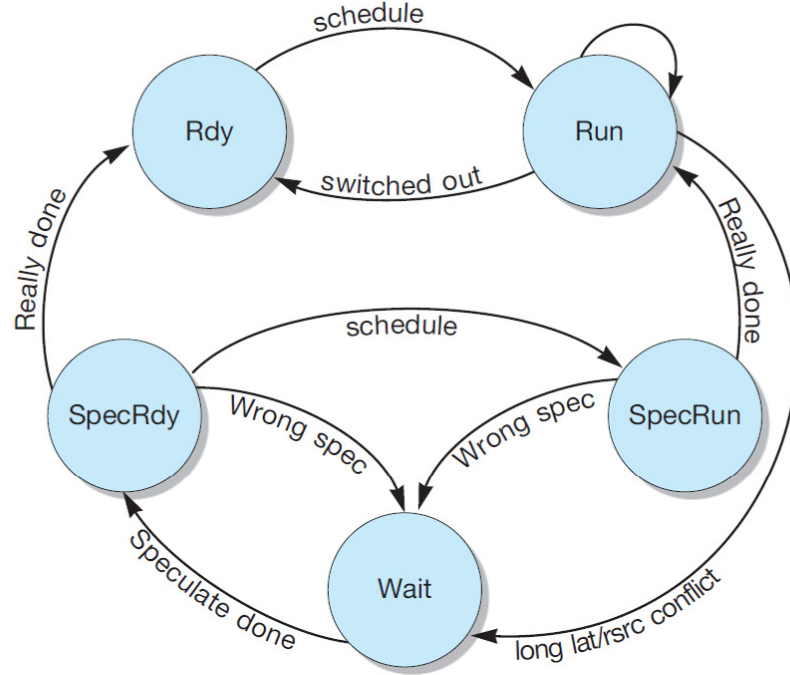


Figure 2.11: State Transition for a Thread in Speculative States

2.4.12 Thread Scheduling

A thread can be scheduled when it is in one of the following five states: idle (which happens infrequently, and generally results from a reset or resume interrupt), Rdy, SpecRdy, Run, and SpecRun. The thread priority in each state is different at the time for scheduling. The priority scheme can be characterized as follows:

$$\text{Idle} > \text{Rdy} > \text{SpecRdy} > (\text{Run} = \text{SpecRun})$$

The fairness scheme for threads in the Run state or the SpecRun state is a roundrobin algorithm with the least recently executed thread winning the selection.

Within Idle threads, the priority scheme is as follows:

$$T0 \text{ (thread 0)} > T1 \text{ (thread 1)} > \text{~~T2 (thread 2)}~~ > \text{~~T3 (thread 3)}~~$$

2.4.13 Rollback Mechanism

The rollback mechanism provides a way of recovering from a scheduling error. The two reasons for performing a rollback include:

1. All of the stall conditions, or switch conditions, were not known at the time of the scheduling.
2. The scheduling was done speculatively on purpose.

For example, after issuing a load, the scheduler will speculate a level 1 D-cache hit performance reasons. If the speculation was incorrect (because of encountering a load miss), all of the instructions after the speculative load instruction must be rolled back. Otherwise, the performance gain would be a substantial.

Rolled back instructions must be restarted from the S-stage or F-stage of the SPARC core pipeline. Figure 2.12 illustrates the pipeline graph for the rollback mechanism.

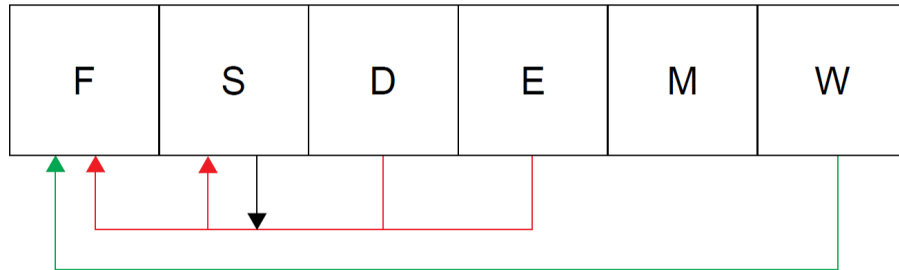


Figure 2.12: Rollback Mechanism Pipeline Graph

The three rollback cases include:

1. E to S and D to F
2. D to S and S to F
3. W to F

The possible conditions causing a rollback case 1 or a case 2 include:

- Instruction(s) following a load miss
- Resource conflict due to long latency
- Store buffer full
- I-cache instruction parity error
- I-fetch retry

The possible conditions causing rollback case 3 include:

- Encountering an ECC error during the instruction register file access.
- The floating-point store instruction encountering an ECC error during the floating-point register file access.
- Instruction(s) following a load hits the store buffer and the level 1 D-cache, where the data has not been bypassed from the store buffer to the level 1 D-cache.
- Encountering D-cache parity errors.
- Launching an idle or resume interrupt where the machine states must be re-stored.
- An interrupt has been scheduled but not yet taken.

2.4.14 Instruction Decode

The IFU decodes the SPARC V9 instructions, and the floating-point frontend unit (FFU) decodes the floating-point instructions. Unimplemented floating-point instructions will cause an `fp_exception_other` trap with a `FSR.ftt=3` (`unimplemented_FPop`). These operations will be emulated by the software.

The privilege is checked in D-stage of the SPARC core pipeline. Some instructions can only be executed with hypervisor privilege or with supervisor privilege.

The branch condition is also evaluated in the D-stage, and the decision for annulling a delay slot is made in this stage as well.

2.4.15 Instruction Fetch Unit Interrupt Handling

All interrupts are delivered to the instruction fetch unit (IFU). For each received interrupt, the IFU shall check the bits `pstate.ie` (the interrupt enable bit in the processor state register) and `hpstate` (the hypervisor state) before scheduling the interrupt. All interrupts will be prioritized (refer to the *Programmers Reference Manual* for these priority assignments). Once prioritized, the interrupts will be scheduled just like the instructions.

When executing in the hypervisor (HV) state, an interrupt with a supervisor (SV) privilege will not be serviced at all. An hypervisor state execution shall not be blocked by anything with supervisor privilege.

Nothing could block the scheduling of a reset, idle, or resume interrupt.

Some interrupts are asserted by a level while others are asserted by a pulse. The IFU remembers the form the interrupts were originated in order to preserve the integrity of the scheduling.

2.4.16 Error Checking and Logging

Parity protects the I-cache data and the tag arrays. The error correction action is to re-fetch the instruction from the level 2 cache.

The instruction translation lookaside buffer (ITLB) array is parity decoded without an error-correction mechanism, so all errors are fatal.

All on-core errors, and some of the off-core errors, are logged in the per-thread error registers. Refer to the *Programmers Reference Manual* for details.

The instruction fetch unit (IFU) maintains the error injection and the error enabling registers, which are accessible by way of ASI operations.

Critical states (such as program counter (PC), thread state, missed instruction list (MIL), and so on) can be snapped and scanned out on-line. This process is referred to as a *shadow scan*.

2.5 Execution Drafting Synchronization Logic

The Execution Drafting Synchronization Logic (ESL) acts as an alternate thread selection policy which attempts to synchronize threads such that identical instructions are issued consecutively. When duplicate instructions are executed consecutively, transistor switching is reduced. Decoding of duplicate instructions is identical to the lead instruction; thus it only needs to be performed once. All control logic, i.e. ALU opcodes, multiplexer select signals, etc., will not switch. Only the data may differ between the execution of identical instructions. Thus everything other than data buses remains static. In fact, it is possible that some data may be shared between threads. Immediate constants may be identical, the threads may be from the same program and, thus, share memory, virtual memory addresses may be the same, and some data values are just commonly found in programs, such as zero. This concept of reducing switching, thereby reducing energy, by executing duplicate instructions consecutively is called drafting.

Reducing switching is not the only benefit from Execution Drafting that results in reduced energy consumption. Knowing that identical threads are synchronized means that you only need to fetch one stream of instructions. Thus, the fetch stage is disabled for all threads that are synchronized except for the leader. Data dependent control flow may cause threads to diverge. These divergences result in the fetch stage needing to be re-enabled for all diverged threads.

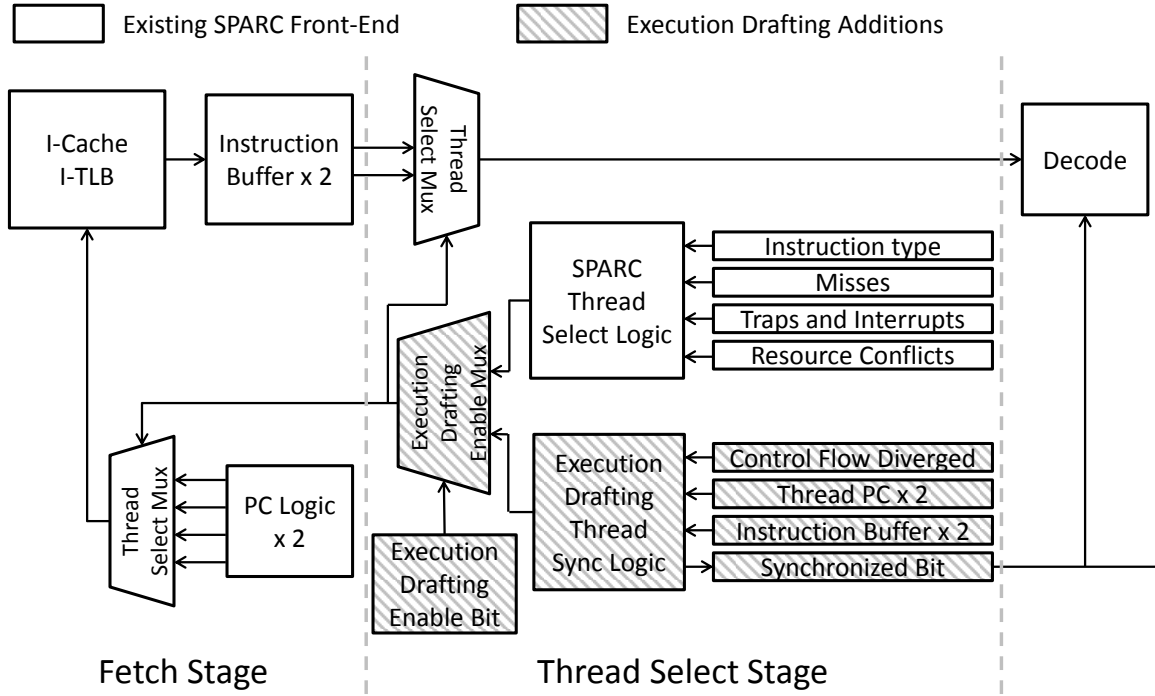


Figure 2.13: ESL Additions and Interactions with SPARC Core Front-End

The ESL performs all logic functions associated with Execution Drafting. It contains logic for selecting threads for execution based on trying to synchronize the threads, for which there are two methods: stall thread synchronization method (STSM) and random thread synchronization method (RTSM). These methods are discussed in more detail in their respective subsections (Sections 2.5.1 and 2.5.2). The ESL also outputs signals to the fetch stage to indicate when it needs to be disabled for a given thread and to stages further down the pipe to indicate whether instructions need to be executed multiple times for each thread (as only one instance of a duplicate instruction is issued down the pipe).

Figure 2.13 illustrates additions to the SPARC core front-end and how the ESL interacts with the SPARC core front-end. A multiplexer selects between the default SPARC core thread selection policy decision and the ESL thread selection decision. The multiplexer is controlled by a Execution Drafting enable configuration bit. The ESL accepts signals from other stages in the pipeline in order to make its thread selection decision (control flow information, thread PCs, and instructions). The ESL outputs the thread selection decision in addition to a synchronized bit, which determines whether the two threads are synchronized. The synchronized bit is used by the fetch stage to disable certain threads and by other stages downstream in the pipe to determine whether to execute instructions once or once for each thread.

The internal structure of the ESL is shown in Figure 2.14. There are three configuration bits associated with the ESL:

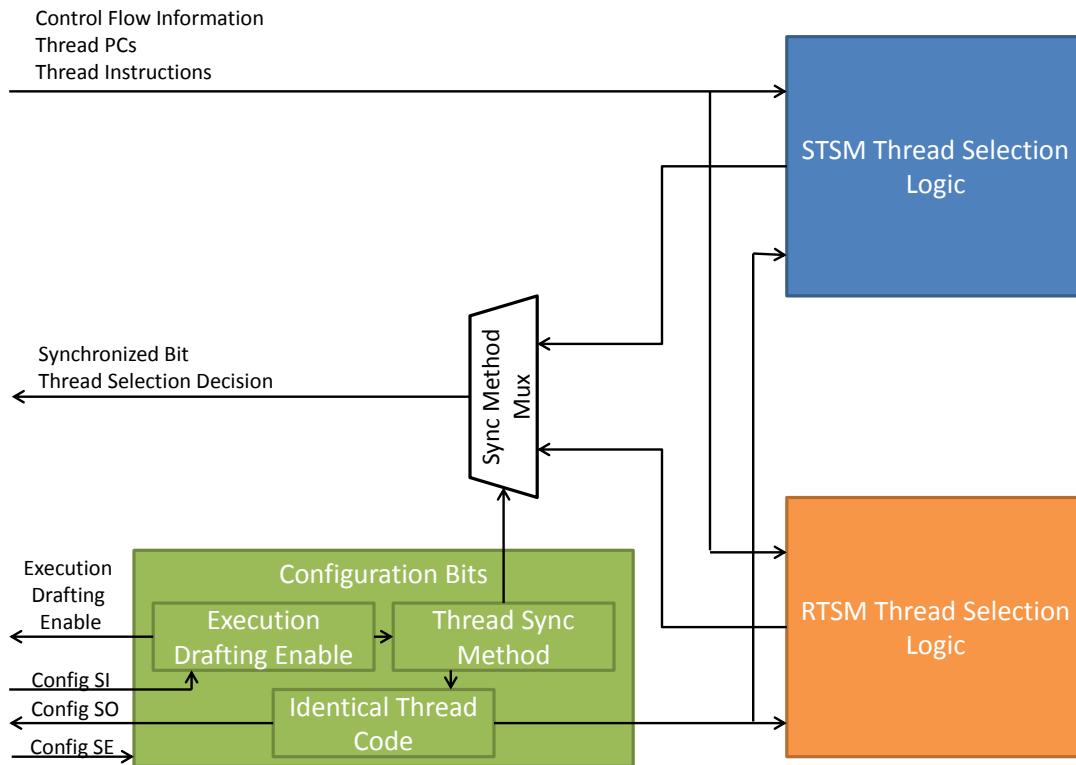


Figure 2.14: Internal Structure of the ESL

- **Execution Drafting Enable** - Whether Execution Drafting is currently enabled for this core
- **Thread Synchronization Method** - Which thread synchronization method to use (STSM or RTSM)
- **Identical Thread Code** - Whether the code for the threads are identical or just similar (Whether identical PCs in the two threads always point to the same instructions)

The configuration can be read and written through a configuration scan chain. The Execution Drafting enable configuration bit is an output from the ESL module to be used by the external multiplexer selecting between the thread selection policies. The identical thread code configuration bit is used by the thread synchronization methods to determine if the threads are synchronized (i.e. whether the fetch stage can be disabled for one thread). The thread synchronization method configuration bit controls the synchronization method multiplexer which selects between the thread selection decision and synchronized bits from the STSM or RTSM schemes, which are outputs from the ESL module. Last, the control flow information, thread PCs, and

thread instructions, which are inputs to the ESL module are used by the thread synchronization methods for synchronization.

2.5.1 Stall Thread Synchronization Method

The stall thread synchronization method (STSM), relies on thread PCs for synchronization. When STSM encounters a divergence in the control flow of threads, it compares the PCs of the threads. It selects the thread with the lowest PC to execute, as this indicates it is earliest in program order. Selecting the thread with the lowest PC is an attempt to accelerate the thread to synchronize with the others.

There is one caveat with STSM and function calls. Consider the case that one thread takes a function call and the other does not (the function call is within a conditional code block). Assume the function code has a larger PC than any other threads' PC, i.e. it is a shared library or is placed at higher memory addresses. The thread that takes the function call will have a larger PC than the other thread. Thus, it will stall the thread in the function call until the other thread reaches a PC larger than it. This may never occur, and the thread may wait in the function until the other thread completes. Ideally, we would like to execute the larger PC in this case, however this is at odds with STSM. Thus, in order to handle this case, we set a threshold value for STSM. If the difference between PCs is outside this threshold, STSM resorts to fine-grain multithreading. Only when the PCs are inside this threshold does STSM operate as described previously. Thus, in the case of the function call, STSM will alternate executing instructions from both threads, until the thread that took the function call returns, in which the PCs should be within the threshold again. Choosing the value for this threshold is somewhat ad hoc, but empirically 100 has proved to be a reasonable value and is used in the PULSA processor.

Another problem STSM has is related to drafting different programs or different versions of the same program. In these cases, if two threads have the same PCs, they do not necessarily point to the same instruction. This means STSM is actively working against drafting instructions and does not synchronize the threads. This reliance on PCs for synchronization is a shortcoming of STSM. We verify that instructions are the same in our results section before counting them as drafted. The PULSA processor contains a configuration bit to indicate if the threads have identical code.

2.5.2 Random Thread Synchronization Method

The random thread synchronization method (RTSM) is a simple mechanism. When RTSM encounters a divergence in the control flow of threads, it chooses a thread to execute at random until threads are synchronized again.

Pseudo-randomness for selecting a thread at random in RTSM is introduced using a linear feedback shift register (LFSR). The LFSR shown in Figure 2.15 is used in the

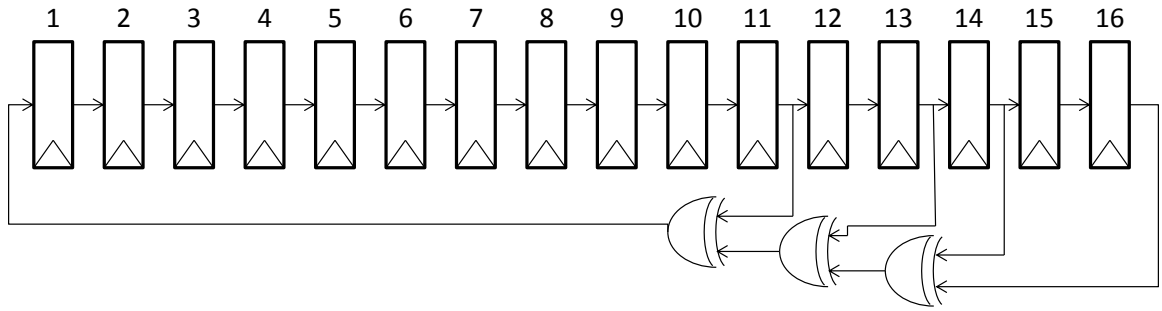


Figure 2.15: **RTSM LFSR**

PULSA processor. The 16-bit LFSR uses a polynomial such that the LFSR cycles through all possible states, besides the all zero state ($2^{16} - 1$), before repeating its sequence. This means it is a maximal LFSR and provides sufficient psuedo-randomness. The figure is a slight simplification as the LFSR has the ability to seed and read the state bits.

2.6 Load Store Unit

The load store unit (LSU) processes memory referencing operation codes (opcodes) such as various types of loads, various types of stores, cas, swap, ldstub, flush, prefetch, and membar. The LSU interfaces with all of the SPARC core functional units, and acts as the gateway between the SPARC core units and the CCX. Through the CCX, data transfer paths can be established with the memory subsystem and the I/O subsystem (the data transfers are done with packets).

The threaded architecture of the LSU can process four loads, four stores, one fetch, one FP operation, ~~one stream operation~~, one interrupt, and one forward packet. Therefore, ~~thirteen~~**twelve** sources supply data to the LSU.

The LSU implements the ordering for memory references, whether locally or not. The LSU also enforces the ordering for all the outbound and inbound packets.

An L1.5 Cache (Section 2.7) and transducer (Section 2.8) sits between the LSU and the NOCs. These structures translate outbound CCX packets to the NOC packet format and inbound NOC packets to the CCX packet format. This minimizes changes to the interface between the core and the interconnect fabric.

2.6.1 LSU Pipeline

There are four stages in the LSU pipeline. Figure 2.16 shows the different stages of the LSU pipeline.

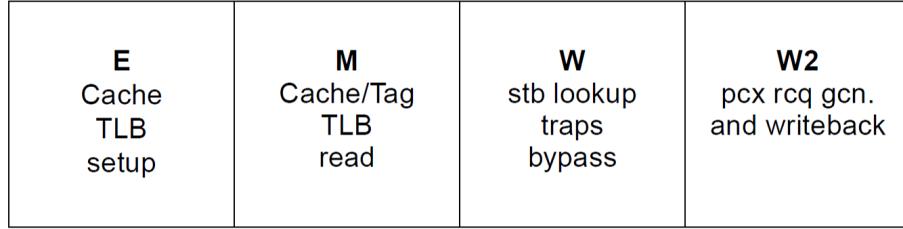


Figure 2.16: LSU Pipeline Graph

The cache access set-up and the translation lookaside buffer (TLB) access set-up are done during the pipelines E-stage (execution). The cache/tag/TLB read operations are done in the M-stage (memory access). The W-stage (writeback) supports the look-up of the store buffer, the detection of traps, and the execution of the data bypass. The W2-stage (writeback-2) is for generating PCX requests and writebacks to the cache.

2.6.2 Data Flow

The LSU includes an 8-Kbyte D-cache, which is a part of the level 1 cache shared by ~~four~~^{two} threads. There is one store buffer (STB) per thread. Stores are total store ordering (TSO) ordered, that is to say that no membar #sync is required after each store operation in order to maintain the program order among the stores. Non-TSO compliant stores include blk-store and blk-init. Bypass data are reported asynchronously, and they are supported by the bypass queue.

Load misses are kept in the load miss (LSM) queue, which is shared by other opcodes such as atomics and prefetch. The LSM queue supports one outstanding load miss per thread. Load misses with duplicated physical addresses (PA) will not be sent to the level 2 (L2) cache.

Inbound packets from the CCX are queued and ordered for distribution to other units through the data fill queue (DFQ).

The DTLB is fully associative, and it is responsible for the address translations. All CAM/RAM translations are single-cycle operations.

The ASI operations are serialized through the LSU. They are sequenced through the ASI queue to the destination units on the chip.

Figure 2.17 illustrates the LSU data flow concept.

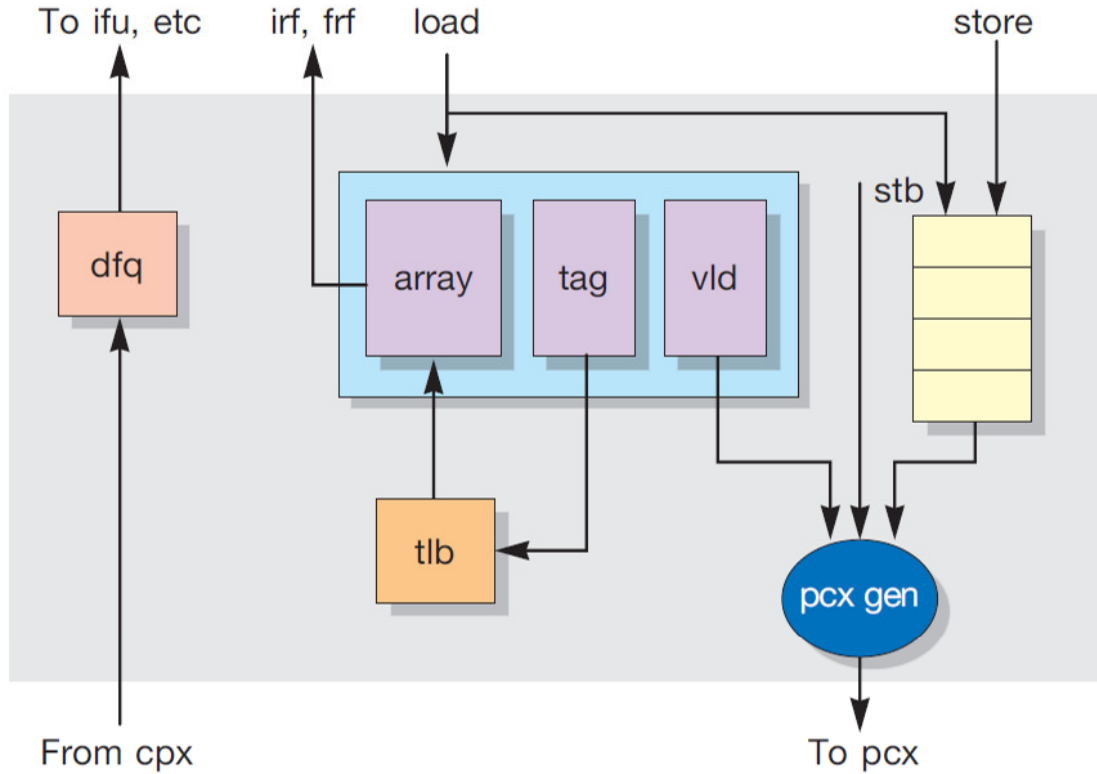


Figure 2.17: LSU Data Flow Concept

2.6.3 Level 1 Data Cache (D-Cache)

The 8-Kbyte level 1 (L1) D-cache is 4-way set-associative, and the line size is 16 bytes. The D-cache has a single read and write port (1 RW) for the data and tag array. The valid bit (V-bit) array is dual ported with one read port and one write port (1R/1W). The valid bit array holds the cache line state of valid or invalid. Invalidations access the V-bit array directly without first accessing the data and tag array. The cache line replacement policy follows a pseudo-random algorithm, where loads are allocating and stores non-allocating.

A cacheable load-miss will allocate a line, and it will execute the write-through policy for stores. Stores do not allocate, and local stores may update the L1 D-cache if it is present in the L1 D-cache, as determined by L2 (Level 2) cache directory. If it is deemed that it is not present in L1 D-cache, the local stores will cause the lines to become invalidated. The line replacement policy is pseudo random based on a linear shift register. The data from the bypass queues will be multiplexed into the L1 D-cache in order to be steered to the intended destination. The D-cache supports up to four simultaneous invalidates from the data evictions.

The L2-cache is always inclusive of the L1 D-cache. The exclusivity of the D-cache dictates that a line present in the L1 D-cache will not be present in the L1 I-cache. The data valid array is dual ported with one read port and one write port (1R1W).

Each line in the L1 D-cache is parity protected. A parity error will cause a miss in the L1 D-cache which, in turn, will cause the correct data to be brought back from the L2-cache.

In addition to the pipeline reads, the L1 D-cache can also be accessed by way of diagnostic ASI operations, BIST operations, and RAMtest operations through the test access port (TAP).

2.6.4 Data Translation Lookaside Buffer

The data translation lookaside buffer (DTLB) is the TLB for the D-cache. The DTLB caches up to the ~~64~~**8 or 16 (TODO)** most-recently-accessed translation table entries (TTE) in a fully associative array. The DTLB has one CAM port and one read-write port (1 RW). All ~~four~~**two** threads share the DTLB. The translation table entries of each thread are kept mutually exclusive from the entries of the other threads.

The DTLB supports the following 32-bit address translation operations:

- VA ->PA [virtual address (VA) to physical address (PA) translation]
- VA = PA [address bypass for hypervisor mode operations]
- RA ->PA [Real Address (RA) to Physical Address (PA) bypass translation for supervisor mode operations]

The TTE tag and the TTE data are both parity protected and errors are uncorrectable. TTE access parity errors for load instructions will cause a precise trap. TTE access parity errors for store instructions will cause a deferred trap (that is, the generation of the trap will be deferred to the instruction following the store instruction). However, the trap PC delivered to the system software still points to the store instruction that encountered the parity error in the TTE access. Therefore, the deferred action of the trap generation will still cause a precise trap from the system software perspective.

2.6.5 Store Buffer

The physical structure of the store buffer (STB) consists of a store buffer CAM (SCM) and a store buffer data array (STBDATA). Each thread is allocated with eight fixed entries in the shared data structures. The SCM has one CAM port and one RW port, and the STBDATA has one read (1R) port and one write (1W) port. All stores reside in the store buffer until they are ordered following a total store ordering (TSO) model and have updated the L1D (level 1 D-cache). The lifecycle of a TSO compliant store follows these four stages:

1. Valid
2. Commit (issued to L2-cache)
3. Acknowledged (L2-cache sent response)
4. Invalidated or L1D updated

Non-TSO complaint stores, such as blk-init and other flavors of bst (block store), will not follow the preceding life-cycle. A response from the L2-cache is not required before releasing the non-TSO complaint stores from the store buffer. Atomic instructions such as CAS, LDSTUB, and SWAP, as well as flush instructions, can share the store buffer.

The store buffer implements partial and full read after write (RAW) checking. Full-RAW data will be returned to the register files from the pipe. Partial RAW hits will force the load to access the L2-cache while interlocked with the store issued to the CCX. Multiple hits in the store buffer will always force access to the L2-cache in order to enforce data consistency.

If a store hits any part of a quad-load (16-byte access), the quad-load checking will force the serialization of the issue to the CCX. This forced serialization enforces that there will be no bypass operation.

Instructions such as a blk-load (64-byte access) will not detect the potential store buffer hit on the 64-byte boundary. The software must guarantee the data consistency using membar instructions.

2.6.6 Load Miss Queue

The load miss queue (LMQ) contains ~~four~~**two** entries in its physical structure, and the queue supports up to one load miss per thread. Instructions similar to load (such as atomics and prefetches) may also reside in the load miss queue. A load instruction speculates on a D-cache miss to reduce the latency in accessing the CCX. The load instruction may also speculate on the availability of a queue entry in the CCX. If the speculation fails, the miss-speculated load instruction can be replayed out of LMQ.

Load requests to the L2-cache from different addresses can alias to the same L2-cache line. Primary versus secondary checking will be performed in order to prevent potential duplication in the L2-cache tags.

The latencies for completing different load instructions may differ (for example, a quad-load fill will have to access integer register file (IRF) twice). The LMQ is also leveraged by other instructions. For example, the first packet of a CAS instruction will be issued out of the store buffer while the second packet will be issued out the LMQ.

2.6.7 Processor to Crossbar Interface Arbiter

The processor-to-crossbar interface (PCX) is the interface between the processor and the CCX. The arbiter takes on ~~13~~12 sources to produce one arbitrated output in one cycle. The ~~13~~12 sources include four load-type instructions, four store-type instructions, one instruction cache (I-cache) fill, one floating-point unit (FPU) access, ~~one stream processing unit (SPU) access~~, one interrupt, and one forward-packet. The ~~13~~12 sources are further divided into four categories of different priorities. The I-cache miss handling is one category. The load instructions (one outstanding per thread) are in one category. The store instructions (one outstanding per thread) are in another category. The rest of accesses are lumped into one category, and include the FPU access, ~~SPU access~~, interrupt, and the forward-packet.

The arbitration is done within the category first and then among the other categories. An I-cache fill is at the highest priority, while all other categories have an equal priority. The priorities can be illustrated in this order:

1. I-cache miss
2. Load miss
3. Stores
4. FPU operations, ~~SPU operations~~, Interrupts

The use of a two-level history allows a fair, per-category scheduling among the different categories. The arbiter achieves a resolution in every cycle. Requests from atomic instructions take two cycles to finish the arbitration.

~~There are five possible targets, which include four L2-cache banks and one I/O buffer (IOB). The FPU access shares the path through the IOB.~~

Speculation on the PCX availability does occur, and a history will be established once the speculation is known to be correct.

2.6.8 Data Fill Queue

A SPARC core communicates with memory and I/O using packets. The incoming packets, destined to a SPARC core, are queued in the data fill queue (DFQ) first. These packets can be acknowledgement packets or data packets from independent sources. The DFQ maintains a predefined ordering requirement for all the inbound packets. The targets for the DFQ to deliver the packets to include the instruction fetch unit (IFU), load store unit (LSU), and trap logic unit (TLU), ~~and stream processing unit (SPU).~~

A store to the D-cache is not allowed to bypass another store to the D-cache. Store operations to different caches can bypass each other without violating the total store ordering (TSO) model.

Interrupts are allowed to be delivered to TLU only after all the prior invalidates have been visible in their respective caches. An acknowledgement to a local I-flush is treated the same way as an interrupt.

~~Streaming stores will be completed to the D-cache before the acknowledgement is sent to the SPU.~~

2.6.9 ASI Queue and Bypass Queue

Certain SPARC core internal alternate space identifier (ASI) accesses, such as the long latency MMU ASI transactions and all IFU ASI transactions, are queued in the ASI queue. The ASI queue is a FIFO that supports one outstanding ASI transaction per thread. For all read-type ASI transactions, regardless whether they originated from the LSU or not, must have their the return data routed through the LSU and be delivered to the register file by way of the bypass queue.

The bypass queue handles all of the load reference data, other than that received from the L2-cache, that must be asynchronously written to the integer register file (IRF). This kind of read data includes full-RAW data from the store buffer, ldxa to the internal ASI data, store data for casa, a forward packet for the ASI transactions, as well as the pending precise traps.

2.6.10 Alternate Space Identifier Handling in the Load Store Unit

In addition to sourcing alternate space identifier (ASI) data to other functional units of a SPARC core, the load store unit (LSU) decodes and supports a variety of ASI transactions, which include:

- Defining the behavior of ld/st ASI transactions such as blk-ls, blk-st, quad-ASI, and so on
- Defining an explicit context for address translation at all levels of privilege, such as primary, secondary, as_if_user, as_if_supv, and so on
- Defining special attributes, such as non_faulting and endianness, and so on
- Defining address translation bypassed, such as [RA=PA], [VA=PA], and so on, where VA stands for virtual address, PA stands for physical address, and RA stands for real address

2.6.11 Support for Atomic Instructions (CAS, SWAP, LDSTUB)

CAS is issued as a two-packet sequence to the Processor to L2-cache Interface (PCX). Packet 1 contains the compare address (rs1) and the data (rs2). Packet 2 contains the swap data (rd). Packet 1 resides in the store buffer in order to be compliant to the TSO ordering, while Packet 2 occupies the threads entry into the load miss queue (LMQ).

Packet 1 and Packet 2 are issued in back-to-back order to the PCX. An acknowledgement to the load is returned to the CPX in response to Packet 1. This acknowledgement contains the data in memory from the address-in (rs1). An acknowledgement to the store is returned on the CPX in response to Packet 2. This acknowledgement will cause an invalidation at address-in (rs1) if the cache line is present in the level 1 D-cache.

SWAP and LDSTUB are single packet requests to the PCX, and they reside in the store buffer.

2.6.12 Support for MEMBAR Instructions

MEMBAR instructions ensure that the store buffer of a thread has been drained before the thread gets switched back in. The completion of draining the store buffer implies that all stores prior to the MEMBAR instruction have reached a global visibility, in compliance with TSO ordering. Before a MEMBAR is released, it ensures

that all blk-init and blk-st instructions have also reached global visibility. This is accomplished by making sure that st-ack counter has been cleared.

There are several flavors of MEMBAR instructions. The implementation for #store-store, #loadstores, and #loadload is to make them behave like NOPs. The implementation for #storeload, #memissue, and #lookaside is to make them to behave like #sync. membar #sync is fully implemented to help enforce the compliance to TSO ordering.

A parity error on a store to the DTLB will cause a deferred trap. It will be reported on the follow-up membar #sync. The trap PC in this case will point to the store instruction encountering the parity error when storing to the DTLB. The deferred trap will look like a precise trap to the system software because of the way the hardware supports the recording of the precise trap PC.

2.6.13 Core-to-Core Interrupt Support

A core-to-core interrupt is initiated by a write to the interrupt dispatch register IINT_VEC_DIS ASI) in Trap Logic Unit (TLU). It will generate a request to LSU for access to PCX. LSU only supports one outstanding interrupt request at any time. An interrupt is treated similar to a membar. It will be sent to PCX once the store buffer of the corresponding thread has been drained. This interrupt will then immediately be acknowledged to TLU.

After the interrupt packet has been dispatched by way of the L2-cache to Core Interface (CCX), the packet would be executed on the destination thread of a SPARC core. It can be invalidated after all prior invalidates have completed and results arrived at L1 D-cache (L1D).

2.6.14 Flush Instruction Support

A flush instruction does not actually flush the instruction memory. It instead, it acts as a barrier to ensure that all of the prior invalidations for a thread have been visible in the level 1 I-cache (L1I) before causing the thread to be switched back in. The flush is issued as an interrupt with the flush bit set, which causes the L2-cache to broadcast the packet to all SPARC cores.

For the SPARC core that issued the flush, an acknowledgement from the DFQ upon receiving the packet will cause all of the prior invalidations to complete with the results arrived at the level 1 I-cache and the level 1 D-cache (L1 I/D).

For the SPARC cores that did not issue the flush, the DFQ will serialize the flushes so that the order of the issuing threads actions, relative to the flushes, will be preserved.

2.6.15 Prefetch Instruction Support

A prefetch instruction is treated as a non-cacheable load. A prefetch that misses in the TLB, or accesses I/O space, will be treated as a NOP. The issuing thread will be switched back in without accessing the processor to L2-cache interface (PCX). The LSU supports a total of eight outstanding prefetch instructions across all ~~four~~**two** threads. The LSU keeps track of the number of outstanding prefetches per thread, which limits the number of outstanding prefetches.

2.6.16 Floating-Point BLK-LD and BLK-ST Instructions Support

Floating-point blk-ld and blk-st instructions are non-TSO compliant. Only one outstanding blk-ld or blk-st instruction is allowed per SPARC core. These instructions will bypass the level 1 caches and will not allocate in the level 1 caches either. On a level 1 D-cache (L1D) hit, a blk-st instruction will cause an invalidation to the L1D. Both blk-st and blk-ld instructions can access the memory space and the I/O space.

The LSU breaks up a the 64-byte packet of a blk-ld instruction into four of 16-byte load packets so that they can access the processor and L2-cache interface (PCX). The Level 2 cache returns four of the 16-byte packets, which in turn, will cause eight of 8- byte data transfers to the floating-point register file (FRF). Errors are reported on the last packet. A blk-ld instruction could cause a partial update to the FRF. Software must be written to retry the instruction later.

A blk-st instruction will be unrolled into eight helper instructions by the floating-point functional unit (FFU) for a total of a 64-byte data transfer. Each 8-byte data gets an entry of the corresponding thread in the store buffer. The blk-st instructions are non-TSO compliant, so the software must do the ordering.

2.6.17 Integer BLK-INIT Loads and Stores Support

The blk-init load and blk-init store instructions were introduced as the substitute for blk-ld and blk-st in block-copy routines. They can access both the memory space and the I/O space. The blk-init loads do not allocate in the level 1 D-cache. On a level 1 D-cache hit, the blk-init stores will invalidate the level 1 D-cache (L1D).

The blk-init load instructions must be quad-word accesses, and violating this rule will cause a trap. Like quad-load instructions, blk-init loads also send double-pump writes (8-byte access) to the integer register file (IRF) when a blk-init load packet reaches the head of the data fill queue (DFQ).

The blk-init stores are also non-TSO compliant, which allows for greater write throughput and higher-performance yields for the block-copy routine.

Up to only eight of all non-TSO compliant instructions can be allowed outstanding for each SPARC core. The LSU keeps a counter per thread to enforce this limit.

2.6.18 ~~STRM Load and STRM Store Instruction Support~~

~~Instructions such as strm-ld and strm-st make requests from the stream processing unit (SPU) to memory by way of the LSU.~~

~~The Store buffer will not be looked-up by the strm-ld instructions, and the store buffer will not buffer strm-st data. Software must be written to enforce the ordering and the maintenance of the data coherency.~~

~~The acknowledgements for strm-st instructions will be ordered through the data fill queue (DFQ) upon the return to the stream processing unit (SPU). The corresponding store acknowledgement (st ack) will be sent to the SPU once the level 1 D-cache (L1D) invalidation, if any, has been completed.~~

2.6.19 ~~Test Access Port Controller Accesses and Forward Packets Support~~

~~TODO: Will we have this?~~

~~Test access port (TAP) controller can access any SPARC core by way of the SPARC interface of the I/O bridge (IOB). A forward request to the SPARC core might take any of the following actions:~~

- ~~• Read or write level 1 I-cache or D-cache~~
- ~~• Read or write BIST control~~
- ~~• Read or write margin control~~
- ~~• Read or write de-feature bits to the de-feature any, or all, of L1I, L1D, ITLB, DTLB~~

~~in order to take a cache off-line, or a TLB offline, for diagnostic purposes A forward reply will be sent back to the I/O bridge (IOB) once the data is read or written. A SPARC core might further forward the request to the L2-cache for an access to the control status register (CSR). The I/O bridge only supports one outstanding forward access at any time.~~

2.6.20 SPARC Core Pipeline Flush Support

A SPARC core pipeline flush is reported through the LSU since the LSU is the source of the latest traps in the pipeline.

The trap logic unit (TLU) gathers traps from all functional units except the LSU, and it then sends them to the LSU. the LSU performs the or function for all of them (plus its own) and then it broadcasts across the entire chip.

The LSU can also send a truncated flush for the internal ASI ld/st to the TLU, the MMU, and the SPU.

2.6.21 LSU Error Handling

Errors can be generated from any, or all, of the following memory arrays DCACHE (D-cache), D-cache tag array (DTAG), D-cache valid bit array (DVA), DTLB, data fill queue (DFQ), store buffer CAM array (SCM), and store buffer data array (STB-DATA). Only the DCACHE, DTAG, and DTLB arrays are parity protected.

- A parity error on a load reference to the DCACHE will be corrected by way of the reloading the correct data from the L2-cache as if there were a D-cache miss.
- A DTAG parity error will result in a correction packet, followed by the actual load request, to the L2-cache. The correction packet synchronizes the L2 directory and the L1 D-cache set. On the load request acknowledgement, the level 1 D-cache will be filled.
- A parity error on the DTLB tte-data will cause an uncorrectable error trap to the originating loads or stores.
- A parity error on the DTLB tte-data can also cause an uncorrectable error trap for ASI reads.
- A parity error on the DTLB tte-tag can only cause an uncorrectable error trap for ASI reads.

2.7 L1.5 Data Cache

The L1.5 is a private cache—directly below the SPARC core L1 data cache (L1D)—providing two necessities: (1) write-back capability to the otherwise write-through L1 cache, and (2) compliance to share-memory protocol of the rest of the system.

The L1.5 is located outside of the core. It communicates with the L1 through the PCX/CPX bus (request/response), and with the distributed L2 through three network-on-chips (NoC1/2/3). It is called L1.5 because, historically in Piton, the cache is sized identically to L1D and thus does not provide additional private data-store. If it was sized larger, it is more correct to call it a private L2. In Piton it is configured to be 8KB in size, with 16-byte cache lines, 4-way set-associative, inclusive of L1D, and is indexed with physical address.

2.7.1 Write-back functionality

Providing write-back functionality, to the core and just before the NoC communication channels, was the primary reason for the L1.5 design. If write-back wasn't available, the high acknowledgment latency and bandwidth usage of stores—every single store—would be detrimental to the core performance. For instance, suppose a 1024 core processor, and that core 0 is repeatedly writing to cache lines homed at core 1023, the round-trip latency for the store and the acknowledgment is 128 cycles. Since the T1 core is TSO-compliant, it cannot issue the next entry in the store-buffer until it receives the acknowledgment for the current store. Round-trip latency further worsens when the home-tile for the cache line is located off-chip.

There were numerous options for adding a write-back buffer to the L1D, including modifying the core RTL to convert write-through to write-back; in the end, the Piton team decided to interpose another level of cache hierarchy, completely decoupled from the L1D, to support write-back. For a clean interface separation, the L1.5 leverages the PCX/CPX request/response bus protocol to communicate with the core. With this approach we minimized the modifications to the core's RTL.

With a write-back and MESI-coherent cache right before the NoC (the L1.5), writes from the core are easily converted to be write-back. Take the previous example where core 0 is repeatedly storing to a cache line, if the L1.5 contains the cache line in state Modified, then the store ack will only take the pipeline length (3 cycles in L1.5) to be returned; if L1.5 does not have it or does not have the right coherence state, then a one-time request would be made to the L2. In any case, subsequent writes are coalesced, minimizing latency and NoC bandwidth usage.

2.7.2 Way-map table for L1D cache coherence

The way-map table (WMT) is critical to support two important cache coherence operations: store acknowledgment and cache line invalidation. The originating reason is because the T1's L1D does not support invalidation (and store-ack) by memory address, rather by precise set index and way in the L1D that contains the cache line. Supposedly this design decision is made to increase L1D's performance, or to remove redundant functionality done by the T1's L2. As implied, the T1's L2 keeps a directory with copies of all tags in all L1Ds, so that on an coherent invalidation (or a store) it can also look up the precise index/way. The L1.5 satisfies this requirement not with a tag directory but with the WMT.

Substituting the tag directory with the LMT is possibly by leveraging an interesting fact: that the L1.5 has the same size as the L1D, with identical set count and set-associativity. The invariant then can be stated as followed: if a cache line \underline{x} is present in a set \underline{n} in the L1D, then it also must be present in the same set \underline{n} in the L1.5. However, there is no guarantee as for way allocation for \underline{x} between the L1D and L1.5, and thus a table is needed to keep this information.

The LMT maps a cache way from the L1.5 to a cache way of the same set in the L1D, possibly invalid (when line exists in L1.5 but not in L1D). The LMT then has the same number of sets and ways as the data cache (128 sets, 4 entries each), and is indexed in the same manner; each entry in the set contains a valid bit and the way value. L1.5 accesses the LMT entry on a cache coherence operation (invalidation or write), and updates it when the L1D allocates (and/or evicts) cache line entries. Figure xx shows an example of the mappings stored in the WMT.

One important detail of the WMT operation is that read-modify-write is necessary when the L1D allocates and deallocates a cache entry.

Since LMT is designed by leveraging how L1.5 is sized the same as L1D, it is not possible to scale up/down the L1.5 through changing the number of sets; it is still relatively easy to scale by increasing or decreasing the set-associativity though. Having said that, a duplicate of L1D tag directory is not needed. The WMT can be augmented with a L1D to L1.5 way-map table to support the needed operations. However, modifications to the L1.5 pipeline are likely needed.

2.7.3 Write-buffer

A write-buffer-16B (cache block size), one per thread¹—was necessitate by the use of allocation-on-fill instead of allocation-on-miss allocation policy.² The write-buffer captures the store, which can be any size from 1B to 8B, then merges this data when the filling cache line comes back from the L2.

¹T1 can issue one outstanding store per thread.

²Changing the allocation policy will likely affect the deadlock-free coherence protocol.

In practice, the implementation is more complex and prone to bug. There are two primary reasons for this. First, stores from thread A can alias to a cache line of another in-flight store from thread B. For correctness, the L1.5 needs to do a full-tag comparison between the two stores (in stage S1), and update (or coalesce write) the store-buffer entry accordingly. Second, in the corner-case where the in-flight store is asking for write-permission from L2 (upgrading MESI state from S to M), and somehow the cache line (in S state) is evicted due to another fill from the L2, some metadata are needed to guarantee correctness. The correctness question here is whether the store acknowledgment to the L1D should set the modify-bit or not, where this bit indicates whether the L1D contain the cache line and should be updated. If the cache line was in S state, was not evicted, and is present in L1D, then the bit must be set; otherwise the bit is not set.

2.7.4 Handling L1I

Instruction cache lines are actually not cached in the L1.5; regardless they still need to be cache coherent. The L1.5 does not have any special data structure to keep track of the L1I, rather, it delegates the instruction cache line tracking to the L2, and only forwards data and other coherence operations (like invalidation) to the core through the CPX bus.

2.7.5 Handling special loads and stores

2.7.5.1 Block loads/stores

With these non-TSO loads and stores, the T1 can issue up to 4 consecutive requests in-flight. However, due to the complexity with handling these in a write-back manner, the L1.5 treats these requests as if they're regular requests. Specifically for stores, the L1.5 will stall the PCX buffer until the current store is finished.

2.7.5.2 Prefetch loads

The L1.5 treats prefetches as if they're regular loads. However, while the fill allocates in the L1.5, it does not in the L1D, as expected. There have been efforts to optimize for prefetch loads—for example, returning prefetch-ack immediately—but complexities with deadlock-free coherent protocol and with manycore cache coherent scheme compelled the simplest design decision.

2.7.5.3 Non-cacheable loads/stores

Non-cacheable (NC) loads/stores are usually to I/O address space, which then has b39 set to 1. The L1.5 generally handles these requests by

1. Checks for cache hit and evict if present
2. Forwards the request to L2

Possibly step 1 is not necessary, and that only a simple forwarding is needed, but it has been observed that NC loads to non-I/O addresses are possible, and so step 1 is always done as a safe-guard.

2.7.5.4 CAS/SWP/LOADSTUB

These requests are handled in the same manner as NC loads/store; that is, evict cache line then forward requests to L2. The returned data from L2 are not cached in neither L1.5 nor L1D.

2.7.5.5 IPI packets

L1.5 primarily forwards inter-processor interrupts (IPI) for both directions: from own core to other core, and reverse. Originating from a core, the IPI passes through the PCX to L1.5, then NoC1 to the local L2, then bounces back to the target L1.5 through NoC2, and finally to the target core through CPX. Otherwise if originating from off-chip (initial wake-up message), the packet is sent to L2 through NoC1, and continues as above.

Note that Piton's modification to core-id querying mechanism, so the L1.5 clears IPI coreid field to 0x0 right before forwarding the IPI to the SPARC core.

2.7.6 Cache coherence

Piton's cache coherence scheme is based on MESI, and L1.5, conforms to it through a deadlock-free NoC-based coherence scheme. The packet format and coherence request diagram are further discussed in Section ??.

2.7.7 Interfaces

L1.5 connects to other components through these main channels: PCX/CPX, NoC1, NoC2, and NoC3. Another bus, UBC, is used for debugging.

PCX/CPX are the bus/protocol used in the T1 to connect the core to L2, and are mostly unchanged in Piton. The core issues read and write requests through the

PCX bus; there can be up to 1 read, 1 write, and 1 icache request outstanding per thread. So, in Piton configured with 2 threads, the L1.5 can receive up to 6 concurrent requests. The number of PCX buses per core is reduced from 5 to 1. In T1, the core can issue requests to 4 banks of L2, plus an IOB memory controller, and each requires a separate PCX channel. In Piton though, the only destination is the L1.5. The PCX bus is buffered by a 2-entry queue at L1.5—the minimum requirement by the PCX bus protocol to support CAS requests.

The L1.5 returns data to the core through CPX interface. Like PCX, it is mostly unmodified. The same CPX bus is multiplexed between the L1.5 and the FPU module, with priority given to the L1.5. While unnecessary, the L1.5 buffers the CPX response to optimize for critical path and thus synthesis.

Three network-on-chip (NoC) channels are used by the L1.5 and L2 to communicate. General detail of the coherence protocol is portrayed in another section. Specifically to L1.5, NoC1/3 are outputs, and NoC2 is input. To break cyclic dependencies between the channels, the L1.5 equipped the NoC1 output with a 8-entry request buffer and 2-entry 8-byte data buffer—just enough to satisfy 6 inflight requests including 2 stores from the core. NoC2 is currently configured with a 512-bit buffer, enough for the largest response from L2 (icache response); and NoC3 is with a 1-entry queue, not for correctness but for performance and relax synthesis timing requirements.

2.7.8 Testing/debugging support

There are two ways to access the L1.5 cache by index and way instead of by address. First, the software can do so through reading and writing the ASI addresses. Second, hardware debugger can access the SRAMs directly by means of JTAG to CTAP to RTAP to BIST engine.

L1.5 also filters accesses to the config register address space and forwards them accordingly. The L1.5 itself does not have any configurable register.

2.7.9 Implementation

2.7.9.1 Pipelined implementation

3 stage, + buffer stages: - message buffer: pcx + noc2; decoder - preselect buffer
decode: to internal, relevant values

L1.5's is pipelined to three stages: S1, S2, and S3. Fundamentally, S1 decodes requests from either PCX or NoC2; S2 does tag check, and issues read/write command to data SRAM; and S3 steers data and makes request to the right output buffer.

S1

- Picks a request from available buffers and decode it to internal opcodes. It gives preference to NoC2 rather than PCX to guarantee deadlock-free coherence protocol.
- Accesses the MSHR structure extensively; allocating new MSHR entries, tag-checking against existing MSHR entries, reload stored request metadata, and storing data to the write-buffer are just a few tasks that the MSHR provides in S1 stage.
- Stalls conflicted requests. These include index-lock (relaxed for some opcodes that just forward), address conflict (to avoid data race), and MSHR entry unavailability.
- Updates write-buffer for stores.

S2

- Does tag-check, and load/store to data SRAMs if condition satisfies.
- Reads to WMT table.
- Reads write-buffer if necessary.
- Updates MESI table.
- Issues command to Config Registers.

S3

- Processes WMT read to indicate whether L1D should update its cache line.
- Checks whether cache line associated with a pending store should be downgraded from S to I.
- Updates LRU table.
- Updates WMT table.
- Deallocates MSHR entry for completed requests.
- Issues requests to CPX, NoC1, and NoC3 (in any combination).

2.8 CPX-NOC Transducer

TODO

2.9 Execution Unit

The execution unit (EXU) contains these four subunits arithmetic and logic unit (ALU), shifter (SHFT), integer multiplier (IMUL), and integer divider (IDIV).

Figure 2.18 presents a top level diagram of the execution unit.

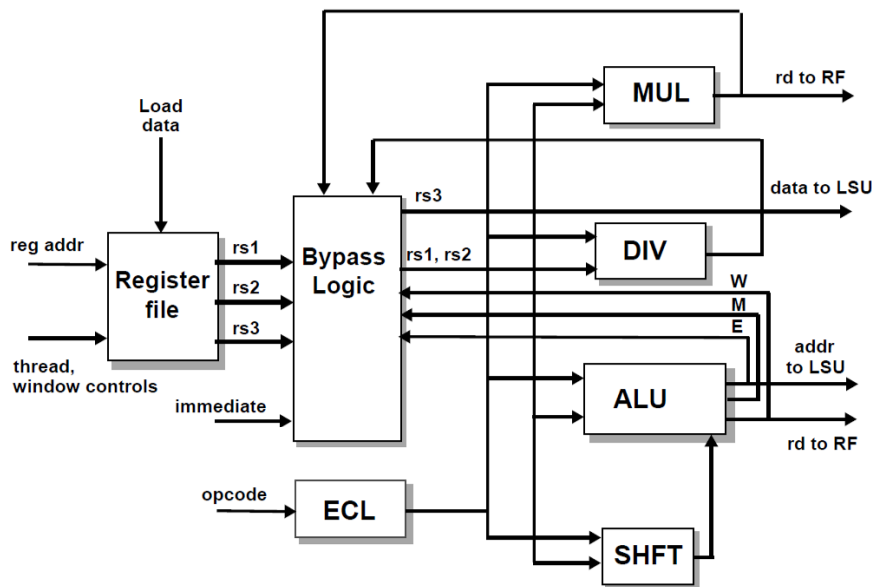


Figure 2.18: Execution Unit Diagram

The execution control logic (ECL) block generates the necessary select signals that control the multiplexors, keeps track of the thread and reads of each instruction, and implements the bypass logic. The ECL also generates the write-enables for the integer register file (IRF). The bypass logic block does the operand bypass from the E, M, and W stages to the D stage. Results of long latency operations such as load, mul, and div, are forwarded from the W stage to the D stage. The condition codes are bypassed similar to the operands, and bypassing of the FP results and writes to the status registers are not allowed.

The shifter block (SHFT) implements the 0 - 63-bit shift, and Figure 2.19 illustrates the top level block diagram of the shifter.

The arithmetic and logic unit (ALU) consists of an adder and logic operations such as ADD, SUB, AND, NAND, OR, NOR, XOR, XNOR, and NOT. The ALU is also reused when calculating the branch address or a virtual address. Figure 2.20 illustrates the top level block diagram of the ALU.

MUL is the integer multiplier unit (IMUL), and DIV is the integer divider unit (IDIV). IMUL includes the accumulate function for modular arithmetic. The latency of IMUL is 5 cycles, and the throughput is 1-half per cycle. IMUL supports one outstanding

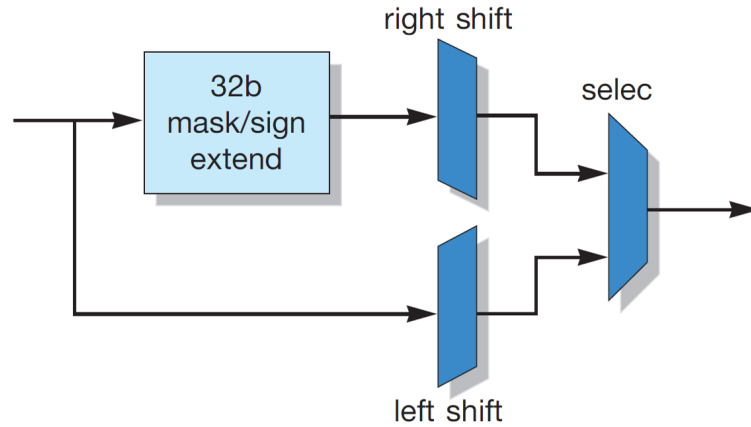


Figure 2.19: Shifter Block Diagram

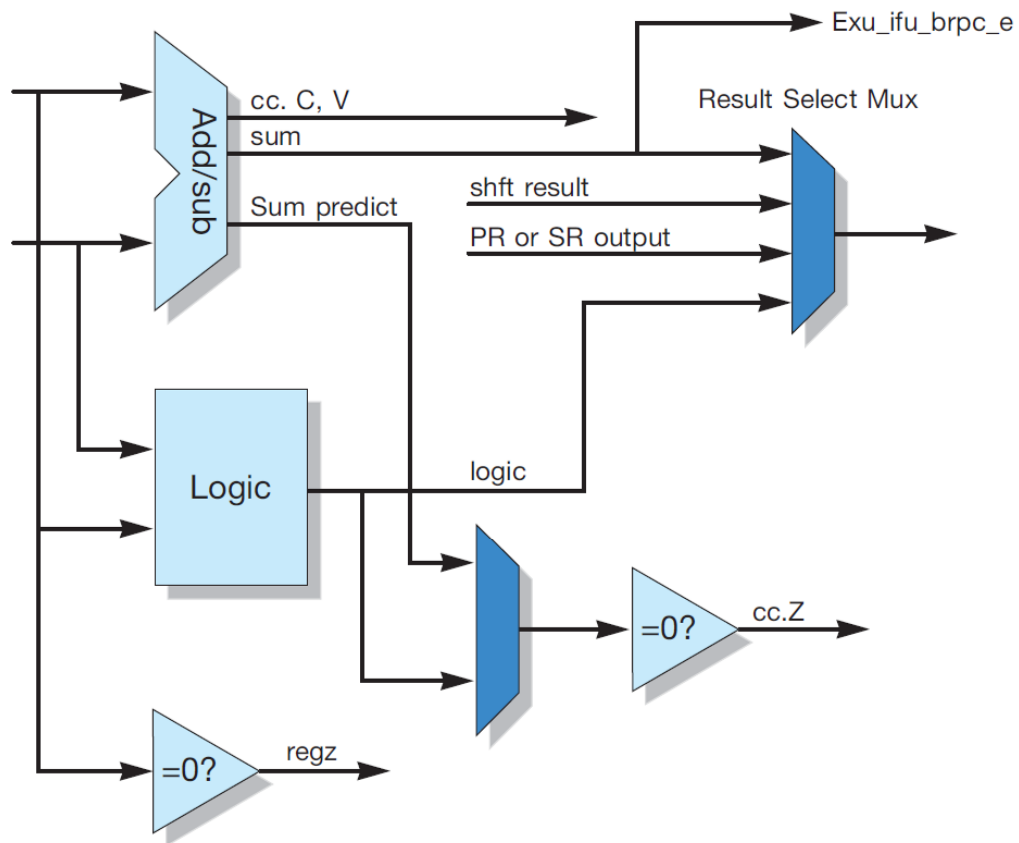


Figure 2.20: ALU Block Diagram

integer multiplication operation per core, and it is shared between a SPARC core pipeline and the modular arithmetic unit (MAU). The arbitration is based on a round-robin algorithm.

IDIV contains a simple non-restoring divider, and it supports one outstanding divide operation per core.

Figure 2.21 illustrates the top level diagram of the IDIV.

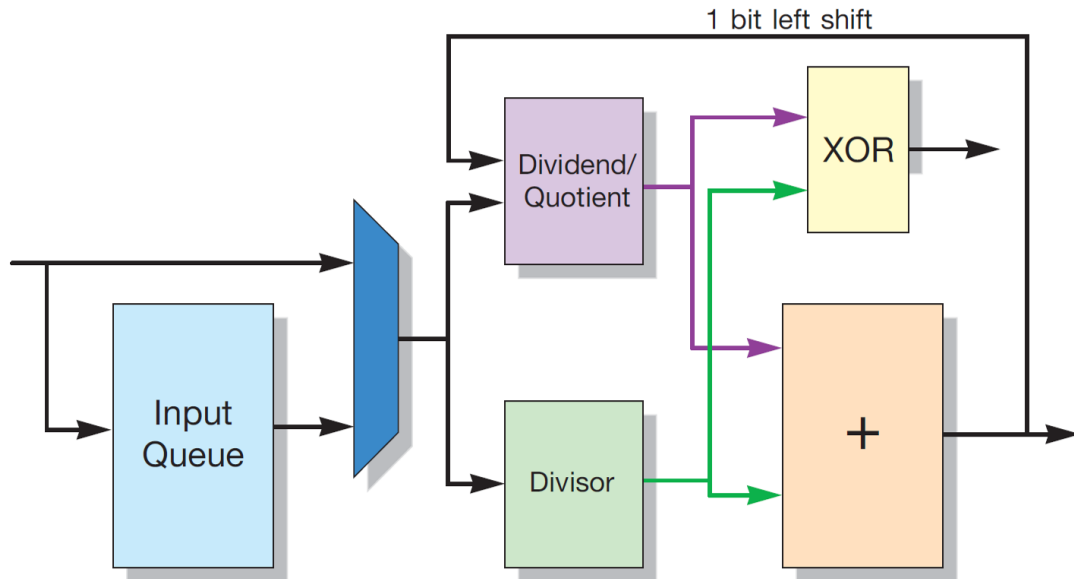


Figure 2.21: IDIV Block Diagram

When either IMUL or IDIV is occupied, a thread issuing a MUL or DIV instruction will be rolled back and switched out.

2.10 Floating-Point Frontend Unit

2.10.1 Functional Description of the FFU

The floating-point frontend unit (FFU) is responsible for dispatching floating-point operations (FP ops) to the floating-point unit (FPU) through the LSU, as well as executing simple FP ops (mov, abs, neg) and VIS instructions. The FFU also maintains the floating-point state register (FSR) and the graphics state register (GSR). There can be only one outstanding instruction in the FFU at a time.

The FFU is composed of four blocks the floating-point register file (FFU_FRF), the control block (FFU_CTL), the data-path block (FFU_DP), and the VIS execution block (FFU_VIS). Figure 2.22 shows a block diagram of the FFU illustrating these four subblocks.

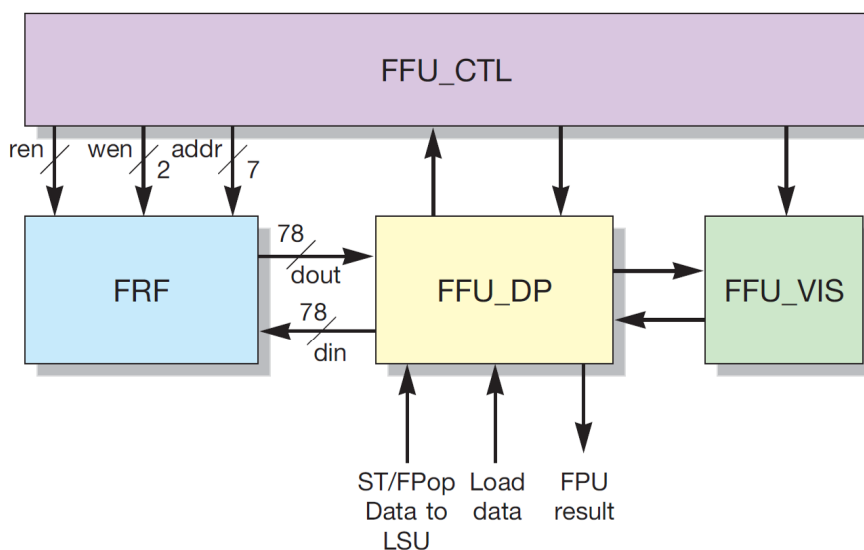


Figure 2.22: Top-Level FFU Block Diagram

2.10.2 Floating-Point Register File

The floating-point register file (FRF) has 128 entries of 64-bits of data, plus 14-bits of ECC. The write port has an enable for each half of the data. Bits [38:32] are the ECC bits for the lower word (data[31:0]) and bits [77:71] are the ECC bits for the upper word (data[70:39]).

2.10.3 FFU Control (FFU_CTL)

The FFU control (FFU_CTL) block implements the control logic for the FFU, and it generates the appropriate multiplexor selects and data-path control signals. The FFU control also decodes the fp_opcode and contains the state machine for the FFU pipeline. It also generates the FP traps and kill signals, as well as signalling the LSU when the data is ready for dispatch.

2.10.4 FFU Data-Path (FFU_DP)

This FFU data-path block contains the multiplexors and the flops for the data that has been read from, or is about to be written to, the FRF. The FFU data-path also dispatches the data for the STF and the FPops to the LSU, receives LDF from the LSU, and receives the results from the FPops from the CPX. The FFU data-path also implements FMOV, FABS, and FNEG, checks the ECC for the data read from the FRF, and generates the ECC for the data written to the FRF.

2.10.5 FFU VIS (FFU_DP)

The FFU VIS (FFU_DP) block implements a subset of the VIS graphics instructions, including partitioned addition/subtraction, logical operations, and faligndata. All the operations are implemented in a single cycle, and the data inputs and outputs are connected to the FFU_DP.

2.11 Multiplier Unit

2.11.1 Functional Description of the MUL

The SPARC multiplier unit (MUL) performs the multiplication of two 64-bit inputs. The MUL is shared between the EXU and the SPU, and it has a control block and data-path block. Figure 2.23 shows how the multiplier is connected to other functional blocks.

2.12 Stream Processing Unit

Each SPARC core is equipped with a stream processing unit (SPU) supporting the asymmetric cryptography operations (public-key RSA) for up to a 2048-bit key size.

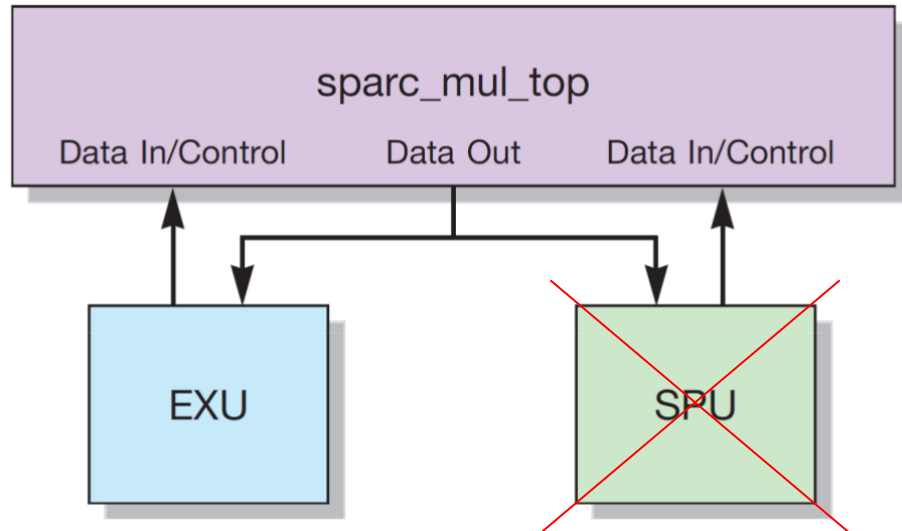


Figure 2.23: Multiplier (MUL) Block Diagram

The SPU shares the integer multiplier with the execution unit (EXU) for the modular arithmetic (MA) operations. The SPU itself supports full modular exponentiation. While the SPU facility is shared among all threads of a SPARC core, only one thread can use the SPU at a time. The SPU operation is set up by a storing a thread to a control register and then returning to normal processing. The SPU will initiate streaming load or streaming store operations to the level 2 cache (L2) and compute operations to the integer multiplier. Once the operation is launched, it can operate in parallel with SPARC core instruction execution. The completion of the operation is detected by polling (synchronous fashion) or by interrupt (asynchronous fashion).

2.12.1 ASI Registers for the SPU

All alternate space identifier (ASI) registers for the SPU are 8 bytes in length. Access to all of the ASI registers for the SPU have hypervisor privilege, so they can only be accessed in hypervisor mode. The following list highlights those ASI registers.

- Modular arithmetic physical address (MPA) register

This register carries the physical address used to access the main memory. MA_LD requests must be on the 16-byte boundary while MA_ST requests must be on the 8-byte boundary.

- Modular arithmetic memory addresses (MA_ADDR) register

This register carries the memory address offsets for various operands, and the size of the exponent. Figure 2.24 highlights the layout of the bit fields.

- Modular arithmetic N-prime value (MA_NP) register

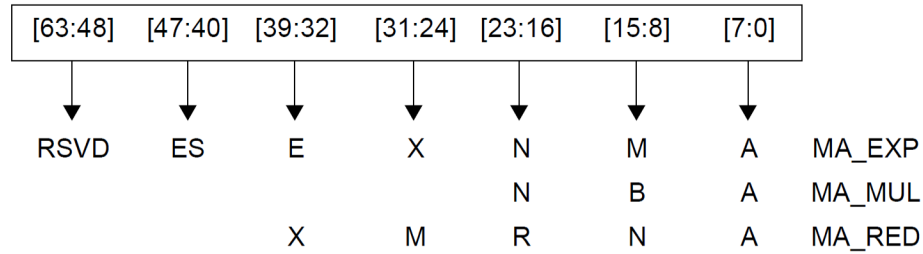


Figure 2.24: Layout of MA_ADDR Register Bit Fields

This register is used to specify the modular arithmetic N-prime value.

- Modular arithmetic synchronization (MA_SYNC) register

A load operation from this register is used to synchronize a thread with the completion of asynchronous modular arithmetic operations performed by the SPU.

- Modular arithmetic control parameters (MA_CTL) register

This register contains several bit-fielded fields that provide these control parameters:

- PerrInj Parity error injection

When this parameter is set, each operation that writes to modular arithmetic memory will have the parity bit inverted.

- Thread Thread ID for receiving interrupt

If the Int bit is set, this set of bits specifies the thread that will receive the disrupting trap on the completion of the modular arithmetic operation.

- Busy SPU is BUSY

When this parameter is set, the SPU is busy working on the specified operation.

- Int Interrupt enable

When this parameter is set, the SPU will generate a disrupting trap to the current thread on completion of the current modular arithmetic operation. If cleared, software can synchronize with the current modular arithmetic operation using the MA_Sync instruction.

The disrupting trap will use the *implementation dependent exception 20* as the modular arithmetic interrupt.

- Opcode Operation code of modular arithmetic operation (see Table 2.3)

- Length Length of modular arithmetic operations

Table 2.3: Modular Arithmetic Operations	
Opcode Value	Modular Arithmetic Operation
0	Load from modular arithmetic memory
1	Store to modular arithmetic memory
2	Modular multiply
3	Modular reduction
4	Modular exponentiation loop
5-7	Reserved

This field contains the bits for the value of the (length - 1) for the modular arithmetic operations.

2.12.2 Data Flow of Modular Arithmetic Operations

Figure 2.25 illustrates the data flow of modular arithmetic operations.

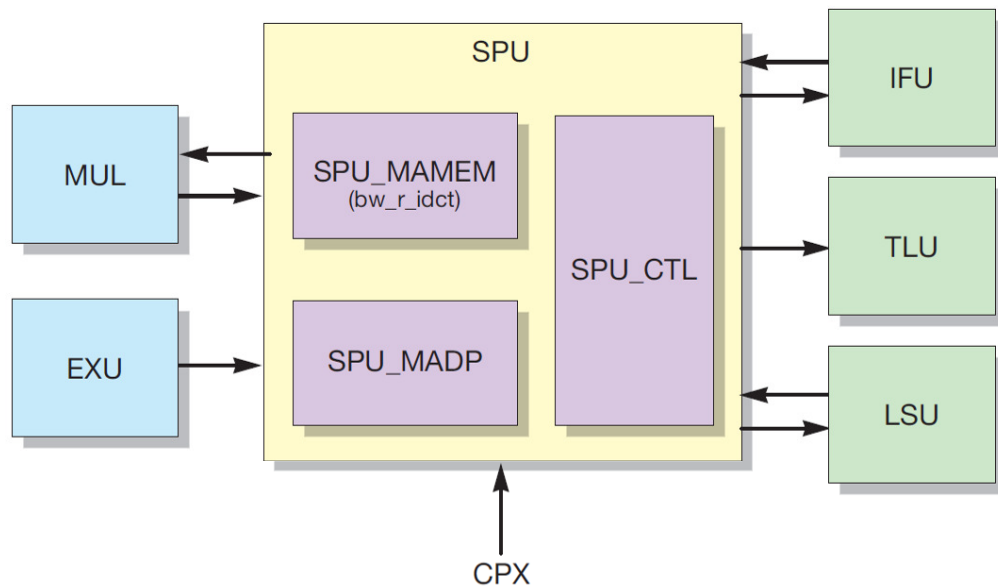


Figure 2.25: Data Flow of Modular Arithmetic Operations

2.12.3 Modular Arithmetic Memory (MA Memory)

A total of 1280 bytes of local memory with a single read/write port (1RW) is used to supply operands to modular arithmetic operations. The modular arithmetic memory (MA memory) will house 5 operands with 32 words each, which supports a maximum key size of 2048 bits. This MA memory is parity protected with a 2-bit parity for each 64-bit word.

~~MA memory requires software initialization prior to the start of MA memory operations. Three MA_LD operations are required to initialize all 160 words of memory because the MA_CTL length field allows up to 64 words to be loaded into MA memory.~~

~~Write accesses to the MA memory can be on either the 16-byte boundary or the 8-byte boundary. Read accesses to the MA memory must be on the 8-byte boundary.~~

2.12.4 Modular Arithmetic Operations

All modular arithmetic registers must be initialized prior to launching a modular arithmetic operation. Modular arithmetic operations (MA ops) start with a stx to the MA_CTL register if the store buffer for that thread is empty. Otherwise, the thread will wait until the store buffer is emptied before sending stx.ack to the LSU. An MA operation that is in progress can be aborted by another thread by way of a stx to the MA_CTL register.

An ldx to MA registers are blocking. All except ldx to the MA_Sync register will respond immediately. An ldx to the MA_Sync register will return a 0 to the destination register upon the operation completion. The thread ID of this ldx should be equal to that stored in the thread ID field of the MA_CTL register. Otherwise, the SPU will respond immediately and send signals to the LSU to not update the register file. In case of aborting an MA operation, the pending ldx to MA_Sync is unblocked, and the SPU signals the LSU will not update the register file.

Figure 2.26 illustrates the MA operations using a state transition diagram.

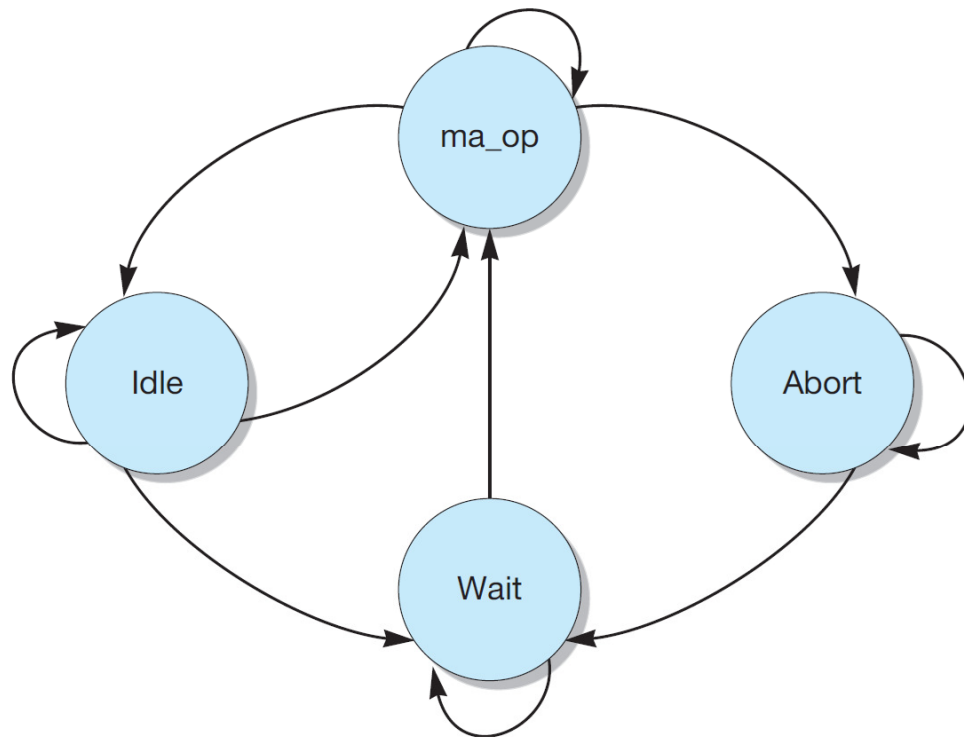


Figure 2.26: State Transition Diagram Illustrating MA Operations

The state transitions are clarified by the following set of equations:

```

tr2_maop_frm_idle = cur_idle & stxa_2ctlreq & ~wait_4stb_empty & ~wait_4trapack_set;
tr2_abort_frm_maop = cur_maop & stxa_2ctlreg;
tr2_wait_frm_abort = cur_abort & ma_op_complete;
tr2_maop_frm_wait = cur_wait & ~(stxa_2ctlreg | wait_4stb_empty | wait_4trapack_set);
tr2_idl_frm_maop = cur_maop & ~stxa_2ctlreg & ma_op_complete;
tr2_wait_frm_idle = cur_idle & stxa_2ctlreg & (wait_4stb_empty | wait_4trapack_set);

```

A MA-ST operation is started with a stxa to the MA_CTL register opcode equals the MA-ST, and the length field specifies the number of words to send to the level 2 cache (L2-cache). The SPU sends a processor to cache interface (PCX) request to the LSU and waits for an acknowledgement from the LSU prior to sending another request. If needed, store acknowledgements, which are returned from the L2-cache on level 2 cache to processor interface (CPX), will go to the LSU in order to invalidate the level 1 D-cache (L1D). The LSU will then send the SPU an acknowledgement. The SPU then decrements a local counter and waits for all the stores sent out to be acknowledged and transitioned to the done state.

On a read from the MA Memory, the operation will be halted if a parity error is encountered. The SPU waits for all posted stores to be acknowledged. If the Int bit is cleared (Int = 0), the SPU will signal the LSU and the IFU on all ldxa to the MA registers.

An MA-LD operation is started with a stxa to MA_CTL register opcode equals MA-LD, and the length field specifies the number of words to be fetched from the L2-cache. The SPU sends a PCX request to the LSU and waits for an acknowledgement from the LSU before sending out another request. The L2-cache returns data to the SPU directly on CPX.

Any data returned with an uncorrectable error will halt the operation. If the Int bit is cleared (Int = 0), the SPU will send a signal to the LSU and the IFU on any ldxa to MA register.

Any data returned with a correctable error will cause the error address to be sent to IFU and be logged, while the operation will continue until completion.

Table 2.4 illustrates the error handling behavior.

Table 2.4: Error Handling Behavior			
NCEEN	Int	LSU	IFU
0	0	-	error_log
0	1	-	error_log
1	0	precise trap	error_log
1	1	-	error_log

The MA_MUL, MA_RED, and the MA_EXP operations all started with a stxa to MA_CTL register with an opcode equal to the respective operation, and the length field specifies the number of 64-bit words for each operation. The maximum length of these operations should never exceed 32 words.

The MA_MUL operates on A, B, M, N and N operands. The result will be stored in the X operand.

The MA_RED operates on A and N operands and the result will be stored in the R operand.

The MA_EXP performs the inner loop of modular exponentiation of A, M, N, X, E, operands stored in the MA Memory. This is the binary approach where the MA_MUL, followed by MA_RED functions, are called and will have the results stored in X operand.

The parity error encountered on an operand read will cause the operation to be halted. The LSU and the IFU will be signaled.

Figure 2.27 shows a pipeline diagram that illustrates the sequence of the result generation of the multiply function.

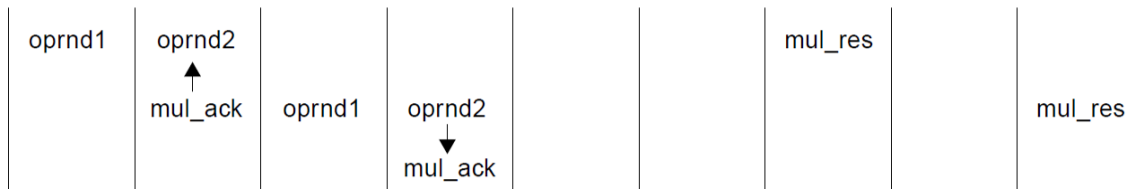


Figure 2.27: Multiply Function Result Generation Sequence Pipeline Diagram

2.13 Memory Management Unit

The memory management unit (MMU) maintains the contents of the instruction translation lookaside buffer (ITLB) and the data translation lookaside buffer (DTLB). The ITLB resides in instruction fetch unit (IFU), and the DTLB resides in load and store unit (LSU). Figure 2.28 shows the relationship among the MMU and the TLBs.

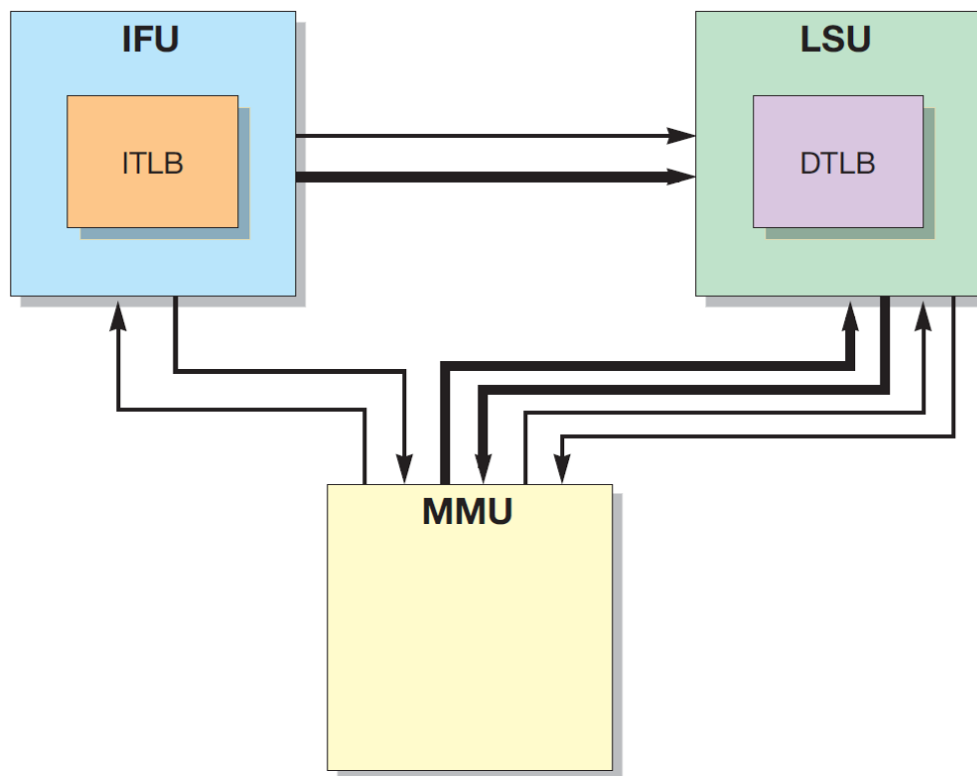


Figure 2.28: MMU and TLBs Relationship

2.13.1 The Role of MMU in Virtualization

The ~~OpenSPARC T1~~**PULSA** processor provides hardware support for the virtualization where multiple images and/or instances of the operating system (OS) coexist on top of the underlying chip multiple threading (CMT) microprocessor.

Figure 2.29 illustrates the view of virtualization.

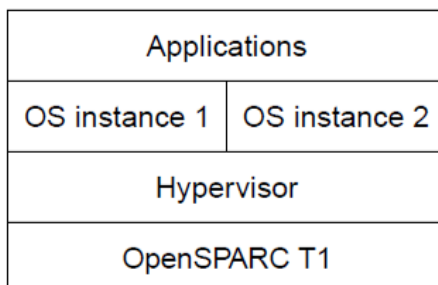


Figure 2.29: Virtualization Diagram

The hypervisor (HV) layer virtualizes the underlying central processing units (CPU). The multiple instances of the OS images form multiple partitions of the underlying

virtual machine. The hypervisor improves the OS portability to the new hardware and insures that failure in one domain would not affect the operation in the other domains. The ~~OpenSPARC T1~~**PULSA** processor supports up to eight partitions, and the hardware provides 3 bits of partition ID in order to distinguish one partition from another.

The hypervisor (HV) layer uses physical addresses (PA) while the supervisor (SV) layer views real addresses (RA) where the RAs represent a different abstraction of the underlying PAs. All applications use virtual addresses (VA) to access memory. (The VA will be translated to RA and then to PA by TLBs and the MMU.)

2.13.2 Data Flow in MMU

The MMU interacts with TLBs to maintain the content of TLBs. The system software manages the content of MMU by way of three kinds of operations - reads, writes, and demap. All TLB entries are shared among the threads, and the consistency among the TLB entries is maintained through auto-demap. The MMU is responsible for generating the pointers to the software translation storage buffers (TSB), and it also maintains the fault status for the various traps.

The access to the MMU is through the hypervisor-managed ASI operations such as `ldxa` and `stxa`. These ASI operations can be asynchronous or in-pipe, depending on the latency requirements. Those asynchronous ASI reads and writes will be queued up in LSU. Some of the ASI operations can be updated through faults or by a data access exception. Fault data for the status registers will be sent by trap logic unit (TLU) and the load and store unit (LSU).

2.13.3 Structure of Translation Lookaside Buffer

The translation lookaside buffer (TLB) consists of content addressable memory (CAM) and randomly addressable memory (RAM). CAM has one compare port and one read-write port (1C1RW), and RAM has one read-write port (1RW). The TLB supports the following mutually exclusive events.

1. CAM
2. Read
3. Write
4. Bypass
5. Demap
6. Soft-reset
7. Hard-reset

CAM consists of the following field of bits partition ID (PID), real (identifies a RA-to-PA translation or a VA-to-PA translation), context ID (CTXT), and virtual address (VA). The VA field is further broken down to page-size based fields with individual enables. The CTXT field also has its own enable in order to allow the flexibility in implementation. The CAM portion of the fields are for comparison purposes. RAM consists of the following field of bits, namely, physical address (PA) and attributes. The RAM portion of the fields are for read purposes, where a read could be caused by a software read or a CAM based 1-hot read.

Figure 2.30 illustrates the structure of the TLB.

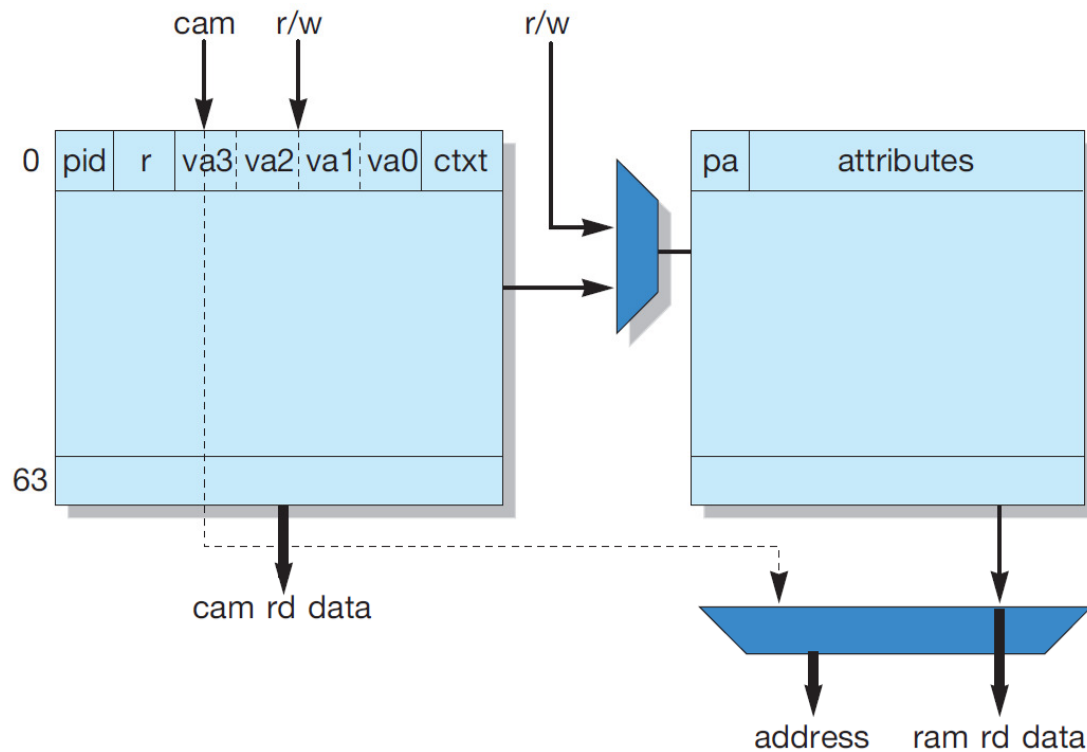


Figure 2.30: Translation Lookaside Buffer Structure

2.13.4 MMU ASI Operations

The types of regular MMU ASI operations are as follows:

- Writes
 - IMMU Data-In
 - DMMU Data-In
 - IMMU Data-Access
 - DMMU Data-Access
- Reads
 - IMMU Data-In
 - DMMU Data In
 - IMMU Tag-Read
 - DMMU Tag-Read
- Demap
 - IMMU Demap Page
 - DMMU Demap Page
 - IMMU Demap Context
 - DMMU Demap Context
 - IMMU Demap All (cannot demap locked pages)
 - DMMU Demap All (cannot demap locked pages)
- Soft-Reset
 - IMMU Invalidate All (including locked pages)
 - DMMU Invalidate All (including locked pages)
- Fault Related ASI Accesses to Registers
 - IMMU Synchronous Fault Status Register (SFSR)
 - DMMU Synchronous Fault Status Register (SFSR)
 - DMMU Synchronous Fault Address Register (SFAR)
 - IMMU Tag Access
 - DMMU Tag Access
 - IMMU Tag Target

- DMMU Tag Target
- ASI Accesses to Registers as Miss Handler Support
 - IMMU TSB Page Size 0
 - IMMU TSB Page Size 1
 - DMMU TSB Page Size 0
 - DMMU TSB Page Size 1
 - IMMU Context 0 TSB Page Size 0
 - IMMU Context 0 TSB Page Size 1
 - DMMU Context 0 TSB Page Size 0
 - DMMU Context 0 TSB Page Size 1
 - IMMU Context non-0 TSB Page Size 0
 - IMMU Context non-0 TSB Page Size 1
 - DMMU Context non-0 TSB Page Size 0
 - DMMU Context non-0 TSB Page Size 1
 - IMMU Context 0 Config
 - DMMU Context 0 Config
 - IMMU Context non-0 Config
 - DMMU Context non-0 Config

2.13.5 Specifics on TLB Write Access

A stxa to data-in or data-access causes a write operation that is asynchronous to the pipeline flow. Write requests are originated from the four-entry FIFO in the LSU. The LSU passes the write request to the MMU, which forwards it to the ITLB or the DTLB. A handshake from the target completes the write operation, which in turn enables the four-entry FIFO in the LSU to proceed with the next entry.

Write access to the data-in algorithmically places the translation table entry (TTE) in the TLB. Writes occur to the least significant unused entry. In contrast, write access to the data-access places the TTE in the specified entry in the TLB. For diagnostics purposes, a single bit parity error can be injected on writes.

A page may be specified as a real-on write, and a page will have a partition assigned to it on a write.

2.13.6 Specifics on TLB Read Access

TLB read operations follow the same handshake protocol as TLB write operations. The ASI data-access operations will read the RAM portion (that is, the TTE data). The ASI tag-read access operations will read the TTE tag from the RAM. The TLB read data will be returned to the bypass queue in the LSU. If no parity error is detected, the LSU will forward the data. Otherwise, the LSU will take a trap.

2.13.7 Translation Lookaside Buffer Demap

The system software can invalidate entries in the translation lookaside buffer (TLB) selectively using demap operations in any one of the following forms for the ITLB and the DTLB respectively and distinctly. Each demap operation is partition specific.

- Demap by page real Match VA-tag and translate RA to PA
- Demap by page virtual Match VA-tag and translate VA to PA
- Demap by context Match context only (has no effect on real pages)
- Demap all Demap all but the locked pages

The system software can clear the entire TLB distinctly through an invalidate all operation, which includes all of the locked pages.

2.13.8 TLB Auto-Demap Specifics

Each TLB is shared by all ~~four~~^{two} threads. The ~~OpenSPARC-T1~~^{PULSA} processor provides a hardware auto-demap to prevent the threads from writing to overlapping pages. Each auto-demap operation is partition specific. The sequence of an auto-demap operation is as follows.

1. Schedule a write from the four entry FIFO in the LSU.
2. Construct an equivalent auto-demap key.
3. Assert demap and complete with a handshake.
4. Assert write and complete with a handshake.

2.13.9 TLB Entry Replacement Algorithm

Each entry has a *Used* bit. An entry is picked to be a candidate for a replacement if it is the least significant unused bit among all 64 entries.

A used bit can be set on a write, or on a CAM hit, or when locked. A locked page will have its used bit always set. An invalid entry has its used bit always cleared. All used bits will be cleared when the TLB reaches a saturation point (that is, when all entries have their used bit set while a new entry needs to be put in a TLB). If a TLB remains saturated because all of the entries have been locked, the default replacement candidate (entry 0x63) will be chosen and an error condition will be reported.

2.13.10 TSB Pointer Construction

An MMU miss will cause the write of the faulting address and the context in the tag access. The tag access has a context 0 copy or a context non-0 copy, which is updated depending on the context of the fault. The miss handler will read the pointer of page-size 0 or page-size 1. The hardware will continue with the following sequence in order to complete the operation.

1. Read `zero_ctxt_cfg` or `nonzero_ctxt_cfg` to determine the page size.
2. Read `zero_ctxt_tsb_base_ps0` or `zero_ctxt_tsb_base_ps1` on `nonzero_ctxt_tsb_base_ps0` or `nonzero_ctxt_tsb_base_ps1` to get the TSB base address and size of the TSB.
3. Access tag.

Software will then generate a pointer into the TSB based on the VA, the TSB base address, the TSB size, and the Tag.

2.14 Trap Logic Unit

The trap logic unit (TLU) supports six trap levels. A trap can be in one of the following four modes: reset-error-debug (RED) mode, hypervisor (HV) mode, supervisor (SV) mode, and user mode. Traps will cause the SPARC core pipeline to be flushed, and a thread-switch to occur, until the trap vector (redirect PC) has been resolved.

Software interrupts are delivered to each of the virtual cores using the *interrupt_level_n* trap through the `SOFTINT_REG` register. I/O and CPU cross-call interrupts are delivered to each virtual core using the *interrupt_vector* trap. Up to 64 outstanding interrupts can be queued up per thread for each interrupt vector. Interrupt vectors are implicitly prioritized, with vector 0x63 being at the highest priority, while vector 0x0 is at the lowest priority. Each I/O interrupt source has a hardwired interrupt number that is used as the interrupt vector by the I/O bridge block.

The TLU is in a logically central position to collect all of the traps and interrupts and forward them. Figure 2.31 illustrates the TLU role with respect to all other backlogs in a SPARC core.

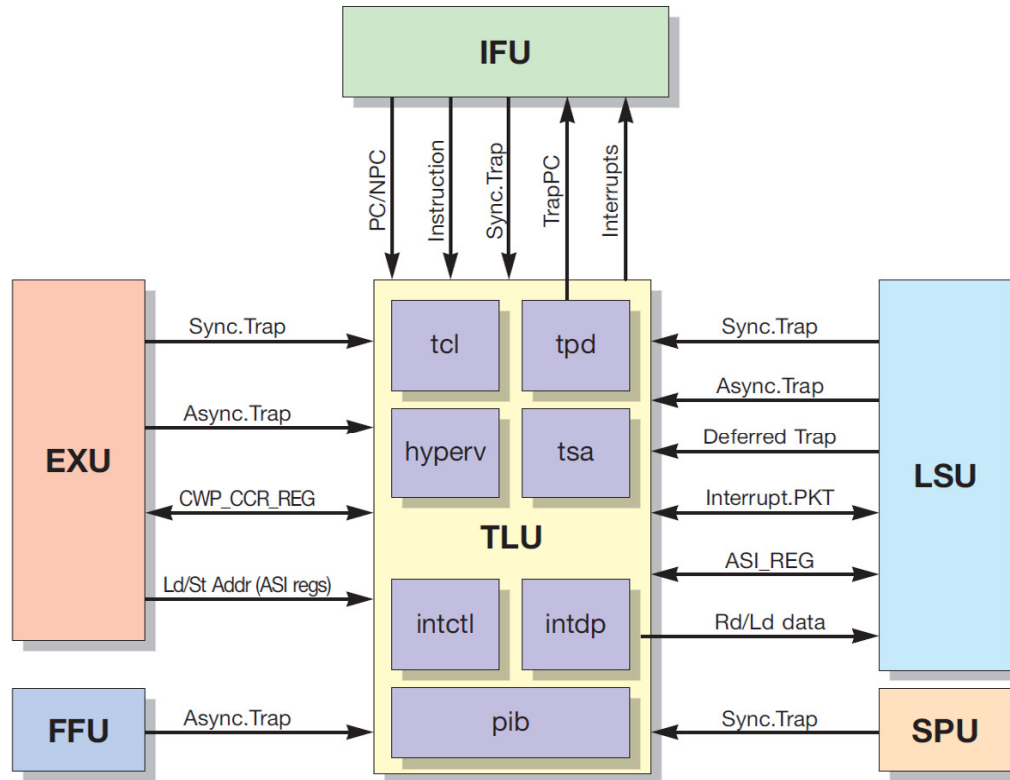


Figure 2.31: TLU Role With Respect to All Other Backlogs in a SPARC Core

The following list highlights the functionality of the TLU:

- Collects traps from all units in the SPARC core
- Detects some types of traps internal to the TLU
- Resolves the trap priority and generates the trap vector
- Sends flush-pipe to other SPARC units using a set of non-LSU traps.
- Maintains processors state registers
- Manages the trap stack
- Restores the processor state from the trap stack on done or retry instructions
- Implements an inter-thread interrupt delivery
- Receives and processes all types of interrupts
- Maintains tick, all tick-compares, and the SOFTINT related registers
- Generates timer interrupts and software interrupts (interrupt_level_n type)
- Maintains performance instrumentation counters (PIC)

2.14.1 Architecture Registers in the Trap Logic Unit

The following list highlights the architecture registers maintained by the trap logic unit (TLU). Only supervisor (SV) or hypervisor (HV) privileged code can access these registers.

1. Processor state and control registers
 - PSTATE (processor state) register
 - TL (trap level) register
 - GL (global register window level) register
 - PIL (processor interrupt level) register
 - TBA (trap base address) register
 - HPSTATE (Hypervisor processor state) register
 - HTBA (Hypervisor trap base address) register
 - HINTP (Hypervisor interrupt pending) register
 - HSTICK_CMPR_REG (Hypervisor system tick compare) register
2. Trap stack (six-deep)
 - TPC (trap PC) register
 - TNPC (trap next PC) register
 - TTYPE (trap type) register
 - TSTATE (trap state) register
 - HTSTATE (Hypervisor trap state) register
3. Ancillary state registers
 - TICK_REG (tick) register
 - STICK_REG (system tick) register
 - TICK_CMPR_REG (tick compare) register
 - STICK_CMPR_REG (system tick compare) register
 - SOFTINT_REG (software interrupt) register
 - SET_SOFTINT (set software interrupt register) register
 - CLEAR_SOFTINT (clear software interrupt register) register
 - PERF_CONTROL_REG (performance control) register
 - PERF_COUNTER (performance counter) register

4. ASI mapped registers

- Scratch-pad registers (eight of them)
- CPU and device mondo registers
- Head and tail pointers of resumable and non-resumable error queue
- CPU interrupt registers
 - Interrupt receive register
 - Incoming vector register
 - Interrupt dispatch registers (for cross-calls)

2.14.2 Trap Types

Traps can be generated from the user code, the supervisor code, or from the hypervisor code. A trap will be delivered to different trap handler levels for further processing, namely the supervisor level (SV level; otherwise known as the privileged level) or the hypervisor level (HV level). The way the traps are generated can help categorize a trap into either an asynchronous trap (asynchronous to the SPARC core pipeline operation) or a synchronous trap (synchronous to the SPARC core pipeline operation).

There are three defined categories of traps: precise trap, deferred trap, and disrupting trap. The following paragraphs briefly describe the nature of each category of trap.

1. Precise trap

A precise trap is induced by a particular instruction and occurs before any program-visible state has been changed by the trap-inducing instruction. When a precise trap occurs, several conditions must be true:

- The PC saved in TPC[TL] points to the instruction that induced the trap, and NPC saved in NTPC[TL] points to the instruction that was to be executed next.
- All instructions issued before the one that induced the trap must have completed their execution.
- Any instructions issued after the one that induced the trap remain unexecuted.

2. Deferred trap

A deferred trap is induced by a particular instruction. However, the trap may occur after the program-visible state has been changed by the execution of either the trap inducing instruction itself, or one or more other instructions.

If an instruction induces a deferred trap, and a precise trap occurs simultaneously, the deferred trap may not be deferred past the precise trap.

3. Disrupting trap

A disrupting trap is caused by a condition (for example, an interrupt), rather than directly caused by a particular instruction. When a disrupting trap has been serviced, the program execution resumes where it left off. A reset type of trap resumes execution at the unique reset address and it is not a disrupting trap.

Disrupting traps are controlled by a combination of the processor interrupt level (PIL) and the interrupt enable (IE) bit field of the processor state register (PSTATE). A disrupting trap condition is ignored when the interrupts are disabled ($PSTATE.IE = 0$) or the conditions interrupt level is lower than that specified in the PIL.

A disrupting trap may be due to either an interrupt request not directly related to a previously executed instruction, or to an exception related to a previously executed instruction. Interrupt requests may be either internal or external, and can be induced by the assertion of a signal not directly related to any particular processor or memory state.

A disrupting trap, related to an earlier instruction causing an exception, is similar to a deferred trap in that it occurs after instructions, follows the trap-inducing instruction, and modifies the processor or memory state. The difference is that the condition which caused the instruction to induce the trap may lead to unrecoverable errors, since the implantation may not preserve the necessary states.

Disrupting trap conditions should persist until the corresponding trap is taken.

Table 2.5 illustrates the type of traps supported by the ~~OpenSPARC T1~~**PULSA** processor.

Table 2.5: Supported ~~OpenSPARC T1~~**PULSA** Trap Types

Trap Type	Deferred	Disrupting	Precise
Asynchronous	None	None	Spill traps, FPU traps, DTLB parity error on loads, SPU-MA memory error return on load to SYNC reg
Synchronous	DTLB parity error on stores (precise to SW)	Interrupts and some error traps	All other traps

Asynchronous traps are taken opportunistically. They will be pending until the TLU can find a *trap bubble* in the SPARC core pipeline. A maximum of one asynchronous trap per thread can be pending at a time. When the other ~~three~~ threads ~~are~~**is** taking

traps back-to-back, an asynchronous trap may wait a maximum three SPARC core clock cycles before the trap is taken.

2.14.3 Trap Flow

An asynchronous trap is normally associated with long latency instructions and saves/restores, so the occurrence of such a trap is not synchronous with the SPARC core pipeline operation. These traps are all precise traps in the ~~OpenSPARC T1~~**PULSA** processor. A trap bubble is identified in the W-stage when there is no valid instruction available, or the instruction there is taking a trap. Asynchronous traps will be taken at the W-stage when a trap bubble has been identified.

Disrupting traps are associated with certain particular conditions. The TLU collects them and forward them to the IFU. The IFU sends them down the pipeline as interrupts instead of sending instructions down. A trap bubble is thus guaranteed at the W-stage, and the trap will be taken.

Figure 2.32 illustrates the trap flow sequence.

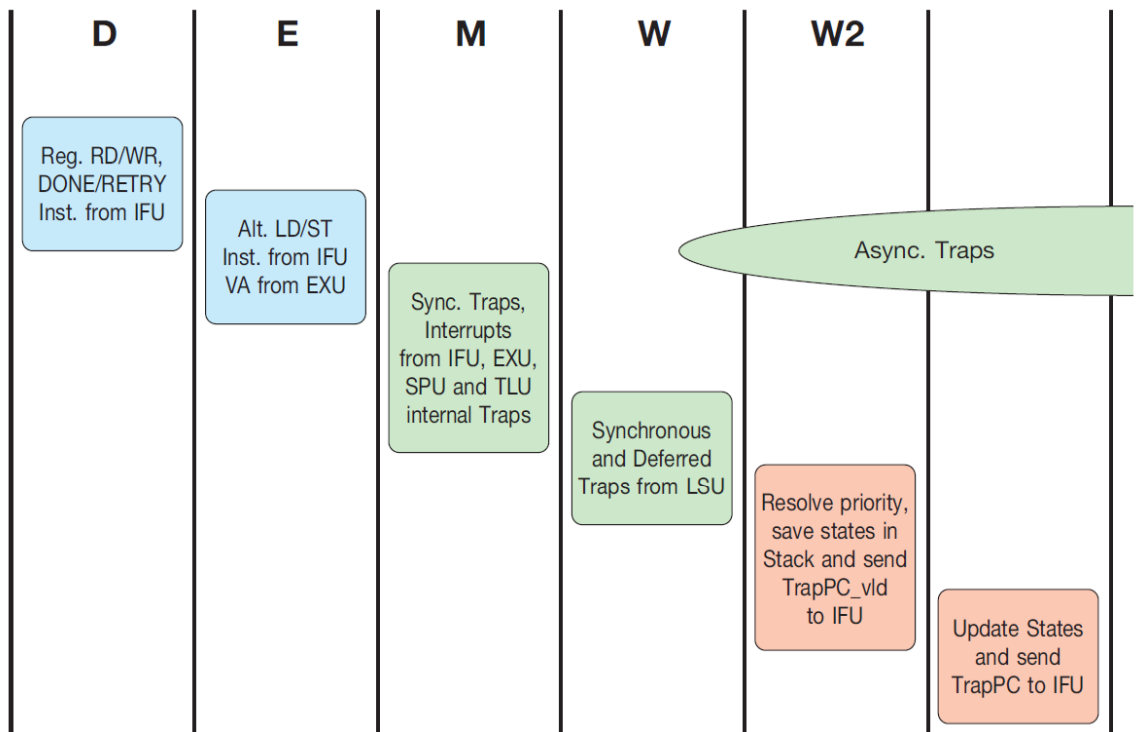


Figure 2.32: Trap Flow Sequence

All the traps from the IFU, EXU, SPU, LSU, and the TLU will be sorted through in order to resolve the priority first, and also to determine the following trap type (TTYTYPE) and trap vector (redirect PC). After these are resolved, the trap base address (TBA) will be selected to travel down the pipeline for further execution.

Figure 2.33 illustrates the trap flow with respect to the hardware blocks.

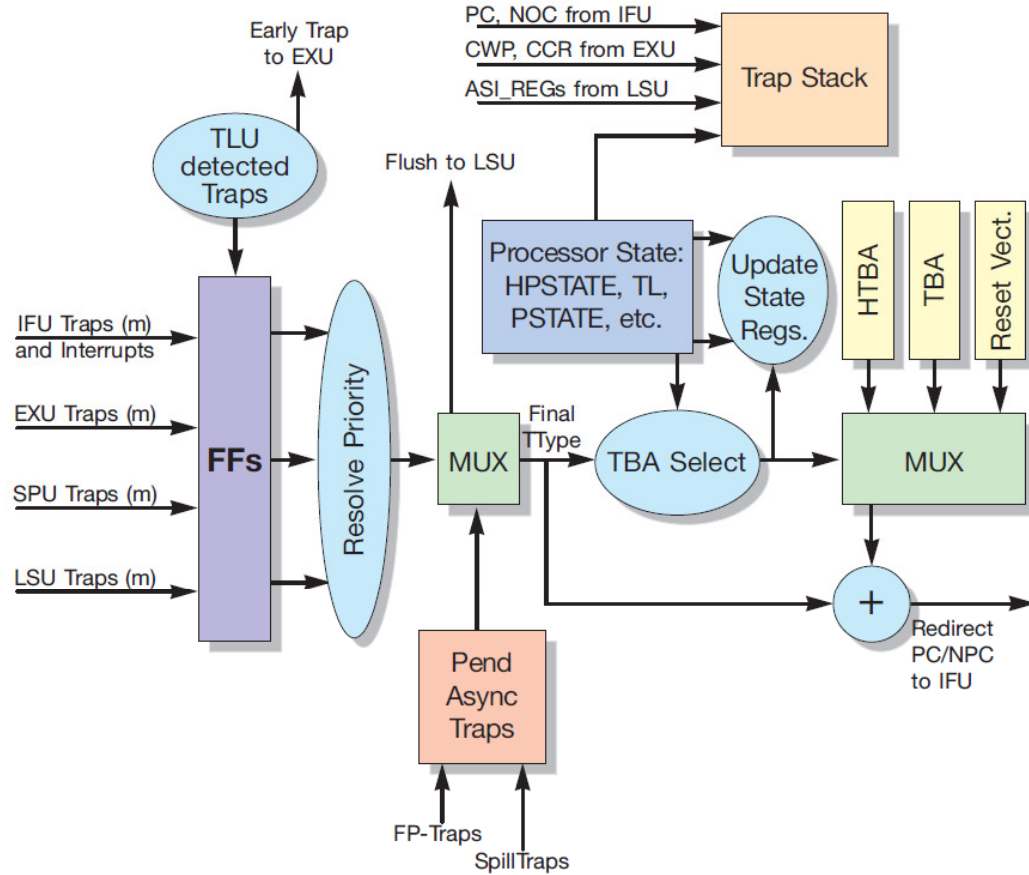


Figure 2.33: Trap Flow With Respect to the Hardware Blocks

2.14.4 Trap Program Counter Construction

The following list highlights the algorithm for constructing the trap program counter (TPC).

- Supervisor trap (SV trap)
Redirect PC \leq {TBA[47:15], (TL>0), TTYPE[8:0], 5b000000}
- Hypervisor trap (HV trap)
Redirect PC \leq {TBA[47:14], TTYPE[8:0], 5b000000}
- Traps in non-split mode
Redirect PC \leq {TBA[47:15], (TL>0), TTYPE[8:0], 5b000000}
- Reset trap
Redirect PC \leq {RSTVAddr[47:8], (TL>0), RST_TYPE[2:0], 5b000000}
– RSTVAddr = 0xFFFFFFFF00000000
- Done instruction
Redirect PC \leq TNPC[TL]
- Retry instruction
Redirect PC \leq TPC[TL]
Redirect NPC \leq TNPC[TL]

2.14.5 Interrupts

The software interrupts are delivered to each virtual core using the `interrupt_level_n` traps (0x41-0x4f) through the `SOFTINT_REG` register. I/O and CPU cross-call interrupts are delivered to each virtual core using the *interrupt_vector* trap (0x60).

Interrupt_vector traps for software interrupts have a corresponding 64-bit `ASLSWVR_INTR_RECEIVE` register.

I/O devices and CPU cross-call interrupts contain a 6-bit identifier, which determines which interrupt vector (level) in the `ASLSWVR_INTR_RECEIVE` register the interrupt will target. Each strand's `ASLSWVR_INTR_RECEIVE` register can queue up to 64 outstanding interrupts, one for each interrupt vector. Interrupt vectors are implicitly prioritized with vector 63 being the highest priority and vector 0 being the lowest priority.

Each I/O interrupt source has a hard-wired interrupt number, which is used to index a table of interrupt vector information (`INT_MAN`) in the I/O bridge unit. Generally, each I/O interrupt source will be assigned a unique virtual core target and vector level.

This association is defined by the software programming of the interrupt vector and the VC_ID fields in the INT_MAN table of the I/O bridge (IOB). The software must maintain the association between the interrupt vector and the hardware interrupt number in order to index the appropriate entry in the INT_MAN and the INT_CTL tables.

2.14.6 Interrupt Flow

Figure 2.34 illustrates the flow of hardware interrupts and vector interrupts.

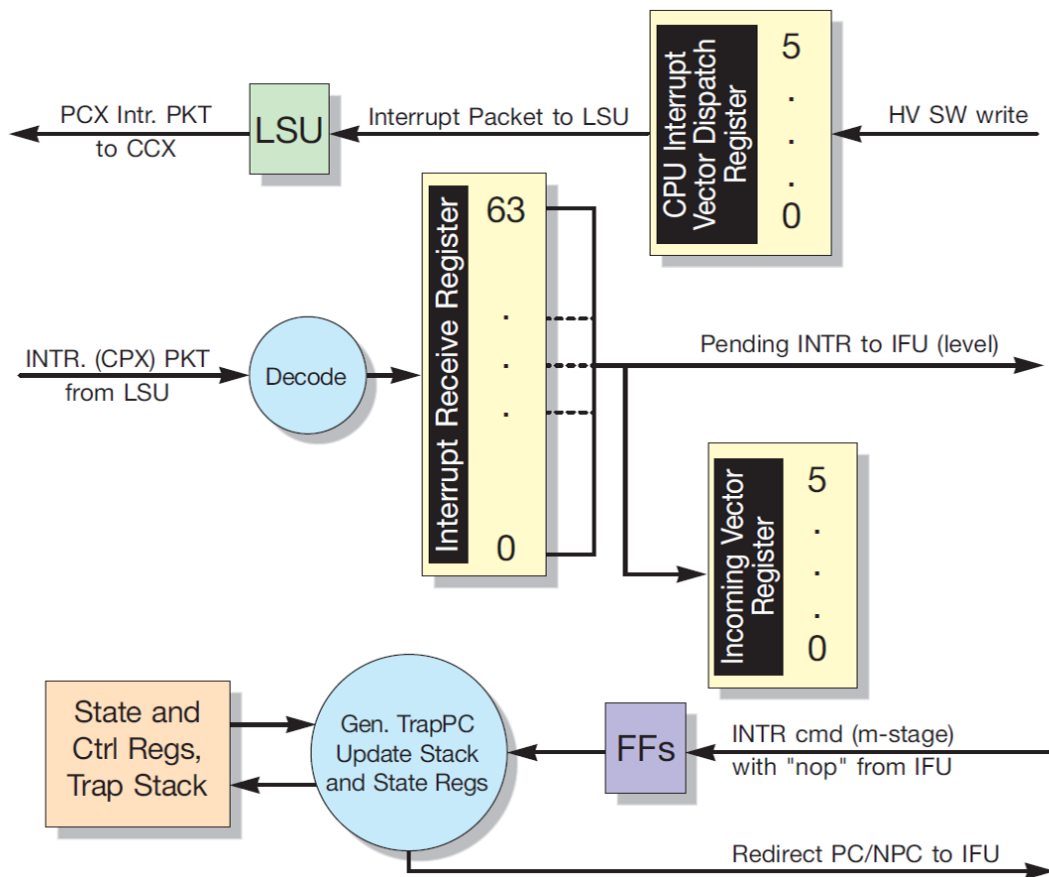


Figure 2.34: Flow of Hardware and Vector Interrupts

Figure 2.35 illustrates the flow of reset, idle, or resume interrupts.

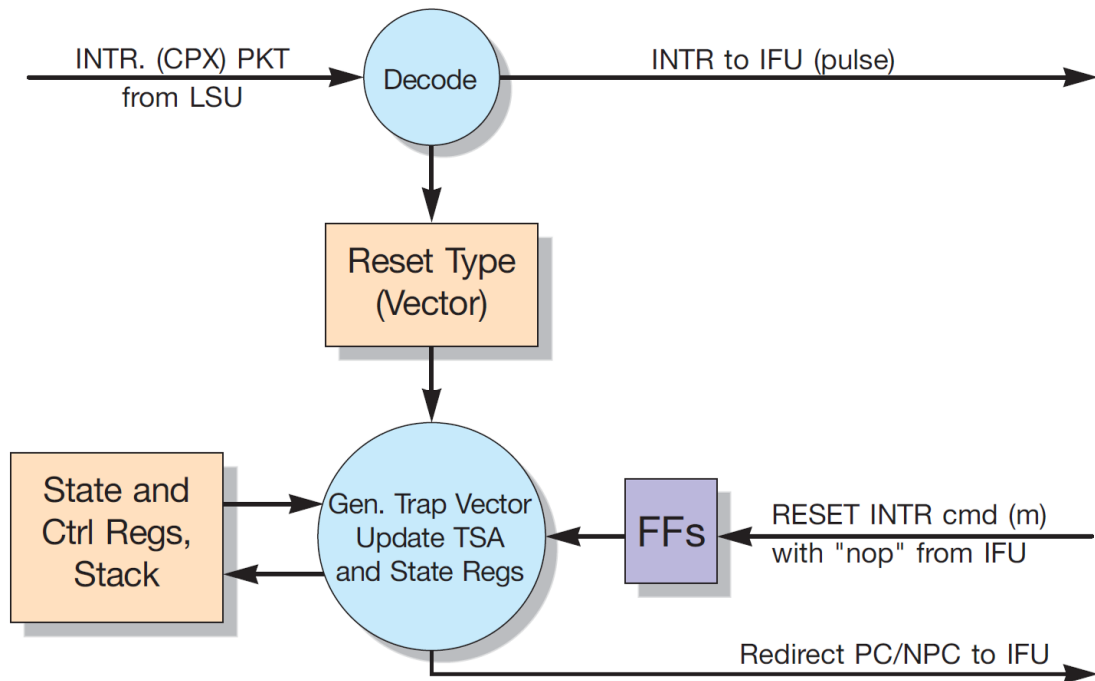


Figure 2.35: Flow of Reset, Idle, or Resume Interrupts

Figure 2.36 illustrates the flow of software and timer interrupts.

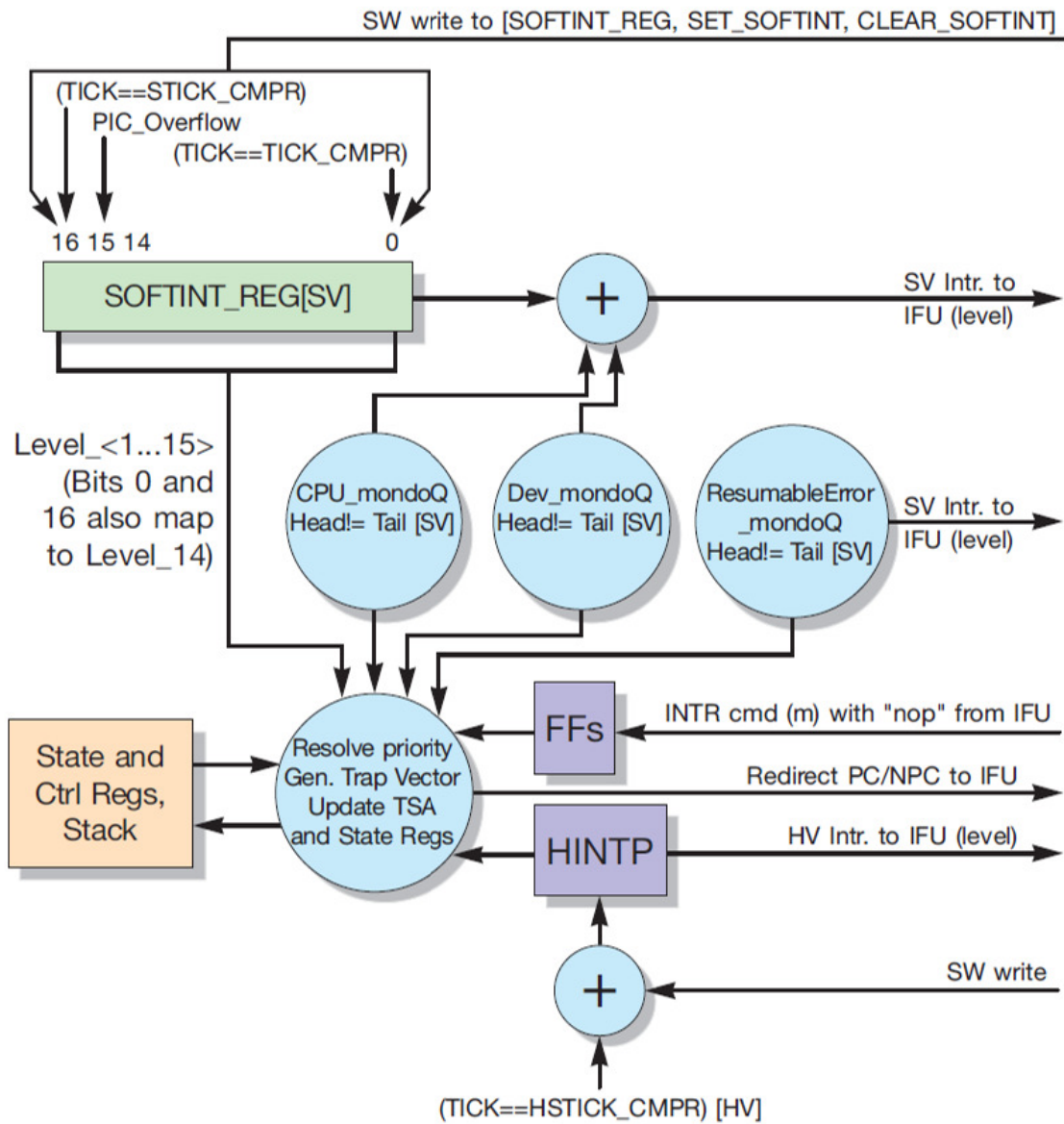


Figure 2.36: Flow of Software and Timer Interrupts

2.14.7 Interrupt Behavior and Interrupt Masking

The following list highlights the behavior and the masking of interrupts.

1. Hypervisor interrupts cannot be masked by the supervisor nor the user and can only be masked by the hypervisor by way of the PSTATE.IE bit. Such interrupts include hardware interrupts, HINTP, and so on.
2. Normal inter-core or inter-thread interrupts such as cross-calls can be sent by software writing to the CPU INT_VEC_DIS_REG register.
3. Special inter-core or inter-thread interrupts (such as reset, idle, or resume) can only be sent by software through the I/O bridge (IOB) by writing to the IOB INT_VEC_DIS_REG register.
4. Hypervisor will always suspend supervisor interrupts.
5. Some supervisor interrupts such as Mondo-Qs can only be masked by the PSTATE.IE bit.
6. Interrupts of *Interrupt_level_n*-type can only be masked by the PIL and the PSTATE.IE bit at the supervisor or user level.

2.14.8 Privilege Levels and States of a Thread

Split mode is referred to as the operating mode where hypervisor and supervisor modes are uniquely distinguished. Otherwise, the mode is referred to as non-split mode.

Table 2.6 illustrates the privilege levels and states of a thread.

Table 2.6: Privilege Levels and Thread States

	Red	Split Mode			Non-Split Mode	
		Hypervisor	Supervisor	User	Privileged	User
HPSTATE.enb	X	1	1	1	0	0
HPSTATE.red	1	0	0	0	0	0
HPSTATE.priv	1	1	0	0	X(1)	0
PSTATE.priv	1	X	1	0	1	0

2.14.9 Trap Modes Transition

Figure 2.37 illustrates the mode transitions among the different levels of traps.

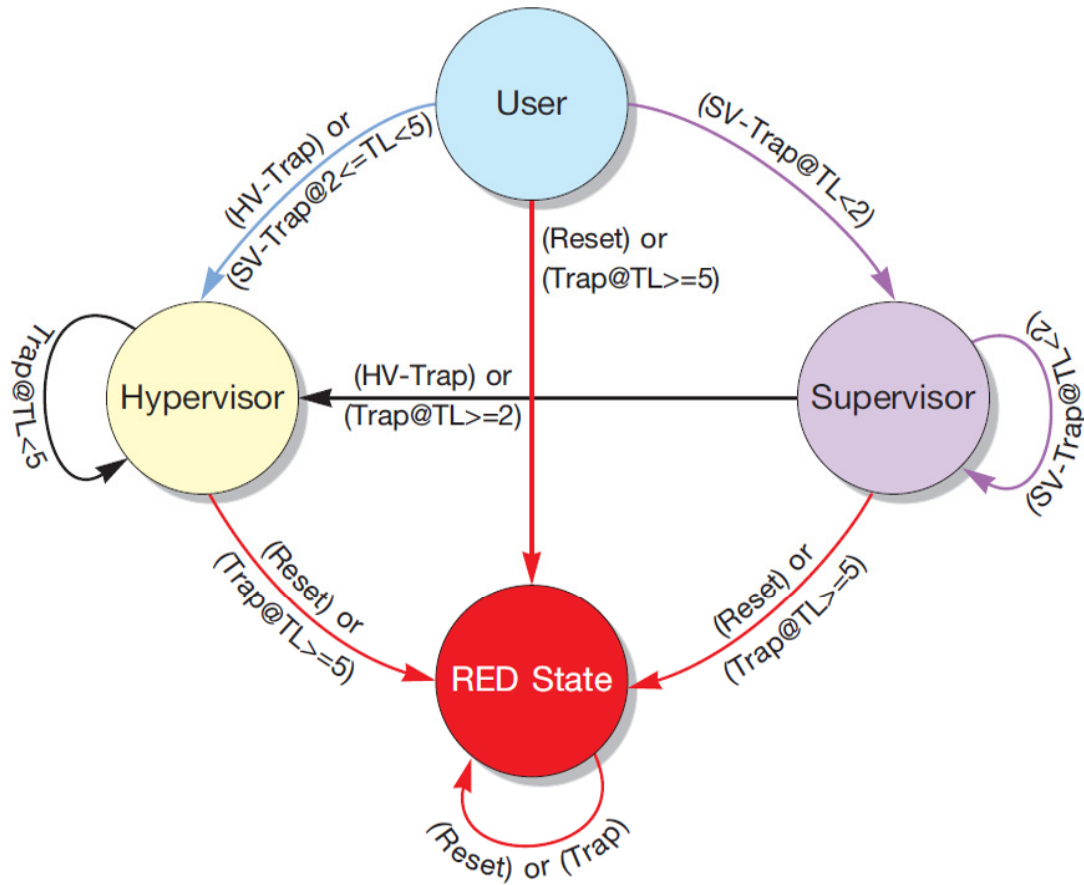


Figure 2.37: Trap Modes Transition

2.14.10 Thread States Transition

A thread can be in any one of these four states RED (reset, error, debug), supervisor (SV), hypervisor (HV), or user. The privilege level is very different in each different states. Figure 2.38 illustrates the state transition of a thread.

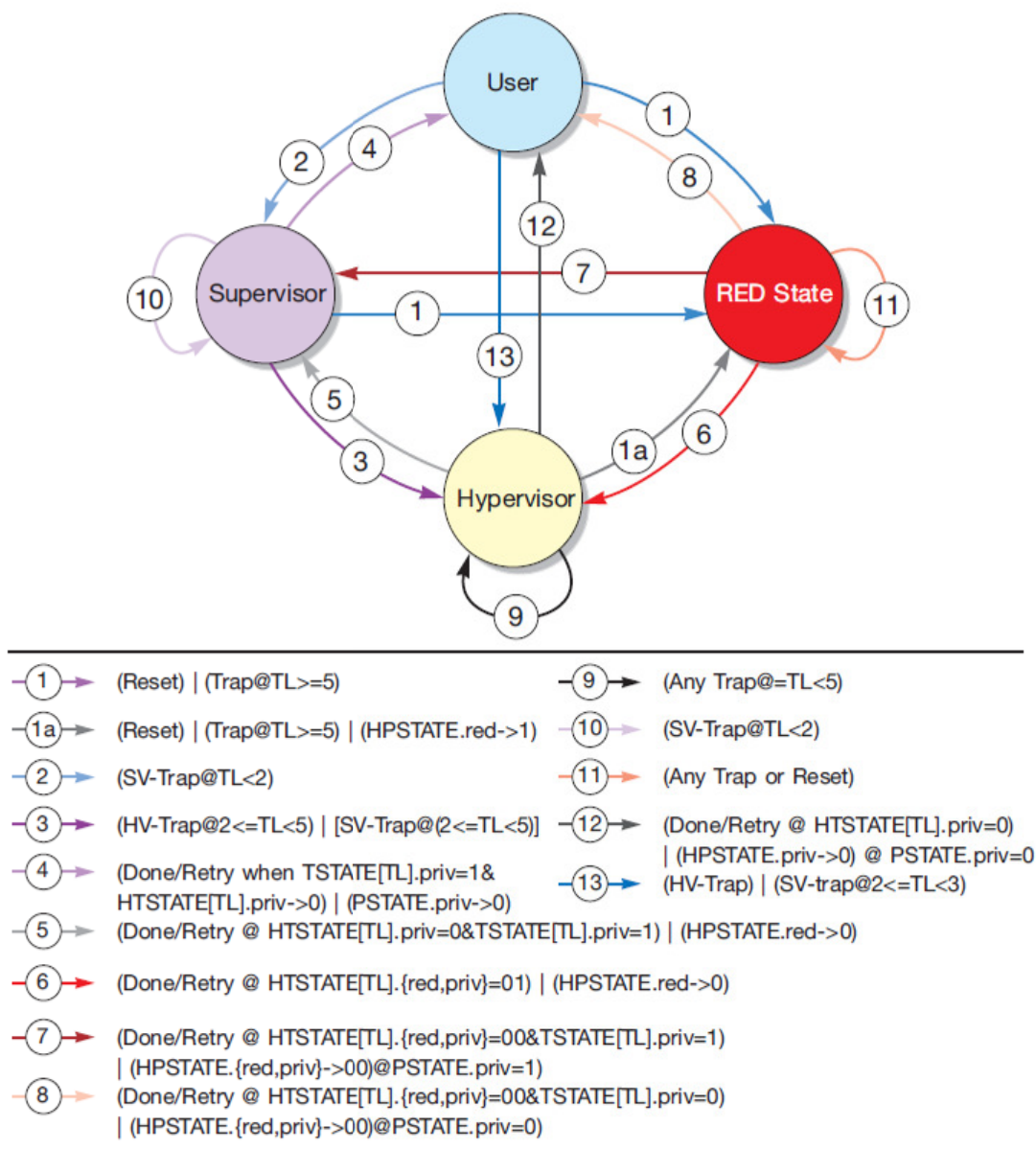


Figure 2.38: Thread State Transition

2.14.11 Content Construction for Processor State Registers

Processor state registers (PSRs) carry different content in different situations, such as, traps, interrupts, done instructions, or retry instructions. The following list highlights the register contents.

1. On traps or interrupts save states in the trap stack and update them

- (a) Update trap level (TL) and global level (GL)

- i. On normal traps or interrupts

$$TL = \min(TL+1, MAXTL)$$

$$GL = \min(GL+1, MAXGL) \text{ for hypervisor}$$

$$GL = \min(GL+1, 2) \text{ for supervisor}$$

- ii. On power-on reset (POR) or warm reset

$$TL = MAXTL (=6)$$

$$GL = MAXGL (=3)$$

- iii. On software write

For hypervisor:

$$TL \leq \min(wr\text{-}data[2:0], MAXTL) \text{ for hypervisor}$$

$$GL \leq \min(wr\text{-}data[3:0], MAXGL) \text{ for hypervisor}$$

For supervisor:

$$TL \leq \min(wr\text{-}data[2:0], 2) \text{ for supervisor}$$

$$GL \leq \min(wr\text{-}data[3:0], 2) \text{ for supervisor}$$

- (b) $PC \Rightarrow TPC[TL]$

- (c) $NPC \Rightarrow TNPC[TL]$

- (d) $\{ASL_REG, CCR_REG, GL, PSTATE\} \Rightarrow TSTATE[TL]$

- (e) $Final_Trap_Type \Rightarrow TTYPE[TL]$

- (f) $HPSTATE \Rightarrow HTSTATE[TL]$

- (g) Update $HPSTATE[enb, red, priv, \text{and so on}]$ register

- (h) Update $PSTATE[priv, ie, \text{and so on}]$ register

2. On done or retry instructions restore states from trap stack

(a) Update the trap level (TL) and the global level (GL)

TL <= TL -1

GL <= Restore from trap stack @[TL] and apply CAP

(b) Restore all the registers including PC, NPC, HPSTATE, PSTATE, from the trap stack @[TL]

(c) Send CWP and CCR register updates to the execution unit (EXU)

(d) Send ASI register update to load store unit (LSU)

(e) Send restored PC and NPC to the instruction fetch unit (IFU)

(f) Decrement TL

2.14.12 Trap Stack

The ~~OpenSPARC T1~~**PULSA** processor supports a six deep trap stack for six trap levels. The trap stack has one read port and one write port (1R1W), and it stores the following registers:

- PC
- NPC
- HPSTATE (Note: The HPSTATE.enb bit is not saved)
- PSTATE
- GL
- CWP
- CCR
- ASI_REG
- TTYPE

Synchronization based on the HTSTATE.priv bit and the TSTATE.priv bit for the non-split mode is not enforced on software writes, but synchronized while restoring done and retry instructions.

Software writes in supervisor mode to the TSTATE.gl bit do not cap at two. The cap is applied while restoring done and retry instructions.

2.14.13 Trap (Tcc) Instructions

Traps number 0x0 to 0x7f are all SPARC V9 compliant. They can be used by user software or by privileged software. The trap will be delivered to the supervisor if $TL < MAXPTL(2)$. Otherwise, it will be delivered to the hypervisor.

Traps number 0x80 to 0xff can only be used by privileged software. These traps are always delivered to hypervisor. User software using trap number 0x80 to 0xff will result in an *illegal instruction* trap if the *condition code* evaluates to true. Otherwise, it is just a NOP.

The instruction decoding and condition code evaluation of Tcc instructions are done by the instruction fetch unit (IFU) and the seventh bit of the Trap# is checked by the TLU.

2.14.14 Trap Level 0 Trap for Hypervisor

Whenever the trap level (TL) changes from non-zero to zero, and if the HPSTATE.tlz bit is set to 1, and the thread is not at Hypervisor privilege level, then a precise trap level 0 (TLZ) trap will be delivered to the hypervisor on the next following instruction.

The trap level can be changed by the done or the retry instructions or a WRPR instruction to TL. The trap is taken on the instruction immediately following these instructions. The change could be stepping down the trap level, or changing the TL from >0 to 0. The HPSTATE.tlz bit will not be cleared by the hardware when a trap is taken so the TLZ trap (tlz-trap) handler has to clear this bit before returning in order to avoid the infinite tlz-trap loop.

2.14.15 Performance Control Register and Performance Instrumentation Counter

Each thread has a privileged performance control register (PCR). Non-privileged accesses to this register causes a *privileged_opcode* trap.

Each thread has a performance instrumentation counter (PIC) register. The access privileged is controlled by the setting the PERF_CONTROL_REG.PRIV bit. When PERF_CONTROL_REG.PRIV=1, non-privileged accesses to this register cause a *privileged_action* trap.

Figure 2.39 highlights the layout of PCR and PIC.

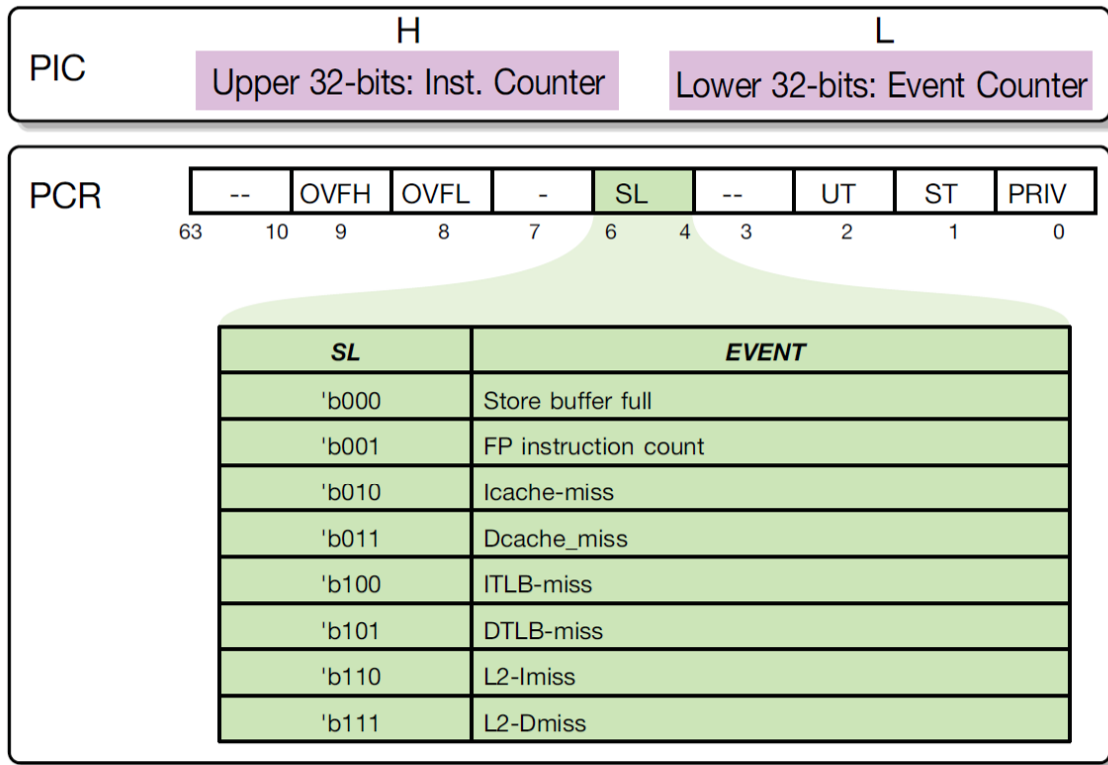


Figure 2.39: PCR and PIC Layout

If the PCR.OVFH bit is set to 1, the PIC.H has overflowed and the next event will cause a disrupting trap that appears to be precise to the instruction following the event.

If the PCR.OVFL bit is set to 1, the PIC.L has overflowed and next event will cause a disrupting trap that appears to be precise to the instruction following the event.

If the PCR.UT bit is set to 1, it counts events in user mode. Otherwise, it will ignore user mode events.

If the PCR.ST bit is set to 1 and HPSTATE.ENB is also set to 1, it counts events in supervisor mode. Otherwise, it will ignore supervisor mode events.

If the PCR.ST bit is set to 1 and HPSTATE.ENB is also set to 0, it counts events in hypervisor mode. Otherwise, it will ignore hypervisor mode events.

If the PCR.PRIV bit is set to 1, it prevents user code access to the PIC counter. Otherwise, it allows the user code to access the PIC counter.

The PIC.H bits form the instruction counter. Trapped or canceled instructions will not be counted. The Tcc instructions will be counted even if some other trap is taken on them.

The PIC.L bits form the event counter. The TLU includes only the counter control logic, while the other functional units in the SPARC core provide the logic to sig-

nal any event. An event counter overflow will generate a disrupting trap, while a performance counter overflow will generate a disrupting but precise trap (of a type level_15 interrupt) on the next following instruction and set the PCR.OVFH or the PCR.OVFL bits and bit-15 of the SOFTINT_REG register.

Software writes to the PCR that set one of the overflow bits (OVFH, OVFL) will also cause a disrupting but precise trap on the instruction following the next incrementing event.

2.15 Core Debug Features

2.15.1 Resetting a Thread

A thread may be reset by sending an interrupt packet to the core with the interrupt type set to reset (See Table??). The format of the interrupt data field will be as follows:

Table 2.7: Interrupt Data Field - Reset					
17:16	15:13	12:10	9:8	7:6	5:0
Interrupt Type 01=re-set	000	CPU_ID	TID	00	Trap Type 000001=POR 000011=XIR

The thread can be resumed by sending a second interrupt packet with the interrupt type set to resume.

2.15.2 Interrupting a Thread

A thread may be interrupted by sending an interrupt packet to the core with the interrupt type set to hardware interrupt (See Table??). The format of the interrupt data field will be as follows:

Table 2.8: Interrupt Data Field - HW Int					
17:16	15:13	12:10	9:8	7:6	5:0
Interrupt Type 00=hw int	000	CPU_ID	TID	00	Interrupt Vector

2.15.3 Shadow Scan

The core contains a shadow scan chain which allows several important architectural state registers to be captured and scanned out. In a full chip, this chain will normally be accessed through a special JTAG instruction. In a single core, the chain may be accessed by following the proper protocol to capture and scan out the data.

The capture and scan process does not affect the state of the core, and may be performed while the core is running.

Note In the original OpenSPARC T1 release, the shadow scan chain was unconnected in the RTL. It was assumed that the chain would be connected using DFT tools at the same time the regular scan chains were connected. In Release 1.6, the shadow scan chain is connected in RTL if the design is compiled with the variable `CONNECT_SHADOW_SCAN` defined. This makes it easier for FPGA implementations, where no regular scan chains are used.

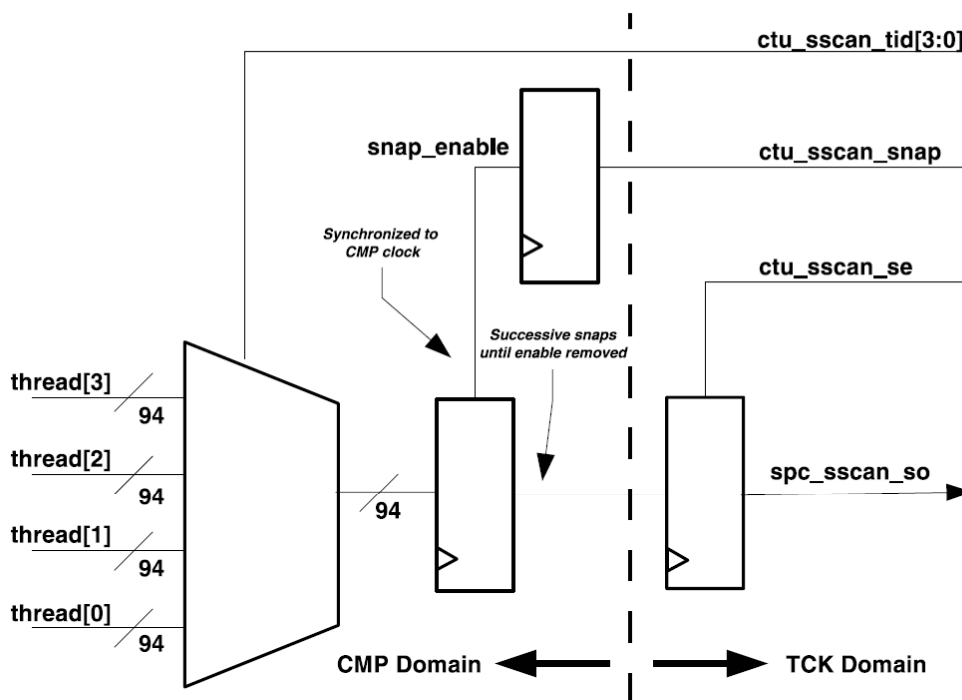


Figure 2.40: SPARC Shadow Scan Chain

The shadow scan snap block is located within the instruction fetch unit. Figure 2.40 shows the implementation of this block.

The thread to be examined is selected by signal `ctu_sscan_tid[3:0]`. The `ctu_sscan_snap` signal, after being synchronized to the CMP clock domain enables the sampling of

the shadow scan data for the selected thread. The captured data is then synchronized to the TCK clock domain, and scanned out using the TCK clock.

The shadow scan signals are usually connected to a JTAG TAP Controller. The thread ID, `ctu_sscan_tid[3:0]`, is decoded from the JTAG instruction. It is valid for as long as the instruction is held. The `ctu_sscan_snap` and `ctu_sscan_en` signals are decoded from the Capture-DR and Shift-DR states of the TAP controller. The timing relationship is shown in Figure 2.41.

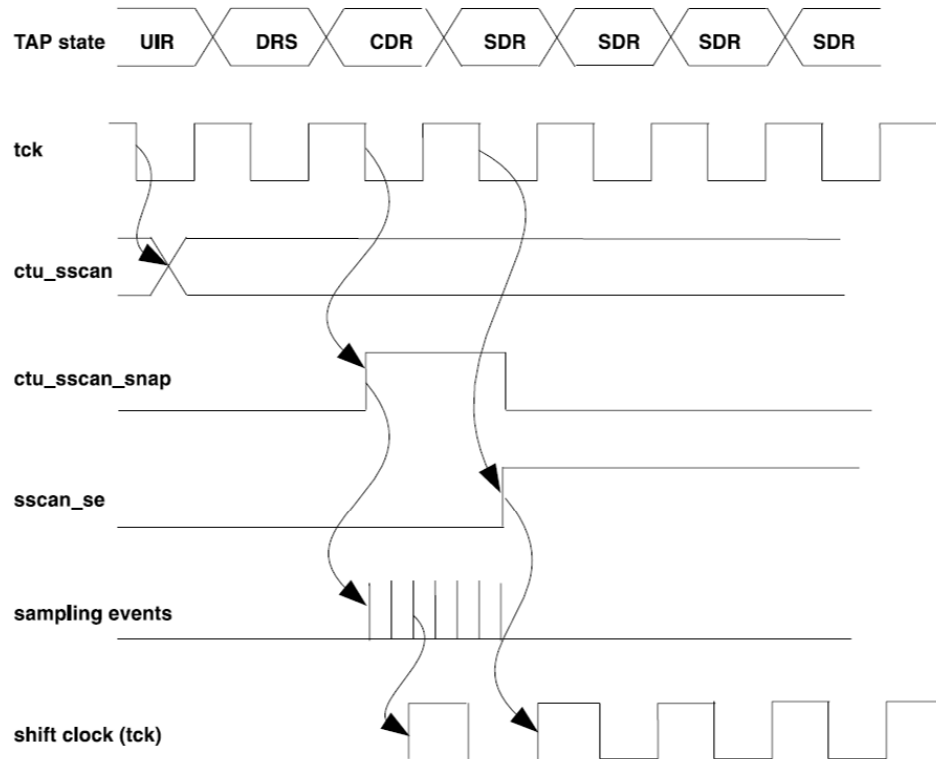


Figure 2.41: Shadow Scan Snap Timing Diagram

The core shadow scan chain is 94 bits long. It captures the following information:

Table 2.9: SPARC Physical Core Shadow Scan Chain

Bit Position	Register	Definition
93:91	fdm_busy[2:0]	Impl dep Mul/Div/FPU busy mask (fdm_busy[0]=bit 93)
90:88	Str_wait[2:0]	Impl dep Strand wait mask (Imiss/Other/STB)
87:83	Str_state[4:0]	Impl dep Strand State: 0x00=Idle 0x01=Waiting for long latency event 0x02=Halted (by instruction) 0x05=Executing Instruction 0x07=Speculatively executing 0x13=Speculatively ready to execute 0x19=Ready to execute
82:79	Id_pcx_rq_vid[3:0]	Impl dep Load busy mask
78:75	st_pcx_rq_vld[3:0]	Impl dep Store buffer busy mask
74	imiss_rcx_rq_vld	Impl dep PCX crosssbar arbitration mask - Imiss request
73	strm_pcx_rq_vld	Impl dep PCX crosssbar arbitration mask - Store request
72	fwdpkt_rq_vld	Impl dep PCX crosssbar arbitration mask - Forwarded packet
71	intrpt_pcx_rq_vld	Impl dep PCX crosssbar arbitration mask - Interrupt
70	fpop_pcx_rq_vld	Impl dep PCX crosssbar arbitration mask - FP op
69	lsu_stb_empty	Impl dep - Store Buffer Empty
68	tlb_ld_st	Impl dep - Asynchronous ASI Access Busy
67	dfq_byp_full	Impl dep - DFQ Read Hang
66:21	PC[47:2]	Architectural Program Counter
20	PSTATE[IE]	Architectural PSTATE - Interrupt Enable
19	PSTATE[PRIV]	Architectural PSTATE - Privileged
18	HPSTATE[TLZ]	Architectural HPSTATE - Trap Level Zero
17	HPSTATE[PRIV]	Architectural HPSTATE - Hyper-privileged
16	HPSTATE[RED]	Architectural HPSTATE - Red mode
15:7	TT[8:0]	Architectural Trap Type register
6:4	TL[2:0]	Architectural Trap Level register
3:0	mil_state[3:0]	Impl dep - I-cache Miss State Machine State

There is a single shadow scan chain of 94 bits per physical core. When a shadow scan sample is triggered, the shadow scan block muxes 94x4 bits down to 94 bits to be shifted out. The shadow scan chains for each physical core are placed on separate chains. Bit 0 of the chain is the first bit shifted out, but each field is arranged in the

shadow scan chain such that the MSB is shifted out first. For example, bit 0 of the shadow scan chain is `mil_state[3]`.

Chapter 3

SPARC Uncore

The uncore part of a tile includes the transceiver, the L1.5 cache, and possibly the FPU, the Clumpy Share Memory mapping module, as well as the Memory Limiter module.

The architecture of the Sparc Uncore is drawn in Figure 3.1. The transceivers for the NoC are not shown because implementation is still not finalized, ie. whether to buffer the input/output. PCX buffers, PCX decoder, CPX, and the CPX arbitrator as a whole can be considered to be the CCX Transceiver blocks. They represent the logics needed to seamlessly interface with the unmodified Sparc core.

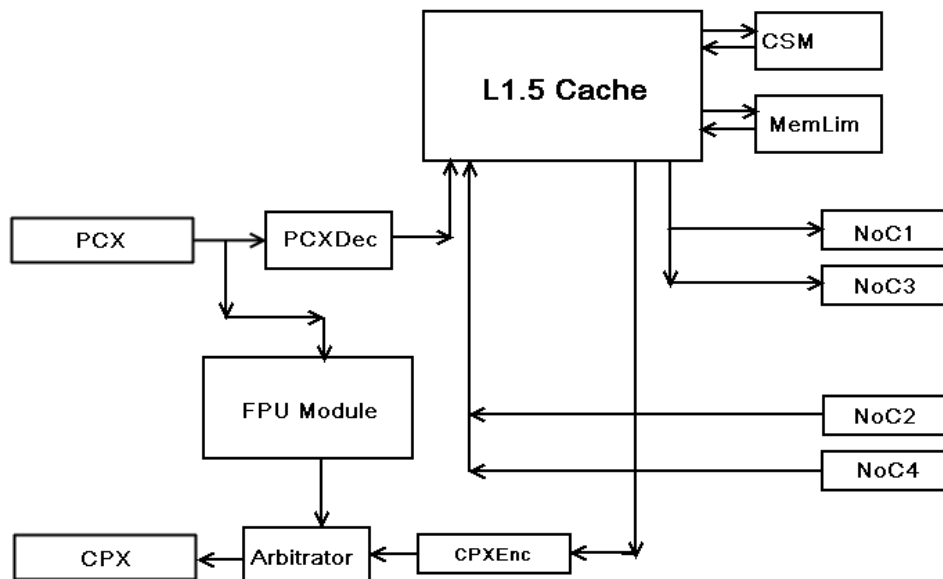


Figure 3.1: Architecture of the Uncore

The following sections will describe the major components of the uncore.

3.1 CCX Transceiver

The core sends and receives information encoded in PCX and CPX format as described in tables 3-1,3-2,3-3, and 3-4 in the OpenSPARC T1 Micro Specification documentation. While the FPU module is likely to receive and transmit in the same packet format, the L1.5 will defer the packet decoding to an external module, the PCXDec. This design decision is made solely to decouple any changes to the CCX packet format to the design of the L1.5.

The transceiver has a few needed buffers built-in, specifically the 2-deep buffer for PCX packets. The PCX is required by the T1 core to have 2-entry buffer for certain atomic requests to work. On the CPX path we are not required to have any buffer because the core has an efficient datapath to drain incoming packet in every cycle. The CPX arbitrator and the CPX encoder are likely to be simple combinational logics.

The logic of the arbitrator is not specified at the moment, but is likely to give priority to the CPX Encoder (memory) at all time. Forward progress should be warranted because a thread is stalled upon an FP operation. However, the thread might be starved if the other thread is not stalled.

3.2 L1.5 Cache

Note: * means to look for references in the official T1 text.

The L1.5 is an ad-hoc solution to add write-back capability for the T1's L1D, which is a write-through design. This section describes the high-level design of the L1.5. First, we will review the design of the L1D. Then we will look at the goals that the L1.5 will try to achieve, its current implementation, some input/output listings with neighbor blocks, and finally a detail description of the pipeline.

3.2.1 L1 Design Summary

The original cache is write through. On a data read miss the processor write to some miss status handling register* which contains the address and the register needed, and wait for the result. The core also predetermines the index and way that it will put the new data in, unless the load is specifically marked as uncacheable (IO or atomic loads). Subsequent load miss to the same line (or sub-cacheline) will hit and be chained to the MSHR, effectively making multiple loads to the same line impossible to encounter

in the L1.5. Load misses that hit the store-buffer will also not be propagated to the L1.5.

Instruction load misses are the same as data load miss, except that the miss handler also has the instruction cache slot*. To the L1.5, ifill and load misses are two different PCX packets.

On a write miss the processor write the result to a write buffer, which is 4-entry deep*, and continue. Any subsequent load to these addresses will be looked up and loaded from the write buffer. The write is then propagated to the L2 through the PCX one at a time, except for block store and block init instructions where store instructions are sent without waiting for acknowledgement. At this point the write is still not yet updated in the L1; yet the entry is still valid. Only after the write got the L2 and is returning to the L1 through a CPX packet, the L1 updates its data cache (if the entry is still not replaced by other loads) and the entry in the write buffer popped.

Of special note is that the L1/LSU does not have invalidation logic. What this means is that we can't just send an address to the LSU and tell it to invalidate the cache line. Instead what we have to do is to send both the cache index and way index, necessitating the need to keep a duplicated tag outside of the LSU; the L2 keeps these tags in the T1 design.

3.2.2 L1.5 Design Summary

Provides write-back capability for L1D

Does not serve ifill requests from the L1I

L1.5 does not have the capacity to cache both data and instructions. All requests from the icache will be bypassed to the L2.

Does not cross-invalidate the L1I on an L2 data load response

L1.5 does not duplicate tag for the icache, therefore the L2 will need to keep track of the icache and dcache.

Supports all-way index invalidation of the icache

The L2 can send command to invalidate all 4 ways of an index in the icache in one cycle.

Does not cross-invalidate the L1D on an L2 ifill response

Self-modifying code is relatively rare so it is not worth it to check the cache tag on every ifill.

Adheres to the L1-L2 cache coherence protocol

Does not self-invalidate on an I/O non-cacheable ld/st

I/O data should be be cached in the first place.

Self-invalidates on a returned atomic operation

Atomics are also uncacheable. In this case invalidations are efficient because data is warranted to not cause a write-back.

Atomic ops on Modified line are done in L1.5 without getting back to L2

Alternatives are (1) write back first then send the atomic ops to L2, or (2) do the atomics in L1.5, then write back to L2.

3.2.3 L1.5 Design Elaboration

The L1.5 is designed as a separate, independent cache level below the L1D. There were thoughts of making it act as a duplicated L1D cache to provide another read port, but those designs were either low performance or requiring modification to the LSU in the SPARC core.

In the L1.5 we have the data array the same size as the L1D (8KB), an address tag array (1 ported), a MESI tag array (using flops), and a way-map cache to keep the L1D synchronized with the L1.5 (for invalidation purposes).

The L1.5 will have a pipeline design with 2 stages. However, there are also a source stage and a sink stage that adds to the total latency of an operation.

3.2.3.1 Optimizations elaborations

Some of the bullet points above are adhoc optimizations. For example, the reason that the L1.5 doesn't cross-invalidate the L1I is simply because we don't have the L1I tag duplicated. We don't cross invalidate the L1D because it is uncommon and will incur at least 1 cycle. We will let the L2 send the invalidations instead.

There are also some optimizations with regard to the coherence protocol because it is not well defined right now. For example, how to deal with NC and Atomic operations in our write-back caching scheme.

3.2.4 Interactions With Blocks in Uncore

PCX decoder

the l15 receives memory requests, decoded from the PCX buffer.

Clumpy shared memory module

l15 will ask CSM for directory addresses when sending out memory read/write requests. TODO: which pipeline stage

Bandwidth limiter module

for sending data to the network. There will probably

3.2.4.1 I/O listing

From PCX decoder Mostly just fields in the PCX message table. TODO: right now does not handle any interrupt.

- valid/ack
- request type: load, ifill, st, cas1, cas2, swp/ldstb
- non-cachable: load non allocate for I/O. Always 1 for CAS.
- size: valid for I/O
- invalidate: tells the L1.5 to invalidate all way of an index with b123 (inval all). Used when CRC check failed in a cache way in L1
- threadid
- prefetch: only valid for load. NC should also be asserted.
- block store, block init store: only valid for stores
- L1 way replacement information (used to update the way-map table)

To CPX encoder Mostly just fields in the PCX message table (closely matches CPX packet format in page 3-7)

- valid/ack
- types, L2 miss (TODO is this needed?), threadid
- NC (tells the L1 to not cache this data)
- Prefetch: Only valid with load requests, tells the CPU to not fill data to registers
- Four-byte fill: ifill from PROM and IO
- Atomic: valid in load and store acks. TODO necessary? What does it do in the RTL?
- Data
- wayvalid & waytoinvalid: invalid icache if enabled on a load, and invalid dcache if enabled on an ifill
- invalidation fields:
 - icache inval all way
 - dcache inval all way
 - address [11:6]
 - address [5:4]
 - dcache_inval_way

- dcache_inval
- icache_inval_way
- icache_inval

From NoC input buffers The messages from NoCs will be fully buffered first before they are processed. While we can reduce the number of flops by not fully buffering, first, it is easier to write the state machines when data are fully buffered, and second, writing to SRAM (dcache fill) and to CPX encoder/buffer (icache fill) requires the whole cache line anyway. Since an ifill requires 32B (4 packets), we need a 40B buffer for NoC4. In the current specification the L1.5 will read in NoC2&4. NoC2 are FwdReqs with no data, and NoC4 are Read/Write responses from the L2 (with data).

This listing could also serve as a guideline for the specification for coherence messages.

- valid/ack
- return types: FwdERd, FwdEWr, FwdMRd, FwdMWr, Inv, AckDt, AckEDt, AckWb, AckAtomicDt, AckAtomicWb, AckDtIfill
- Ld/St/Atomic MSHR ID
- Non-Cacheable Bit
- L2 miss
- Four-byte fill: ifill from PROM and IO (carried from T1)
- Data field (up to 32B or 4 packet long)
- Address field (for invalidation, 40b long)
- Invalidate icache index# (for keeping icache and dcache exclusive on a dcache load)

The following fields are probably not needed

- - Prefetch: Only valid with load requests, prevents the CPU from loading data to registers Prefetch bit is not needed because it will be kept in the MSHR
- - Atomic: valid in load and store acks. necessary? What does it do in the RTL? Atomic bit is likely not needed
- - Data length: 0 (coherence) or 2 (data) or 4 (instruction) 64b Data length is implicitly specified by the return type
- - ThreadID: implicit in MSHR

To NoC output buffers It is not yet decided whether we will have buffers at the output or not, as we can have the data streamed out directly from the SRAM array.

However, that means not only the L1.5 will not be able to write to NoC1&3 concurrently, but also effectively stall the pipeline by blocking the data array structure.

Another concern with having/not having buffers is deadlock avoidance, specifically having a lower priority network blocking a higher one in the pipeline. Since we don't actually know whether the message in the pipeline is warranted to be fully flushed to the network, we need at least one buffer between the pipeline and the network serializer.

- valid/ack
- types, L2 miss, threadid
- NC (copied from L1 requests)
- Types: 16B load, 32B load (ifill), 16B writeback, 16B write req, atomic, 64B FwdDtAck
- Prefetch: Only valid with load requests, prevents the CPU from loading data to registers
- Four-byte fill: ifill from PROM and IO
- Address: 40b for Reqs (TODO is this also valid for FwdAck??? Depends on L2 implementation)
- MSHR id: for FwdAck???
- Data: up to 64B (only for FwdAcks, NoC3)

3.2.5 Pipeline Design

3.2.5.1 Tenets of designing pipeline

The design of the pipeline revolves around these few tenets for avoiding message-dependant deadlocks:

1. There can be no cyclic dependency, although it is okay to have cycles composed only of weak dependencies.
2. Definition: strong dependencies are protocol dependencies.
3. Definition: weak dependencies are non-protocol (implementation) dependencies.
4. If by receiving message A, message B had to be produced before message A unblocks or deallocates some resources, then A is strongly dependent on B, ie. $A \Rightarrow B$.
5. Strong dependencies are not transitive, but a message A can be strongly dependent on multiple other message types.

6. If message A can be sinked indefinitely AND does not use resources, then A is a terminal message and doesn't strongly depend on anything else.
7. Message A is also a terminal message if no other messages are generated after A is processed.
8. If message A can be blocked and backed off to the network, then every other message type in A's channel is weakly dependent on A, eg. $B \rightarrow A$, $C \rightarrow A$, $A \rightarrow A$.
9. We can break the above dependency by having enough buffer space for incoming message A.
10. If generating message A blocks other messages in the pipeline from processing, then those messages are also weakly dependent on A, eg. $A \rightarrow A$, $B \rightarrow A$, $C \rightarrow A$.
11. The above dependency can be broken by having enough output buffer for A, and/or through careful admittance of A to the pipeline, or having separate pipelines for A, B, and C.

A simple example Suppose we have four messages: A, B, C and D. A is strongly dependent on B, and C on D, as shown in Figure 3.2.5.1.

Now, if we allocate two channels (NoC) for these four messages, A and C on one, and B and D on the other, as shown in Figure 3.2.5.1. Intuitively, deadlock shouldn't be possible, and we see no cycles in the graph except for the weak dependencies. Now, on the other hand, if we allocate the channels such as in Figure 3.2.5.1, where we put A and D, and C and B together, a cyclic dependency is possible through $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$.

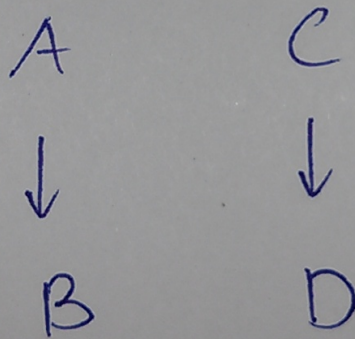
However, we can make the second channel allocation work, by breaking one of the strong dependency. We will do this by having enough buffer space for receiving message A, and break the weak dependency from D to A, ie. $D \nrightarrow A$. This is shown in Figure 3.2.5.1

Another way, but very unlikely, to make the allocation work, is to have the receiving of A uses no resources at all and can be sinked indefinitely (such as a write-back). It's unlikely because the server needs to remember to send out B after receiving A, and thus using some limited resources. Assuming that it is the case though, we have now broken the strong dependency $A \Rightarrow B$ and everything should work. This is shown in Figure 3.2.5.1.

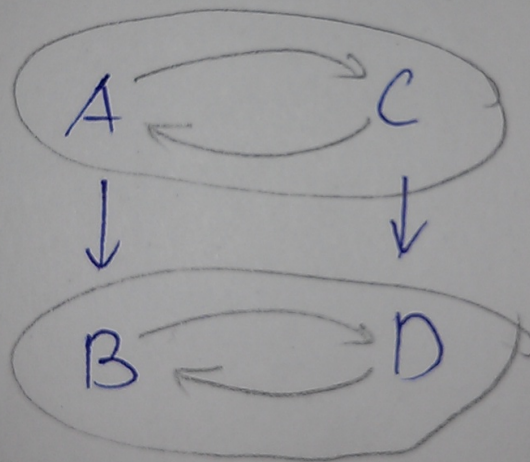
The L1.5pipeline has 4 stages, as follows:

1. S0: Pre-pipeline: MSHR allocation, replacement way conflict check, coherence conflict check, buffer check.
2. S1: Address tag read or write, MESI read

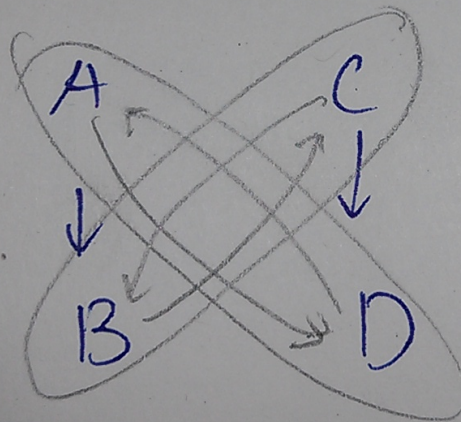
Base dependencies graph



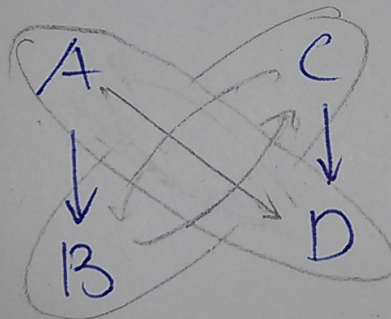
Channel allocation 1

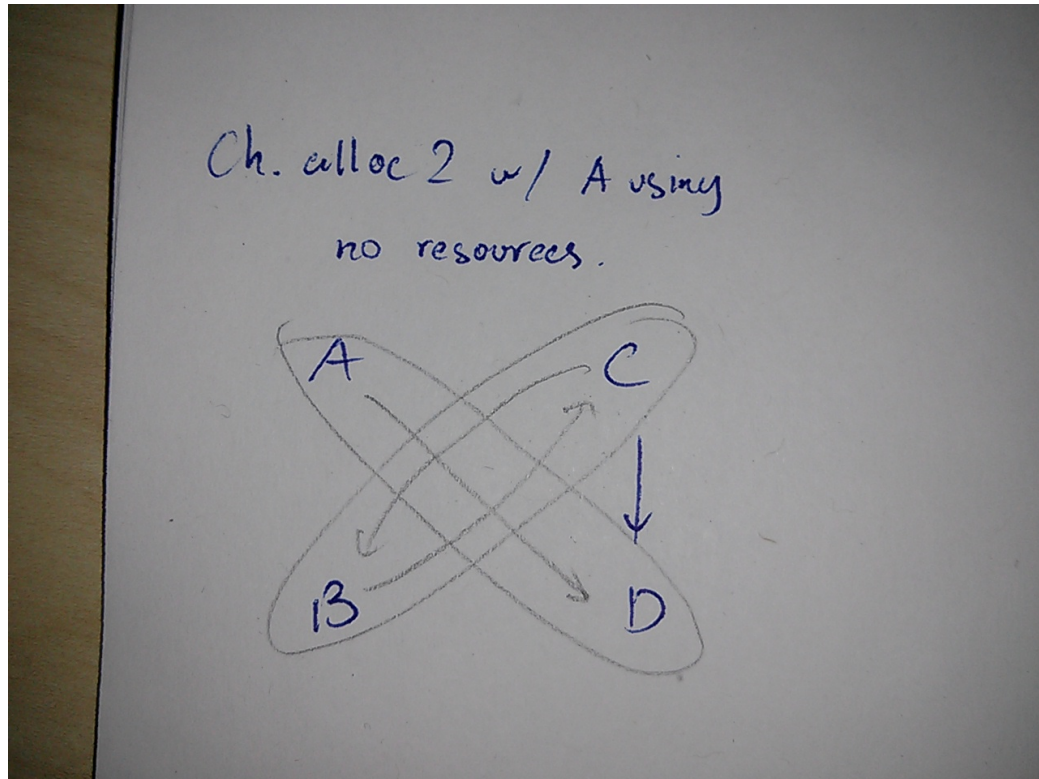


Channel allocation 2



Ch. alloc 2 w/ ^{enough} ~~unlim.~~ buffer for A





3. S2: Data array read or write, MESI write, L1 way-map cache read or write, MSHR deallocation
4. S3: Post-pipeline: CPX sink, Network serializer sink

The pre-pipeline stage is like a reservation station that arbitrates and selects which to run through the pipeline among different sources (NoC1-4 and PCX). It is very important not to let requests that can potentially stall the pipeline be in the pipeline. Such stall sources can be replacement way conflict, coherence conflict (eg. write to an address in state MI), or stall at the network interface.

The post-pipeline stage is a sink that might or might not be buffered. Most likely we will need at least a one-entry buffer to avoid deadlock. Note: the CPX sink does not need any buffer because it is efficiently handled in the core.

The reason that we must be careful with deadlocks in the pipeline is because we do not have any provisions for cancelling and restarting a transaction. It might be feasible to do it but there are a few concerns that must be considered first, like out-of-order transaction after restarting, how to cancel a serializing NoC message, when to restart a NoC message (as the deadlock might be transient), cost of restarting, and so on. It should be easier just to avoid deadlock.

Figure 3.2 structurally describes the pipeline.

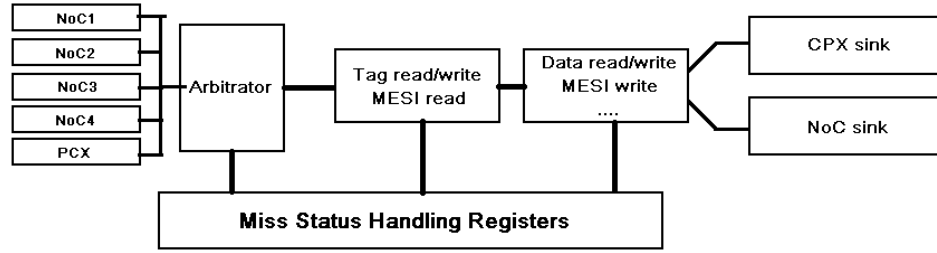


Figure 3.2: Pipeline diagram of the L1.5

3.2.5.2 Pipeline operation detail

Brief list of incoming messages

PCX 16B dcache load, 32B icache load, 16B dcache load NC, 8B store, CAS 1&2, SWP/LDSTUB, block store, block init store

NoC2 FwdERd, FwdEWr, FwdMRd, FwdMWr, Inv (All are operating at the 64B cache line), TODO ****AckWb**** (No buffer)

NoC4 AckDt, AckEDt (at least 32B buffer)

Brief list of outgoing messages

CPX Responses for 16B dcache load, 32B icache load, 16B dcache load NC, 8B store, CAS 1&2, SWP/LDSTUB, block store, block init store

NoC1 Equivalents of incoming PCX messages, write-back requests... (at least 16B buffer)

NoC3 Acknowledgement equivalents to NoC2 requests (at least 64B buffer)

Checking for replacement way conflicts All loads and stores from the core are potentially evicting a cache line when they are processing through the pipe line. The arbitrator must make sure that a subsequent load/store does not also select the same way when mapped to the same index. This means that an incoming load/store need to check its cache index and cache way index against the load/store MSHRs; and if found to be the same, need to stalled in S0.

So the problem with this is that

Checking for coherence conflicts In a similar vein to checking for replacement way conflicts, we can also have loads/stores that reference a cache line in transitioning states that will trigger stalling. There are only a few possibilities:

- Loads that hit a victim Modified in the MI state. This is legal and should be allowed to go.
- Stores that hit a victim Modified in the MI state. This should not be allowed.
- Loads that hit a Shared line in state SM (being upgraded). This cannot happen (load should hit the store buffer).
- Stores that hit a Shared line in state SM (being upgraded). This cannot happen (the previous store is still not acked).

Looking at the state transition table of the L1 controller, I don't see any other possibilities that could result in stalling. Atomics are unlikely to change the picture.

Dealing with non-cacheable loads and stores At this point I am unsure on how to deal with non-cacheable loads and stores. I presume that initiating a NC ld/st means that the data has not been entered to the cache yet, so we will not waste energy to check the address tag for invalidation. Therefore, on a returned NC ld/st, we will also not check the tag; we'll simply bypass the data to the L1.

Another source of NC ld/st returns are the atomics. The assumption is as follows:

- If a CAS/SWP/LDSTUB operation is initiated on a cached data in E/M state, we will do the CAS and write back the data.
- If a CAS/SWP/LDSTUB is done on cached data in S state, we will invalidate the cache line and forward the CAS to L2.
- If a CAS/SWP/LDSTUB is done on cached data in I state, we will forward the CAS to L2.
- If a block store is done on cached data in E/M, we will write to L1.5 then write back to L2.
- If a block store is done on cached data in S, we will invalidate and forward.
- If a block store is done on cached data in I, we will forward.

With no exception, any data returned will not be cached.

Dealing with Fwd requests with multiple cache line states The problem is that while the coherence protocol was written for cache line size 64B, the actual cache line size between L1 and L2 is 16B, therefore we can have impossible cases happen. For example, let's say we have 4 segments A,B,C,D of state M, I, IS, and S***, and the L2 sends a FwdMRd. Naturally, we should write back segment A and ignore segment B, but what do we do with C and D? While not defined, the logical thing to do is also to ignore them.

But what if you have segments with state SM and M? Maybe the best thing to do here is to stall for the WriteReq to be finished and put segment SM to M before the write back.

Note***: the sequence to get this is RdReq D, WrReq A, then RdReq C, with the last transaction still in flight.

Summary of operation latency and occupancy

Chapter 4

L2 Cache

4.1 Overview

The L2 cache is a distributed write-back cache shared by all cores. The default cache size is 64KB per core. It is 4-way set associative and the block size is 64 bytes. A directory array is also integrated with the L2 cache with 64 bits per entry. Therefore, the directory is able to keep track of up to 64 sharers. The L2 cache is inclusive of private L1.5 and L1 caches so every private cache line has a copy in the L2 cache.

Each distributed L2 cache receives input requests from NoC1 and NoC3 and sends output responses to NoC2. The NoC interface is converted from credit-based to val/rdy before connecting to the L2 cache, so the val/rdy interface is used in those I/O ports of L2.

4.2 Architecture Description

4.3 Pipeline Flow

4.4 Instruction Operations

4.5 Special Accesses to L2

I/O addresses starting from 0xA0 to 0xAF are assigned for special accesses to L2 cache. Non-cacheable loads and stores to those addresses are translated into special accesses to L2 based on other bits of the address. All types of special accesses are listed as follows:

0xA0: Diagnostic access to the data array
 0xA1: Diagnostic access to the directory array
 0xA2: Diagnostic access to the shared map cache (SMC)
 0xA3: Coherence flush on a specific cache line
 0xA4: Diagnostic access to the tag array
 0xA5: Flush the shared map cache (SMC)
 0xA6: Diagnostic access to the state array
 0xA7: Access to the coreid register
 0xA8: Access to the error status register
 0xA9: Access to the L2 control register
 0xAA: Access to the L2 access counter
 0xAB: Access to the L2 miss counter
 0xAC, 0xAD, 0xAE, 0xAF: Displacement line flush on a specific address

The detailed formats of different types of special accesses are explained below.

4.5.1 Diagnostic access to the data array

The L2 data array is a 4096x144 SRAM array. Each line contains 128 bits of data and 16 bits of ECC protection bits. Each half of the 128 bits is protected by 8 ECC bits. Each diagnostic load or store can either access 64 bits of data or 8 bits of ECC bits, depending on the 31:30 bit of the address. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA0

31:30 \Rightarrow access op: 2'b00 means data bits, 2'b01 means ECC bits, other values are undefined

29:24 \Rightarrow home node

23:16 \Rightarrow undefined

15:14 \Rightarrow way selection

13:6 \Rightarrow index selection

5:3 \Rightarrow offset selection

2:0 \Rightarrow 3'b000

A stx or ldx instruction can be used to read or write the data array. For accesses to the data bits the entire 64 bits are stored or loaded, while for ECC bits only the lowest 8 bits (7:0) are stored or loaded.

4.5.2 Diagnostic access to the directory array

The L2 directory array is a 1024x64 SRAM array. Each line contains 64 bits of sharers to keep track of up to 64 sharers. Each diagnostic load or store is access on the entire 64 bits. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA1

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:16 \Rightarrow undefined

15:14 \Rightarrow way selection

13:6 \Rightarrow index selection

5:3 \Rightarrow offset selection

2:0 \Rightarrow 3'b000

A stx or ldx instruction can be used to read or write the directory array.

4.5.3 Diagnostic access to the shared map cache (SMC)

The L2 SMC is a 16-entry fully-associative array. Each SMC entry contains 4 global sharer ids, each of which is 30 bits. It is accessed by 16 bits of address composed of a 10-bit sharer clump id and a 6-bit local sharer id. The higher 14 bits of the address are tag bits and the lower 2 bits are offset bits. Therefore, each entry also contains 14 tag bits and 4 valid bits corresponding to those 4 global sharer ids. Each diagnostic load or store can either access 30 bits of data, 14 tag bits or 4 valid bits, depending on the 31:30 bit of the address. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA2

31:30 \Rightarrow access op: 2'b00 means data bits, 2'b01 means valid bits, 2'b10 means tag bits, 2'b11 is undefined

29:24 \Rightarrow home node

23:21 \Rightarrow undefined

20:6 \Rightarrow tag selection

5:4 \Rightarrow offset selection, only used for data bits accesses

3:0 \Rightarrow 4'b0000

A stx or ldx instruction can be used to read or write the data array. For accesses to the data bits only the lowest 30 bits (29:0) are stored or loaded. Similarly for tag or valid accesses only bits 13:0 or 3:0 are considered.

4.5.4 Coherence flush on a specific cache line

This request flushes a L2 cache line in a selected set and way. It also sends out invalidations to flush related L15 cache lines if needed. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA3

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:16 \Rightarrow undefined

15:14 \Rightarrow way selection

13:6 \Rightarrow index selection

5:0 \Rightarrow 6'b000000

A load instructions can be used to flush a L2 line regardless of the data width (ldx, ldub ...). The loaded data has no meaning and will not be checked.

4.5.5 Diagnostic access to the tag array

The L2 tag array is a 256x104 SRAM array. Each line contains 4 tags, each of which is 26 bits. Each diagnostic load or store is access on one tag. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA4

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:16 \Rightarrow undefined

15:14 \Rightarrow way selection

13:6 \Rightarrow index selection

5:0 \Rightarrow 6'b000000

A stx or ldx instruction can be used to read or write the directory array. Only the lowest 26 bits (25:0) are stored or loaded.

4.5.6 Flush the shared map cache (SMC)

The L2 SMC can be flushed in 3 different ways: flushing the entire SMC, clump based flush or flushing a specific pair of clump id and local sharer id. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA5

31:30 \Rightarrow access op: 2'b00 means flushing all, 2'b01 means flushing one id pair, 2'b10 means flushing a clump, 2'b11 is undefined

29:24 \Rightarrow home node

23:21 \Rightarrow undefined

20:6 \Rightarrow tag selection

5:4 \Rightarrow offset selection

3:0 \Rightarrow 4'b0000

A stx instruction can be used to do a flush. The stored data has no meaning and will not be checked.

4.5.7 Diagnostic access to the state array

The L2 state array is a 256x66 SRAM array. Each entry contains state bits for 4 ways, each of which is 15 bits, as well as 6 LRU bits shared by or 4 ways. Each diagnostic load or store can either access 60 bits of state data or 6 bits of LRU bits, depending on the 31:30 bit of the address. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA6

31:30 \Rightarrow access op: 2'b00 means state bits, 2'b01 means LRU bits, other values are undefined

29:24 \Rightarrow home node

23:14 \Rightarrow undefined

13:6 \Rightarrow index selection

5:0 \Rightarrow 6'b000000

A stx or ldx instruction can be used to read or write the data array. For accesses to the state bits the lowest 60 bits (59:0) are stored or loaded, while for ecc bits only the lowest 6 bits (5:0) are stored or loaded.

4.5.8 Access to the coreid register

The coreid register is a 64-bit register in L2 cache. The lowest 34 bits store the node id of the L2 (chipid, x, y and fbits). The higher 30 bits store the maximum number of cores in the system aggregated across multiple chips (in the format of max chipid, max y, max x). The access format of the address is described below.

39:32 \Rightarrow access type: 0xA7

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:0 \Rightarrow undefined

A stx or ldx instruction can be used to read or write the data array. Each diagnostic load or store operates on the entire 64 bits.

4.5.9 Access to the error status register

The error status register is a 64-bit register in L2 cache. The lowest bit indicates the last error is correctable or not. The second bit indicates whether the last error is uncorrectable. The 3rd bit indicates whether there are multiple errors. Bit 14 to 4 stores the line address of the error data array entry. Bit 54 to 15 stores the physical address of the request that causes the error. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA8

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:0 \Rightarrow undefined

A stx or ldx instruction can be used to read or write the data array. Each diagnostic load or store operates on the entire 64 bits.

4.5.10 Access to the L2 control register

The L2 control register is a 64-bit register in L2 cache. The lowest bit is used as the enable bit for clumpy shared memory. The 2nd bit is the enable bit for the error status register. The 3rd bit is the enable bit for l2 access counter and the 4th Bit is the enable bit for l2 miss counter. 32 to 53 are used as the base address for the sharer map table (SMT). Other bits are undefined. The access format of the address is described below.

39:32 \Rightarrow access type: 0xA9

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:0 \Rightarrow undefined

A stx or ldx instruction can be used to read or write the data array. Each diagnostic load or store operates on the entire 64 bits.

4.5.11 Access to the L2 access counter

The L2 access counter stores the total number of L2 accesses. It can be reset by writing all zeros into it. The access format of the address is described below.

39:32 \Rightarrow access type: 0xAA

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:0 \Rightarrow undefined

A stx or ldx instruction can be used to read or write the data array. Each diagnostic load or store operates on the entire 64 bits.

4.5.12 Access to the L2 miss counter

The L2 access counter stores the total number of L2 misses. It can be reset by writing all zeros into it. The access format of the address is described below.

39:32 \Rightarrow access type: 0xAB

31:30 \Rightarrow undefined

29:24 \Rightarrow home node

23:0 \Rightarrow undefined

A stx or ldx instruction can be used to read or write the data array. Each diagnostic load or store operates on the entire 64 bits.

4.5.13 Displacement line flush on a specific address

This request flushes a specific address in the L2 cache. It also sends out invalidations to flush related L15 cache lines if needed. In order to convey both the tag and the index information in the address, actually only the highest 6 bits of the address are used as the access type (this is why the highest 8 bits can be either 0xAC, 0xAD, 0xAE or 0xAF), and the highest 6 bits of the tag are stored in bit 5:0 instead. L2 will

be rearrange the address to be the correct format. The access format of the address is described below.

39:34 \Rightarrow access type: 0b101011

33:14 \Rightarrow lower part of the tag

13:6 \Rightarrow index selection

5:0 \Rightarrow higher part of the tag

A ldub instructions can be used to flush a L2 line. It cannot be replaced by ldx because the offset bits are actually part of the tag so they can be arbitrary value and violate the address alignment requirement for data width larger then one byte. The loaded data has no meaning and will not be checked.

Bibliography

- [1] Sun Microsystems, Santa Clara, CA. OpenSPARC T1 Microarchitecture Specification, 2006.